100%

ONE HUNDRED PERCENT

COMPREHENSIVE
AUTHORITATIVE
WHAT YOU NEED

ONE HUNDRED PERCENT

Master the material for the Oracle8*i* DBA Exam 1Z0-001

Test your knowledge with assessment questions, scenarios, and lab exercises

Practice on state-of-the-art test-preparation software

Oracle8*i*™ DBA: SQL and PL/SQL Certification Bible

Hungry Minds Test Engine powered by

Boson Software

Damir Bersinic, Stephen Giles, Susan Ibach, and Myles Brown

Oracle8*i*™ DBA: SQL and PL/SQL Certification Bible

Oracle8*i*™ DBA: SQL and PL/SQL Certification Bible

Damir Bersinic, Stephen Giles, Susan Ibach, Myles Brown

Best-Selling Books • Digital Downloads • e-Books • Answer Networks • e-Newsletters • Branded Web Sites • e-Learning

New York, NY ✦ Cleveland, OH ✦ Indianapolis, IN

Oracle8*i*™ DBA: SQL and PL/SQL Certification Bible

Published by
Hungry Minds, Inc.
909 Third Avenue
New York, NY 10022
www.hungryminds.com

Library of Congress Control Number: 2001092738

ISBN: 0-7645-4832-8

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

1P/TQ/QY/QR/IN

Distributed in the United States by Hungry Minds, Inc.

Distributed by CDG Books Canada Inc. for Canada; by Transworld Publishers Limited in the United Kingdom; by IDG Norge Books for Norway; by IDG Sweden Books for Sweden; by IDG Books Australia Publishing Corporation Pty. Ltd. for Australia and New Zealand; by TransQuest Publishers Pte Ltd. for Singapore, Malaysia, Thailand, Indonesia, and Hong Kong; by Gotop Information Inc. for Taiwan; by ICG Muse, Inc. for Japan; by Intersoft for South Africa; by Eyrolles for France; by International Thomson Publishing for Germany, Austria, and Switzerland; by Distribuidora Cuspide for Argentina; by LR International for Brazil; by Galileo Libros for Chile; by Ediciones ZETA S.C.R. Ltda. for Peru; by WS Computer Publishing Corporation, Inc., for the Philippines; by Contemporanea de Ediciones for Venezuela; by Express Computer Distributors for the Caribbean and West Indies; by Micronesia Media Distributor, Inc. for Micronesia; by Chips Computadoras S.A. de C.V. for Mexico; by Editorial Norma de Panama S.A. for Panama; by American Bookshops for Finland.

For general information on Hungry Minds' products and services please contact our Customer Care department within the U.S. at 800-762-2974, outside the U.S. at 317-572-3993 or fax 317-572-4002.

For sales inquiries and reseller information, including discounts, premium and bulk quantity sales, and foreign-language translations, please contact our Customer Care department at 800-434-3422, fax 317-572-4002 or write to Hungry Minds, Inc., Attn: Customer Care Department, 10475 Crosspoint Boulevard, Indianapolis, IN 46256.

For information on licensing foreign or domestic rights, please contact our Sub-Rights Customer Care department at 212-884-5000.

For information on using Hungry Minds' products and services in the classroom or for ordering examination copies, please contact our Educational Sales department at 800-434-2086 or fax 317-572-4005.

For press review copies, author interviews, or other publicity information, please contact our Public Relations department at 650-653-7000 or fax 650-653-7500.

For authorization to photocopy items for corporate, personal, or educational use, please contact Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, or fax 978-750-4470.

About the Authors

Damir Bersinic has over 17 years of industry experience working with Oracle, SQL Server, Microsoft Windows NT/2000 and BackOffice, and other advanced products. He is president and founder of Bradley Systems Incorporated, a Microsoft Certified Partner focusing on database, Internet, and system integration consulting and training. He holds a number of industry certifications including Oracle Certified Professional–Database Administrator, Microsoft Certified Systems Engineer (MCSE), Microsoft Certified Database Administrator (MCDBA), Microsoft Certified Trainer (MCT) and Certified Technical Trainer (CTT). His extensive work with Oracle, SQL Server, Windows NT/2000, and BackOffice has enabled him to provide high-level consulting and other assistance to clients in Canada, the United States, and other parts of the globe. He is an avid fan of Formula 1 and CART racing and can be found in front of his large-screen TV on almost all race weekends. In his spare time (a concept that is somewhat foreign to him), he likes to listen to music and enjoys spending time with his wife Visnja and son Anthony. He can be reached via e-mail at `damir@bradsys.com`.

Stephen Giles came to IT through an indirect path. He obtained a master's degree in English from the University of Toronto and held several IT support positions in Toronto until he found his way back to the classroom. He has been a technical trainer since 1999. He currently works for TMI-Learnix in Canada. Stephen brings his support background and a love of databases into the classroom, teaching both Microsoft SQL Server and Oracle development courses. He has also developed a customized SQL course and has contributed to two books on SQL 2000.

Susan Ibach teaches Oracle courses for TMI-Learnix in Ottawa, Canada. After graduating from the University of New Brunswick with a degree in electrical engineering, she worked for a consulting firm. After five years of application programming and development, she became a technical instructor. She teaches SQL, PL/SQL, and Oracle developer courses for TMI-Learnix two weeks a month and spends the rest of her time at home entertaining her son Connor.

Myles Brown is a database programming specialist who currently delivers Oracle certified training for TMI-Learnix. He has a strong background in the design, development, and support of database software solutions in a variety of languages, including COBOL, RPG, PL/SQL, and Visual Basic. Myles has a bachelor of science degree in mathematics (with honors) and a bachelor of education degree from Queen's University. He has been an Oracle Certified Professional since 1999.

Credits

Acquisitions Editor
Terri Varveris

Project Editor
Martin V. Minner

Technical Editor
Harry Liebschutz

Copy Editor
Nancy Crumpton

Project Coordinators
Emily Wichlinski
Regina Snyder

Graphics and Production Specialists
Brian Torwelle
Jeremey Unger
Erin Zeltner

Quality Control Technicians
Laura Albert
Andy Hollandbeck
Carl Pierce

Permissions Editor
Laura Moss

Media Development Specialist
Angela Denny

Media Development Coordinator
Marisa Pearman

Proofreading and Indexing
TECHBOOKS Production Services

I want to dedicate this book to what a colleague of mine refers to as the lightbulbs — that look in the eye of a student or reader when the pieces fall into place. As you begin your journey in Oracle certification with this book, may you be blessed with many lightbulbs.

—Damir Bersinic

Preface

Preparing to become an Oracle Certified Professional in the database administration track is a journey that starts with a single step. The first step is to pass the "Introduction to Oracle: SQL and PL/SQL" exam, which this book is designed to help you accomplish.

About Certifications

In order to become an Oracle Certified Professional (OCP) in the database administrator (DBA) track for Oracle 8*i*, you must pass the following examinations:

- Exam 1Z0-001: Introduction to Oracle: SQL and PL/SQL
- Exam 1Z0-023: Oracle 8*i* Architecture and Administration
- Exam 1Z0-025: Oracle 8*i* Backup and Recovery
- Exam 1Z0-024: Oracle 8*i* Performance and Tuning
- Exam 1Z0-026: Oracle 8*i* Network Administration

The Oracle 8*i* certification path is designed to lead you gradually to the goal of becoming an OCP. You are not required to take the exams in the order they are intended so you do not have to take the exam for which this book prepares you right away. It's acceptable for you to take another one of the exams first if you prefer to gain more experience. To become an Oracle Certified Professional in the DBA track, you must pass all five exams, in no particular order.

For each of the previously listed exams, Oracle provides a set of objectives that you are tested on and required to be familiar with. The objectives for the "Introduction to Oracle: SQLand PL/SQL" exam are listed in Appendix C, as well as a cross-reference of which chapters in this book provide information on that objective. The complete set of objectives for the entire track may be found in the *Oracle Certified Professional Program Candidate Guide, Oracle 8i Certified Database Administrator Track* available on Oracle's Web site at `http://www.oracle.com/education/certification/index.html?ocpguides.html`. The exact link to the Adobe Acrobat PDF file for the DBA track is: `http://www.oracle.com/education/downloads/dba8i_cg.pdf`.

The Candidate Guide document should be reviewed to gain a full understanding of the Oracle Certified Professional Program, as well as how to schedule your exams for your geographical area.

How this Book is Organized

This book is divided into three parts:

- **Part I — The Oracle SQL Language:** This part begins by providing an understanding of an object relational database management system (ORDBMS) and how it can be used. You are then introduced to the Structured Query Language (SQL) and what it is used for, followed by an in-depth discussion of SQL and the use of functions, expressions, and the various types of joins. Finally, we outline how to add, change, and remove data in the database.
- **Part II — Managing Database Objects**: This part of the book is designed to show you how to create and manage objects in the database, as well as to ensure that access is restricted to the appropriate users. This part begins by providing insight into how to make effective use of SQL*Plus, an Oracle utility used to execute commands against the database and format the data returned.
- **Part III — Using PL/SQL:** The goal of this part of the book is to provide a fundamental understanding of the syntax and features of the procedural extensions to SQL introduced by Oracle with PL/SQL. You learn how to create anonymous PL/SQL blocks and to define and make use of cursors and variables. The last chapter of the book also introduces stored program units and how they can be used to add functionality to your databases.

The book has been divided into these three parts from the standpoint of organization and logic and not to satisfy the exam requirements in a one-to-one mapping. Each objective of the exam is covered by material in the book. Appendix C lists the exam objectives and the relevant section of the book where each is covered.

The appendixes of the book provide a number of useful items:

- **Appendix A:** Outlines the programs and other information found on the CD-ROM that accompanies this book, including an electronic copy of the book in PDF format, a sample exam with 300 or so questions that will help you prepare to take the real exam, and many other useful software programs.
- **Appendix B:** Contains a sample exam with the same number of questions found in the actual Oracle exam. The questions in this appendix are designed to closely emulate the type of questions you can expect on the exam and therefore to help to prepare for it. Answers to each question are also provided, along with an explanation of why the correct answer is correct and the others are incorrect.

- **Appendix C:** Provides a mapping of the "Introduction to Oracle: SQL and PL/SQL" exam objectives to the chapters and sections where these objectives are covered in the book. If you find that you are weak in a particular area, this appendix enables you to focus your study on the part of the book containing information you require.
- **Appendix D:** Contains information on how to properly prepare for the exam and how to register to take it.
- **Appendix E:** Lists the objects used by the exercises at the end of each chapter, as well as providing a hard copy of the scripts used to create and populate the objects.
- **Appendix F:** Lists the common Oracle data dictionary views and their structure.
- **Appendix G:** Recommends other books, Web sites, and other resources that may be useful in helping you prepare for the exam and increase your knowledge of relational databases and Oracle.

How to Use this Book

When asked where the best place to start is, a wise individual once said "at the beginning." This truth holds true here. The later parts and chapters of this book are meant to build upon concepts presented in previous chapters. Similarly, the labs of each chapter may also depend upon the results of labs in previous chapters. While this last point is not a hard-and-fast rule, it should be kept in mind. The recommended practice is to read the book from beginning to end and absorb the material in the order presented.

Each chapter, as well, follows a consistent structure. You are first presented with a series of questions to test your knowledge of the material in the chapter. The questions are useful in determining whether you are familiar with the topics about to be presented. The Chapter Pre-Test questions enable you to verify that you have a good grasp of the information.

The main body of the chapter follows the Chapter Pre-Test. You are introduced to concepts and shown examples of code, as appropriate. Feel free to try the code as you are reading the material to verify that you get the same results as the authors. Each chapter ends with a Key Point Summary designed to reinforce the main elements presented in the chapter, and then presents a series of exam-style questions to test your understanding of the material. They are followed by one or more scenario-based questions to further test your understanding of the material. You are then asked to complete Lab Exercises to provide hands-on experience. You are free to do the labs before answering the exam-style Assessment questions.

Answers for the Chapter Pre-Test, Assessment, Scenarios, and Lab Exercises are provided at the end of the chapters. The answers enable you to confirm the results of your work and often provide additional information .

After you have read each chapter, you should use the CD-based exam to further test your knowledge. When you feel that you are ready, you can then try your hand at the Practice Exam in Appendix B. When you feel that you have sufficient information, schedule your exam.

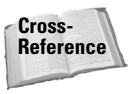
The night before taking the exam, review each chapter's Key Point Summary to make sure you understand the material and answer the Chapter Pre-Test questions again. After passing the exam, the book also is an excellent reference on SQL and PL/SQL, so keep it handy.

Using this book's icons

Watch for the following margin icons to help you get the most out of this book:

Tips provide special information or advice.

Caution icons warn you of a potential problem or error.



This icon directs you to related information in another section or chapter.

Exam Tips offer insider information about the exam.

This icon identifies on-the-job best practices or instances where exam information deviates from actual practice.

Conventions

Whenever command syntax is presented, it appears similar to the following:

```
CREATE [GLOBAL TEMPORARY] TABLE [schema.]tablename
   (columnname datatype [NULL | NOT NULL] [DEFAULT expression]
        [, ...]);
```

- ✦ Elements of the command that are required are presented in uppercase, such as the words CREATE and TABLE in the preceding example.
- ✦ Elements of the command language that are optional are also presented in uppercase and enclosed in square brackets ([and]), such as the [GLOBAL TEMPORARY] option in the preceding example.
- ✦ Elements that are substitution variables for object names in a code example are presented in lowercase, such as "tablename", "columnname", and "datatype" in the preceding example.
- ✦ Elements that are optional substitution variables for object name components, or object names, such as the word "schema" in the preceding example, are all lowercase and enclosed in square brackets.
- ✦ Optional elements of the command syntax always are enclosed in square brackets.
- ✦ Portion of the command syntax where mutually exclusive choices must be made separate those choices by a pipe symbol (|)—for example, the choice between NULL and NOT NULL in the preceding syntax.
- ✦ Elements that can be repeated several times within the code syntax are represented by three dots (...).

Acknowledgments

I want to acknowledge the hard work of my contributing authors. Writing a book was a new experience to some, but you all came through with flying colors. This task was greatly helped by the assistance of the entire Hungry Minds team, who pushed us when we needed to be pushed, but otherwise let us do what we needed to do.

Each of us would also like to acknowledge the contribution our family makes to these types of projects. Their understanding when we need to write but would rather be with them helps make the process less painful. They also know that the smoother the process is for those of us involved in it, the sooner we can spend more of time with them. To our wives, husbands, and children, we are always grateful for your support and understanding, even if we don't show it all the time.

Finally, if you read the snippets of information on the authors contributing to this book, you will note that three out of four work for the same company, TMI-Learnix. A special thank you goes out to Tim Mabey and Mia Hempey who allowed me not only to grab some of their trainers' free time and get them involved with this project, but also actively encouraged it. Working with you folks has always been, and continues to be, a pleasure.

—Damir Bersinic

Contents at a Glance

Preface ix
Acknowledgments xiv

Part I: The Oracle SQL Language 1
Chapter 1: The Oracle Database 3
Chapter 2: Retrieving Data Using Basic SQL Statements 35
Chapter 3: Using Single- and Multi-Row Functions 89
Chapter 4: Advanced SELECT Statements 139
Chapter 5: Adding, Updating, and Deleting Data 191

Part II: Managing Database Objects 225
Chapter 6: The SQL*Plus Environment 227
Chapter 7: Creating and Managing Oracle Database Objects 267
Chapter 8: Configuring Security in Oracle Databases 359

Part III: Using PL/SQL 401
Chapter 9: Introduction to PL/SQL 403
Chapter 10: Controlling Program Execution in PL/SQL 443
Chapter 11: Interacting with the Database Using PL/SQL 479
Chapter 12: Handling Errors and Exceptions in PL/SQL 533
Chapter 13: Introduction to Stored Programs 571

Appendix A: What's on the CD-ROM? 615
Appendix B: Practice Exam 621
Appendix C: Objective Mapping 653
Appendix D: Exam Tips 665
Appendix E: Database Schema for Labs 669
Appendix F: Data Dictionary Views 687
Appendix G: Suggested Readings, Web Sites, and Other Resources 717

Index 721
End-User License Agreement 741
CD-ROM Installation Instructions 744

Contents

Preface. ix

Acknowledgments xiv

Part I: The Oracle SQL Language 1

Chapter 1: The Oracle Database 3

Overview of Database Concepts 5
Relational Database Management System (RDBMS) 6
Object Relational Database Management System (ORDBMS) 9
Database Objects 11
Tables 13
Columns and datatypes 13
Constraints 15
Sequences 17
Views 17
Indexes 18
Synonyms 19
User-defined datatypes 19
Program Units 20
The Oracle Data Dictionary 21
Assessment Questions 25
Scenario 27
Lab Exercise 28
Answers to Chapter Questions 29
Chapter Pre-Test 29
Assessment Questions 31
Scenario 33

Chapter 2: Retrieving Data Using Basic SQL Statements 35

A Quick SQL Overview 37
Data Definition Language statements 38
Data Control Language statements 38
Data Manipulation Language statements 38
The Basic SELECT Statement 40
Arithmetic operations 41
Ordering data in the SELECT statement 55

Limiting Rows Using the WHERE Clause . . . 57
Using logical operators in WHERE clauses . . . 63
Additional comparison operators in the WHERE clause . . . 66
Including the ROWNUM pseudo-column in the WHERE clause . . . 72
Assessment Questions . . . 77
Scenarios . . . 80
Lab Exercises . . . 80
Answers to Chapter Questions . . . 82
Chapter Pre-Test . . . 82
Assessment Questions . . . 83
Scenarios . . . 85
Lab Exercises . . . 86

Chapter 3: Using Single- and Multi-Row Functions . . . 89

Single-Row Functions . . . 91
Conversion functions . . . 91
The TO_DATE function . . . 98
Character functions . . . 99
Number functions . . . 105
Date functions. . . . 107
Additional functions . . . 110
Group/Aggregate Functions . . . 114
Using the GROUP BY Clause . . . 119
Using the HAVING Clause . . . 122
Assessment Questions . . . 126
Scenarios . . . 129
Lab Exercises . . . 129
Answers to Chapter Questions . . . 131
Chapter Pre-Test . . . 131
Assessment Questions . . . 132
Scenarios . . . 133
Lab Exercises . . . 135

Chapter 4: Advanced SELECT Statements . . . 139

Working with Joins . . . 141
Working with equijoins . . . 142
Cross-joins . . . 145
Nonequijoins . . . 146
Outer joins . . . 148
Self-joins . . . 149
Working with Subqueries . . . 151
Working with basic subqueries . . . 151
Working with inline views . . . 157
Working with correlated subqueries . . . 159

Working with SET Operators . . . 161
Using the UNION operator . . . 162
Using the UNION ALL operator . . . 165
Using the INTERSECT operator . . . 165
Using the MINUS operator . . . 166
Using Hierarchical Queries . . . 166
Using Substitution Variables . . . 172
The ROWID pseudo-column . . . 176
Assessment Questions . . . 179
Scenarios . . . 182
Lab Exercises . . . 183
Answers to Chapter Questions . . . 185
Chapter Pre-Test . . . 185
Assessment Questions . . . 185
Scenarios . . . 187
Lab Exercises . . . 188

Chapter 5: Adding, Updating, and Deleting Data . . . 191

DML Statements: Inserting Data into Tables . . . 193
Using the column list . . . 195
Inserting values using additional language elements . . . 196
DML Statements: Modifying Existing Data . . . 198
DML Statements: Removing Data from Tables . . . 199
Using subqueries in DML statements . . . 200
How Oracle Processes DML Statements . . . 203
Controlling Transactions . . . 205
The ACID test . . . 206
Transaction control statements . . . 207
Controlling concurrent operations with locking . . . 210
Assessment Questions . . . 213
Scenarios . . . 216
Lab Exercises . . . 217
Answers to Chapter Questions . . . 220
Chapter Pre-Test . . . 220
Assessment Questions . . . 221
Scenarios . . . 222
Lab Exercises . . . 223

Part II: Managing Database Objects 225

Chapter 6: The SQL*Plus Environment . . . 227

The SQL Buffer . . . 229
Defining Variables . . . 231
Substitution variables . . . 231
DEFINE . . . 234
ACCEPT . . . 235
Bind variables . . . 236

SQL*Plus Commands . . . 237
DESCRIBE . . . 237
SAVE . . . 237
EDIT . . . 238
GET . . . 238
START . . . 239
@ . . . 239
SPOOL . . . 239
EXIT . . . 240
Customizing SQL*Plus with SET Commands . . . 240
ARRAYSIZE . . . 241
COLSEP . . . 241
FEEDBACK . . . 242
HEADING . . . 243
LINESIZE . . . 243
LONG . . . 244
PAGESIZE . . . 244
PAUSE . . . 244
TERMOUT . . . 245
Formatting Output with SQL*Plus . . . 245
Headers and footers . . . 245
COLUMN . . . 247
BREAK . . . 250
COMPUTE . . . 251
Saving Environment Settings . . . 252
Scripts . . . 252
PRODUCT_USER_PROFILE . . . 254
Assessment Questions . . . 257
Scenarios . . . 259
Lab Exercises . . . 260
Answers to Chapter Questions . . . 261
Chapter Pre-Test . . . 261
Assessment Questions . . . 262
Scenarios . . . 262
Lab Exercises . . . 263

Chapter 7: Creating and Managing Oracle Database Objects 267

The Ground Rules for Creating Objects . . . 269
Data Definition Language (DDL) . . . 269
Oracle naming conventions . . . 270
Fully qualified object names . . . 270
Creating and Managing Tables . . . 272
The CREATE TABLE statement . . . 273
The ALTER TABLE command . . . 283
The DROP TABLE command . . . 288
The TRUNCATE TABLE command . . . 289
Documenting tables and columns . . . 289

Data Integrity Using Constraints . . . 291
Naming constraints . . . 293
Defining constraints . . . 293
A little more about constraint types . . . 295
Managing constraints . . . 300
Creating Other Database Objects . . . 308
Views . . . 309
Indexes . . . 318
Sequences . . . 325
Synonyms . . . 333
Assessment Questions . . . 339
Scenarios . . . 342
Lab Exercises . . . 343
Answers to Chapter Questions . . . 345
Chapter Pre-Test . . . 345
Assessment Questions . . . 347
Scenarios . . . 349
Lab Exercises . . . 351

Chapter 8: Configuring Security in Oracle Databases . . . 359

Users and Schemas . . . 361
Creating and managing users . . . 362
Granting and Administering User Privileges . . . 366
System privileges . . . 366
Object privileges . . . 372
Roles . . . 377
Creating and granting roles . . . 378
Determining privileges and roles granted . . . 380
Revoking roles . . . 381
Assessment Questions . . . 384
Scenarios . . . 387
Lab Exercises . . . 387
Answers to Chapter Questions . . . 389
Chapter Pre-Test . . . 389
Assessment Questions . . . 390
Scenarios . . . 391
Lab Exercises . . . 392

Part III: Using PL/SQL 401

Chapter 9: Introduction to PL/SQL . . . 403

Uses and Benefits of PL/SQL . . . 405
Modularity . . . 405
Variables . . . 406
Control structures . . . 406

Superior performance . . . 406
Error handling . . . 406
Support for SQL . . . 406
Portability . . . 407
The PL/SQL Engine and Statement Processing . . . 407
Types of PL/SQL blocks . . . 408
Block structure . . . 409
Comments . . . 412
Variables . . . 413
Scalar variables . . . 414
Bind variables . . . 422
Composite datatypes . . . 423
User-defined types . . . 426
Reference types . . . 427
Executing and Testing PL/SQL Blocks . . . 429
Print command . . . 430
DBMS_OUTPUT package . . . 430
Assessment Questions . . . 433
Scenarios . . . 435
Lab Exercises . . . 436
Answers to Chapter Questions . . . 437
Chapter Pre-Test . . . 437
Assessment Questions . . . 438
Scenarios . . . 439
Lab Exercises . . . 440

Chapter 10: Controlling Program Execution in PL/SQL . . . 443

Loops . . . 445
Basic loop . . . 445
WHILE loop . . . 447
FOR loop . . . 447
Nested loops and labels . . . 451
Conditional Processing . . . 455
IF . . . THEN . . . 455
ELSE . . . 456
ELSIF . . . 457
Nested Blocks . . . 459
Transaction Control . . . 461
Assessment Questions . . . 468
Scenarios . . . 473
Lab Exercise . . . 473
Answers to Chapter Questions . . . 474
Chapter Pre-Test . . . 474
Assessment Questions . . . 475
Scenarios . . . 476
Lab Exercise . . . 477

Chapter 11: Interacting with the Database Using PL/SQL 479

SQL Statements 481
SELECT 481
INSERT 484
UPDATE 486
DELETE 487
Implicit cursors 487
Explicit Cursors 489
Declaring explicit cursors 489
Opening explicit cursors 490
Closing an explicit cursor 491
Fetching from explicit cursors 492
%ROWTYPE 494
The cursor FOR loop 495
Cursor variables 497
Cursors with subqueries 499
Cursor parameters 500
FOR UPDATE and WHERE CURRENT OF 501
Composite and Collection Datatypes 504
PL/SQL records 504
Index-by tables 506
Tables of records 510
Nested tables 511
VARRAYs 513
Collection methods 515
Assessment Questions 519
Scenarios 522
Lab Exercises 523
Answers to Chapter Questions 524
Chapter Pre-Test 524
Assessment Questions 525
Scenarios 526
Lab Exercises 528

Chapter 12: Handling Errors and Exceptions in PL/SQL 533

Types of Errors 535
Compile errors 535
Program logic errors 537
Runtime errors or exceptions 539
Exception Handling 539
Predefined exceptions 541
Non-predefined exceptions 544
User-defined exceptions 546
WHEN OTHERS Clause 548
Error Propagation 549
Coding Conventions 555
Assessment Questions 558

Scenarios . . . 563
Lab Exercises . . . 563
Answers to Chapter Questions . . . 565
Chapter Pre-Test . . . 565
Assessment Questions . . . 566
Scenarios . . . 567
Lab Exercises . . . 567

Chapter 13: Introduction to Stored Programs . . . 571

Subprograms . . . 573
Procedures . . . 574
Client-side procedures . . . 575
Server-side procedures . . . 575
Parameters . . . 576
Nesting . . . 579
Deleting Procedures . . . 580
User-Defined Functions . . . 580
Deleting Functions . . . 582
Packages . . . 583
Specification . . . 583
Package body . . . 584
Accessing programs and variables in packages . . . 586
Removing Packages . . . 587
Listing Package contents . . . 588
Triggers . . . 588
Triggering events . . . 589
Statement-level triggers . . . 589
Row-level triggers . . . 591
Trigger predicates . . . 594
Restrictions on triggers . . . 595
Order of firing . . . 596
INSTEAD OF triggers . . . 597
Event triggers . . . 598
Enabling and disabling triggers . . . 599
Removing triggers . . . 600
Data Dictionary Views . . . 600
USER_SOURCE . . . 600
USER_TRIGGERS . . . 601
USER_OBJECTS . . . 602
Assessment Questions . . . 605
Scenarios . . . 607
Lab Exercises . . . 608
Answers to Chapter Questions . . . 609
Chapter Pre-Test . . . 609
Assessment Questions . . . 610
Scenarios . . . 610
Lab Exercises . . . 611

Appendix A: What's on the CD-ROM? 615

Appendix B: Practice Exam 621

Appendix C: Objective Mapping 653

Appendix D: Exam Tips 665

Appendix E: Database Schema for Labs 669

Appendix F: Data Dictionary Views 687

Appendix G: Suggested Readings, Web Sites, and Other Resources 717

Index 721

End-User License Agreement. 741

CD-ROM Installation Instructions 744

PART I

The Oracle SQL Language

This part of the book deals with the SQL language used by Oracle to retrieve, add, change, and delete data in your database.

Chapter 1 introduces the general concepts of a database and defines the terms *relational database management system* (RDBMS) and *object relational database management system* (ORDBMS), and how Oracle satisfies the requirements for each. The chapter then introduces the database objects that can be created in Oracle 8*i* and their purposes. Next, we briefly discuss the Oracle data dictionary and introduce the common data dictionary views, which are helpful in getting information on the objects that exist in the database.

Chapter 2 provides information on the Structured Query Language (SQL)—what it is and how it is used. You learn how to formulate and execute a basic SELECT statement that can be used to retrieve data from the database. We discuss different operators that are available in Oracle, as well as how to deal with concatenation, NULLs, and aliasing. The chapter provides a discussion of how to limit the data returned using the WHERE clause, and how to sort the resulting rows using the ORDER BY clause of the SELECT statement. Finally, we introduce the DUAL table and describe how to use it to test operations.

Chapter 3 expands upon the information presented in the previous chapter by introducing single- and multi-row functions. We describe the single-row functions that are available in Oracle 8*i* and how they are used, along with examples. We show how to perform calculations on a set of rows and return a single value using multi-row functions. We also discuss the use of the GROUP BY and HAVING clauses to perform aggregation on sets of data for defined columns. Finally, we explain other functions and pseudo-columns available in Oracle.

In This Part

Chapter 1
The Oracle Database

Chapter 2
Retrieving Data Using Basic SQL Functions

Chapter 3
Using Single- and Multi-Row Functions

Chapter 4
Advanced SELECT Statements

Chapter 5
Adding, Updating, and Deleting Data

Chapter 4 builds upon the previous two chapters to discuss advanced SELECT statement topics including how to join data from two or more tables in a single resultset. We discuss the different types of joins available in Oracle and provide examples of their usage, along with an explanation of how to vary the data retrieved through the use of runtime substitution variables in SQL*Plus. We cover such complex topics as hierarchical queries and the use of set operators. The chapter provides a lengthy discussion on the different types and uses of subqueries in Oracle, and why they are necessary, as well as their impact on performance.

After the data retrieval information presented in the previous three chapters, Chapter 5 shows you how to get data in the database using the INSERT statements, as well as how to change or remove it using the UPDATE and DELETE statements. We discuss transaction control and the use of COMMIT, ROLLBACK, and SAVEPOINT, as well as what constitutes a transaction.

CHAPTER 1

The Oracle Database

EXAM OBJECTIVES

- Overview of relational databases, SQL and PL/SQL
  - Discuss the theoretical and physical aspects of a relational database
  - Describe the Oracle implementation of RDBMS and ORDBMS
  - Describe the use and benefits of PL/SQL
- Creating and managing tables
  - Describe the main database objects
  - Describe the datatypes that can be used when specifying column definitions
- Including constraints
  - Describe constraints
- Creating views
  - Describe a view
- Oracle data dictionary
  - Describe the data dictionary views a user may access
  - Query data from the data dictionary
- Other database objects
  - Describe database objects and their uses

CHAPTER PRE-TEST

1. What is the key difference between a relational database management system and database systems that came before it?
2. What are the required attributes that all columns must have?
3. Why would you specify a NOT NULL constraint for a column?
4. What makes Oracle an ORDBMS?
5. What is the difference between a PRIMARY and UNIQUE constraint?
6. How many FOREIGN KEY constraints can you define on a single table?
7. What are the benefits of sequences?
8. Name three types of stored subprograms that can be created in Oracle.
9. What are some benefits to using views?
10. What is the effect of indexes on database queries? Database updates?
11. What is the difference between a trigger and a constraint?

✦ Answers to these questions can be found at the end of the chapter. ✦

The first step in understanding how to use Oracle is to understand what a database is and how Oracle satisfies the requirements of a relational database management system (RDBMS). After this, you need to be aware of how Oracle's implementation of an RDBMS, and its extensions to include support for objects, make it an Object RDBMS or ORDBMS. Finally, the basics of database objects and the Structured Query Language (SQL) used to manipulate database objects must be appreciated to be able to work with Oracle on a daily basis.

This chapter covers the theoretical, as well as physical and logical, characteristics of Oracle and introduces database objects, datatypes, and the Oracle implementation of the SQL language. The rest of this book expands upon the basic concepts covered here and provides more detail on how to make effective use of Oracle at the most basic level.

Overview of Database Concepts

In one form or another, databases have existed throughout history. They may not have had the form we expect today — that of a computer program — but as long as data had to be stored, there was always a method of storing them. In fact, you probably have databases in existence in your home or office that you don't refer to as such, but they perform the basic function — the storage of some form of data.

When you open a file cabinet and take out a folder, you are accessing a database. The content of the file folder is your data (for example, your credit card statements, your bank statements, invoices, purchase orders, and so on). The file cabinet and drawers are your data storage mechanisms. Before the advent of computers, all data was stored in some easily recognizable physical form. The introduction of computers simply moved the data from a physical form that you can touch and feel to a digital form that is represented by a series of 1s and 0s in a binary format. Does the information that you display for a purchase order on the screen differ greatly from the same information in the hard copy version of the purchase order? Sure, the way the information is presented is not the same as on the screen, but the key critical elements — who the PO is issued to, the terms, and the purchase order items — are all the same.

In looking at a database and its most basic set of characteristics, the following hold true:

- ✦ A database stores data. The storage of data can take a physical form, such as a filing cabinet or a shoebox.
- ✦ Data is composed of logical units of information that have some form of connection to each other. For example, a genealogical database stores information on people as they are related to each other (parents, children, and so on).

- ✦ A database management system (DBMS) provides a method to easily retrieve, add, modify, or remove data. This can be a series of filing cabinets that are properly indexed, making it easy to find and change what you need, or a computer program that performs the same function.

When data began to be moved from a physical form to a digital form using computers, several different incarnations of systems to manage the data evolved. Some of the more common types of database management systems in use over the last 50 years include the *hierarchical*, *network*, and *relational*. Oracle is a relational database management system (RDBMS).

Relational Database Management System (RDBMS)

The relational model for database management systems was proposed in 1970 by E.F. Codd in a paper called "A Relational Model of Data for Large Shared Data Banks." For its time, it was a radical departure from established principles because it stated that it was not necessary for tables that have data that is related among them to know where the related information was physically stored. Unlike previous database models, including the hierarchical and network models, which used the physical location of a record to relate information between two sets of data, the relational model for databases stated that data in one table needed to know only the name of the other table and the value on which it is related. It was not necessary for data in one table to keep track of the physical storage location of the related information in another.

The relational model broke down all data into collections of objects or relations (that is, tables) that store the actual data. It also introduced a set of operators to act on the related objects to produce other objects (that is, join conditions to produce a new result set). Finally, the model proposed that a set of elements (that is, constraints) should exist to ensure data integrity so that the data would be consistent and accurate. Codd proposed 12 rules that enable you to determine if the database management system satisfied the requirements of the relational model. Although no database today satisfies all 12 rules, it is generally accepted that any RDBMS should comply with most of them.

The essence of the relational model is that data is made up of a set of relations. These relations are implemented as two-dimensional tables with rows and columns as shown in Figure 1-1. In this example, the Students table stores information about students such as their student IDs, their names, and the name of the courses the students are taking. The Courses table stores information about the courses including the course IDs, the course names, and the start and end dates. The CourseID column in both tables provides the relationship between the two tables and is the source of the relation. The tables themselves are stored in a database that resides on a computer. One thing that is not needed to be known is the physical location of the tables — only their names.

Relational database theory

For more information on the theory of databases and the relational model in particular, see E.F. Codd, *The Relational Model for Database Management Version 2*, Addison-Wesley, 1999. Another good reference is C.J. Date, *An Introduction to Database Systems, 7th Edition*, Addison-Wesley, 1999. The first book is out of print and may be hard to find, while the second is a good introduction to the theory of relational databases and is still available.

You should be aware that these are academic texts and are written for that audience. The language might be a bit hard to follow and, in the case of the E.F. Codd book in particular, be less exciting than you may desire. However, both of these texts are worthwhile in providing a basis for the understanding of how a RDBMS functions at its core.

A third book, which provides more of a historical discussion on the evolution of relational database theory and E.F. Codd's contribution to it (and also explains why some of the quirky things in relational databases are there) is C.J. Date, *The Database Relational Model: A Retrospective Review and Analysis*, Addison-Wesley, 2001.

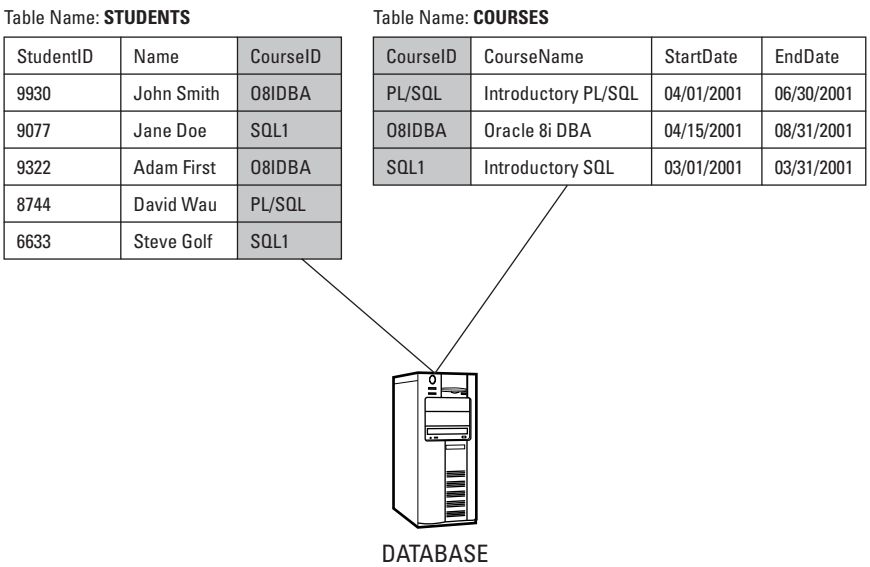


Figure 1-1: A relational database is made up of two-dimensional relations, or tables.

For a database to be considered relational, and because the physical location of rows is not something that users querying data need to know, the table must enable each row to be uniquely identified. The column (or set of columns) that uniquely identifies a row is known as the *primary key*. Each table in a relational database (according to database theory) must have a primary key. In this way, you are certain that the specific value appears only once in the table. In Figure 1-1, the

StudentID column of the Students table is a primary key, and it ensures that each StudentID appears only once in the table. For the Courses table, the CourseID is the primary key.

When relating tables (the whole point of a relational database), the value of a primary key column in one table can be placed in a column in another table. The column in the second table holding the value is known as the *foreign key*. A foreign key states that the value in this column for a row exists in another table and must continue to exist; otherwise, the relationship is broken. In Figure 1-1, the CourseID column in the Students table is a foreign key to the CourseID column in the Courses table. In order for the relationship to be valid, any value placed in the CourseID column of the Students table must already exist in the CourseID column of the Courses table. In other words, in order for a student to take a course, the course must be offered by the institution the student is attending. If the course is not offered, the student cannot take it.

Oracle enforces the primary key/foreign key relationship through the use of constraints, which are discussed later in this chapter.

All of the relations in a relational database are managed by a relational database management system. As indicated earlier, an RDBMS enables you to manipulate relational tables and their contents. It provides a language that enables you to create, modify, and remove objects in the database, as well as add, change, and delete data. The language that Oracle uses is the Structured Query Language, or SQL.

SQL is actually a collection of several different "languages" designed for a particular purpose. It is made up of the following:

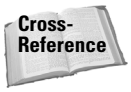
- **Data Definition Language (DDL):** DDL is used to create and modify database objects. Commands used include CREATE, ALTER, DROP, RENAME, and TRUNCATE. When you need to add a new table to the database, you use the CREATE TABLE statement to perform that task. To remove an index, you use DROP INDEX, and so on.

The use of DDL and the syntax to create, alter, and drop database objects is covered in Chapter 7, "Creating and Managing Oracle Database Objects."

- **Data Manipulation Language (DML):** DML is used to modify data in tables in the database. Commands used include INSERT, UPDATE, and DELETE, as well as extensions to control transactions in the database including COMMIT, ROLLBACK, and SAVEPOINT. The SELECT statement used to query data in the database is not technically considered a DML command, although it is sometimes included with the definition of DML because it deals with the retrieval of data.

The use of DML, how to use the SELECT command to query the database, and how to control transactions in Oracle are covered in Chapters 2 through 5.

- **Data Control Language (DCL):** DCL is used to grant and revoke privileges to allow users to perform database tasks and manipulate database objects. Commands used include GRANT and REVOKE. Permissions can be granted to enable a user to perform a task, such as create a table, or to manipulate or query data, such as insert into a table in the database.



The use of DCL, including statement- and object-level security, is covered in Chapter 8, "Configuring Security in Oracle Databases."

Another characteristic of an RDBMS is that tables in a relational database do not have the relationship between them represented by data in one table having a physical location of the data in a related table. As you can see in Figure 1-1, the Students table and the Courses table are related by the data that exists in the CourseID column of both tables. The physical location on disk of each table does not factor in the relationship between them. As long as a user querying the two tables knows the column that relates them, he or she is able to formulate a SQL statement that extracts the data that satisfies the condition of that relationship (also known as a *join condition*). Should one of the tables be moved to a different hard disk used to store data in the database, the relationship still holds true.

A third characteristic of an RDBMS is that the language used to manipulate the database has a rich and varied set of operators that can be used to manipulate the data and explore the relationship between the various tables. The SQL language enables you to determine, through the proper use of operators, data that is related between tables, data where the relationship does not hold true, and much more. The SQL language does not, however, have any elements of a programming language such as loops, conditional logic, and the use of variables. Oracle has extended SQL to include these elements through PL/SQL, a proprietary set of language elements that can be used to create stored procedures, triggers, and other subprograms.

RDBMSes became popular because of the previously mentioned characteristics. However, nothing stays static for long, so Oracle, a relational database since its introduction, has now expanded the definition of what it means to include objects.

Object Relational Database Management System (ORDBMS)

Releases of Oracle prior to Oracle 8 were RDBMSes; that is, they followed the relational model and complied with its requirements, and often improved upon them. With the introduction of Oracle 8, Oracle is now considered an Object Relational Database Management System. An ORDBMS complies with the relational model but also extends it to support the newer object relational database model introduced in the 1980s.

An ORDBMS is characterized by a number of additional features, including the following:

- ✦ Support for user-defined datatypes. This means that users can create their own datatypes based upon the standard Oracle datatypes or other user-defined datatypes. This enables more accurate mapping of business objects to database features and can reduce the time it takes to maintain databases after they have been implemented.
- ✦ The ability to attach methods to objects. A method is a piece of PL/SQL code that performs a particular action. For example, a method could be attached to an OrderItems object that would calculate the extended price for each item in an order.
- ✦ Support for multimedia and other large objects. Oracle 8 and subsequent releases have full support for binary large objects or BLOBs. This means that it is now possible to store large amounts of information, such as video clips, images, and large amounts of text, in the column of a row. Even though earlier releases of Oracle had a feature that enabled this to happen, it lacked functionality and was not implemented in a way that allowed it to conform to object relational standards. The current implementation is much improved.
- ✦ Full compatibility with relational database concepts. Even though object extensions have been added to Oracle 8, in order for it to be called an ORDBMS, it still has to conform to the requirements of an RDBMS. Because of Oracle's strong legacy as an RDBMS, object features enhance the capabilities of the relational part of Oracle and do not replace it.

If you had to state one feature that defines Oracle as an ORDBMS, it is its capability to enable you to create a user-defined datatype, which becomes an object in Oracle. For example, if you want to use a common definition for a telephone number in several tables (Students, Instructors, Employees, and so on) and want to be sure that whenever its characteristics change, that change is inherited by all tables using it, you can create a new datatype "PhoneNumber" with the proper characteristics, and then create the tables using the PhoneNumber datatype as one of the column definitions. If the rules for area codes, for example, change, you can modify the attributes and methods of the PhoneNumber datatype, and all tables will inherit the change.

Exam Tip This exam does not test your knowledge or ability to define and make use of objects within Oracle. You simply need to be aware that Oracle supports objects and that they can be used to create user-defined datatypes, and support multimedia, images, and other large objects.

Database Objects

Every RDBMS needs to support a minimum number of database objects in order to comply with the basic requirements for a relational database. Oracle supports the minimum and many more.

The basic set of functionality that all RDBMSes must adhere to is defined in a standard called the ANSI SQL-92 standard. The set of features for the SQL language that are specified in this standard outline what every relational database must be able to do. Database vendors, such as Oracle, are free to enhance their product beyond the standard, and all of them do to provide greater functionality. Oracle adheres to the SQL-92 standard.

An updated standard, called the SQL-99 standard, includes a richer set of features and changes to the syntax for some SQL operations. Oracle 8*i* does not adhere to the SQL-99 standard, but it is expected that Oracle 9*i* will.

Oracle's collection of database objects includes all of those that are needed for it to be called a relational database (tables, views, constraints, and so on) as well as others that go beyond what is required and are included because they provide additional functionality (packages, object types, synonyms, sequences, and so on). The full list of database objects that Oracle supports is provided in Table 1-1.

Table 1-1
Database Objects in Oracle 8*i*

Object	***Description***
Table	A collection of columns and rows representing a single entity (for example, students, courses, instructors, and so on).
Column	A single attribute of an entity stored in a table. A column has a name and a datatype. A table may have, and typically does have, more than one column as part of its definition.
Row	A single instance of an entity in a table including all columns. For example, a student row stores all information about a single student such as the ID, name, address, and so on.
Constraint	Database objects that are used to enforce simple business rules and database integrity. Examples of constraints are PRIMARY KEY, FOREIGN KEY, NOT NULL, and CHECK.
View	Views are a logical projection of data from one or more tables as represented by a SQL statement stored in the database. Views are used to simplify complex and repetitive SQL statements by assigning those statements a name in the database.

Continued

Table 1-1 *(continued)*

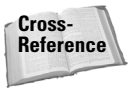
Object	*Description*
Index	Indexes are database objects that help speed up retrieval of data by storing logical pointers to specific key values. By scanning the index, which is organized in either ascending or descending order according to the key value, you are able to retrieve a row quicker than by scanning all rows in a table.
Sequence	Sequences enable you to create and increment a counter that can be used to generate numerical values to be used as primary key values for a table.
Synonym	As in the English language, a synonym is another name for an existing object. Synonyms are used in Oracle as shorthand for objects with long names, or to make it easier to remember a specific object.
Stored Procedure	A collection of SQL and PL/SQL statements that perform a specific task such as insert a row into a table, update data, and so on.
Trigger	A special kind of stored procedure that cannot be invoked manually but rather are automatically invoked whenever an action is performed on a table. Triggers are always associated with a table and a corresponding action such as INSERT, UPDATE, or DELETE.
Functions	Functions are stored programs that must return a value. Unlike stored procedures, which can have parameters passed to them and do not need to return any value as output, a function must return a value.
Packages	Packages are a collection of stored procedures and functions grouped under a common name. This allows you to logically group all program elements for a particular part of the database under a single name for maintenance and performance reasons.
User-Defined Datatype	User-defined datatypes are database objects that can be used in any table or another object definition. Using user-defined datatypes allows you to ensure consistency between tables and also lets you apply methods (i.e., actions that can be performed by the object) as part of the definition.
LOB	LOBs are binary large objects used to store video, images, and large amounts of text. They are defined as a column in a table.

Oracle also includes other objects that are beyond the scope of this book, including clusters, index-organized tables, partitions, and subpartitions. These objects are created by the database administrator (DBA) to ensure the efficient storage and organization of data in the database.

The creation and maintenance of many of the database objects outlined in Table 1-1 is covered in Chapter 7. The creation and management of stored procedures, triggers, functions, and packages is covered in Chapter 13, "Introduction to Stored Programs."

Although Table 1-1 lists almost all database objects available in Oracle, only your understanding of those covered in Chapters 2 through 12 are tested on the exam. Information about other objects is primarily included here for completeness.

Each object in an Oracle database is owned by a user. A user defined in an Oracle database does not have to own any objects, but those that do are known as *schema users*. A schema is a collection of all objects owned by a particular user, including tables, indexes, and views.



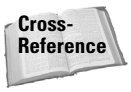
For more information on how to create and maintain users, see Chapter 8.

Tables

The basic element of any database is a table. It is used to store the data and is defined as a collection of rows and columns. A table always represents a single entity (for example, students, courses, instructors, cars, customers, orders, and so on).

The columns in a table are the attributes of the entity that it stores. For example, a training center must store certain types of information about each student such as the name of the student, the address, telephone number, email address, and maybe an enrollment date. All of these things that must be known about each student then become columns in the Students table. Similarly, the courses a training center offers also must have a name, description, retail price, and so on. These then become columns in the Courses table.

Because it is unlikely that a training center would offer only a single course or have only a single student enrolled, a table has the potential to contain information about many students or courses. The information about each registered student becomes a row in the table, as does the information about each offered course. The table is also sometimes referred to as an *entity set*—a collection of information about a set of entities.



The syntax and options for creating tables are outlined in detail in Chapter 7.

Columns and datatypes

Each column in an Oracle table must have at least two properties: It must have a name unique within the table and a valid datatype. Datatypes in Oracle enable you to specify what types of data can be stored in the column (for example, numbers, characters, dates, and BLOBs). The datatypes that are available include the standard Oracle scalar datatypes listed in Table 1-2 or a user-defined datatype that is based upon one of the scalar datatypes or another user-defined datatype.

The "Introduction to Oracle: SQL & PL/SQL" exam does not test your knowledge of the creation or usage of user-defined datatypes.

Table 1-2
Scalar Datatypes

Datatype	*Description*
VARCHAR2(size)	Variable-length character string having a maximum length of 4,000 bytes and a minimum length of 1 byte. You must specify the size for VARCHAR2.
NVARCHAR2(size)	Variable-length character string whose data is stored in the National Language character set of the database having a maximum length of 4,000 bytes and a minimum length of 1 character (which may be 1 or 2 bytes, depending on the National Character set). You must specify the size for NVARCHAR2.
NUMBER(p,s)	Number having precision *p* and scale *s*. The precision p can range from 1 to 38. The scale s can range from –84 to 127. Precision determines how many numbers total can be stored. Scale determines how many decimal places are allowed.
LONG	Character data of variable length up to 2 gigabytes (GB). Previous versions of Oracle used LONG datatype columns to support large character data, but they should no longer be used and are strongly discouraged. Character large object (CLOB) and NCLOB datatype columns are the recommended way to store large amounts of character data.
DATE	Valid date range from January 1, 4712 BC to December 31, 9999 AD.
RAW(size)	Raw binary data of a miximum size of 2,000 bytes and a minimum size of 1 byte. You must specify the size for a RAW value. This datatype is included for backward compatibility and should not be used.
LONG RAW	Raw binary data of variable length up to 2GB. This datatype is included for backward compatibility and should not be used. To store large amounts of binary data, BLOB datatype columns are recommended.
ROWID	Hexadecimal string representing the unique address of a row in its table. This data type is primarily for values returned by the ROWID pseudo-column.
UROWID [(size)]	Hexadecimal string representing the logical address of a row of an index-organized table. The optional size is the size of a column of type UROWID. The maximum size and default is 4,000 bytes.
CHAR(size)	Fixed-length character data with a maximum size of 2,000 bytes. The default and minimum size is 1 byte.

Datatype	*Description*
NCHAR(size)	Fixed-length character data of whose data is stored according to the National Language character set of the database. The maximum size is determined by the number of bytes required to store each character, with an upper limit of 2,000 bytes. The default and minimum size is 1 character or 1 byte, depending on the character set.
CLOB	A character large object containing single-byte characters. Both fixed-width and variable-width character sets are supported, both using the database character set. The maximum size is 4GB.
NCLOB	A character large object containing multibyte characters. Both fixed-width and variable-width character sets are supported using the National Language character set of the database. The maximum size is 4GB.
BLOB	A binary large object with a maximum size of 4GB.
BFILE	Contains a locator to a large binary file stored outside the database such as an MP3 file or an image. The maximum size is of the BFILE object on disk is 4GB.

Notice in Table 1-2 that for character data, you have both NCHAR and CHAR columns, as well as NVARCHAR2 and VARCHAR2. This is because Oracle introduced support for National Language character sets in Oracle 8. In essence, this enables you to store data in CHAR and VARCHAR2 columns with one set of characters (Western European, for example) and data in NCHAR and NVARCHAR2 columns with a different set of characters (Japanese, for example). This allows the same database to house different character data instead of requiring a second database to hold the same information. In real work environments, the character set of a database and the National Language character set of a database typically are either the same or very similar.

Exam Tip This exam does not test your knowledge of how to use National Language characters or how to create a database to support different National Language character sets. It is sufficient to be aware of the two different sets of datatypes and which character set corresponds to which.

Constraints

Constraints are database objects that are used to ensure that data in the database makes sense. They are used to enforce business rules such as "every student must be uniquely identified" or "each student must have a first and last name, but not all students are required to have an email address." Constraints can also be used to prevent deletion of data in one table that may be depended upon by data in another. For example, if you wanted to ensure that an instructor is not deleted from

the Instructors table when he or she is currently teaching or has taught a course, you can create a foreign key on the Classes table that points to the Instructors table's primary key. After these constraints are defined, an instructor cannot be deleted if one row in the Classes table has that instructor's primary key value in a row in the Classes table.

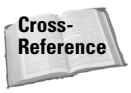
The types of constraints supported by Oracle 8*i* are listed in Table 1-3.

Table 1-3
Constraints Supported by Oracle 8*i*

Constraint	*Description*
NOT NULL	This constraint states that a column must have a value at all times. Oracle supports NULL by default on all columns in a table, which means that a value for the column does not need to be entered. If a value is required, a NOT NULL constraint can be defined on the column. For example, to ensure that all students have a first and last name entered in the Students table, you can specify the NOT NULL constraint for the FirstName and LastName columns.
UNIQUE	A UNIQUE constraint ensures that the value for a column or combination of columns in a table is unique or NULL for the entire table. This can be used to prevent the duplication of data. UNIQUE constraints create or use an existing index to enforce this uniqueness. A table may have multiple UNIQUE constraints.
PRIMARY KEY	A PRIMARY KEY constraint is the combination of a NOT NULL and UNIQUE constraint. This means that any column or columns defined for the PRIMARY KEY constraint ensure that data in the table is UNIQUE and NOT NULL. Like a UNIQUE constraint, a PRIMARY KEY constraint also either creates or uses an existing index to enforce the constraint. A table may have only one PRIMARY KEY constraint.
FOREIGN KEY	A FOREIGN KEY constraint states that data in the column or columns of a table reference a PRIMARY KEY or UNIQUE constraint of another table to ensure that the value entered is valid. For example, when specifying that a class is taught by a specific instructor, a FOREIGN KEY on the InstructorID column of the Classes table can reference the InstructorID of the Instructors table to ensure that a nonexistent instructor is not assigned to a class.
CHECK	CHECK constraints are used to enforce simple business rules, such as an enrollment date for a class is not after the end date for the class. CHECK constraints can only reference data in the same row of the table and cannot perform any kind of lookups in other tables to verify the condition. For the enforcement of more complex business rules, triggers should be used.

Constraint	*Description*
DEFAULT	A DEFAULT is not actually considered a constraint, although it is often grouped with them. A DEFAULT ensures that when data is inserted into a table and no value is specified for a column on which a DEFAULT has been defined, the value specified by the DEFAULT and not NULL is automatically assigned to the column. DEFAULTs are a way to ensure that a NOT NULL constraint is not violated when data is not entered into a column through a user action. It specifies what the value should be, by default.

Constraints are a way to ensure that data in your database is entered to enforce database consistency in order to satisfy business requirements. Should the business rules change, constraints can also be modified without any changes to the client application accessing the database.



The syntax and options for creating tables with constraints and adding constraints to existing tables are outlined in detail in Chapter 7.

Sequences

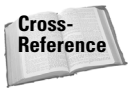
One of the challenges faced by any database developer is ensuring that unique values are always entered for columns with PRIMARY KEY or UNIQUE constraints. Oracle supports the use of sequences to help in this task.

Sequences are database objects that generate incremental numeric values that are always unique for the named sequence. When you need to insert a new value for a row in a table with a primary key, instead of figuring out what the last number was and adding one to it, you can define a sequence and assign the next value in the sequence. An Oracle database can support many sequences simultaneously so that you can define one for each PRIMARY KEY or UNIQUE column where you need to generate new values.

The syntax and options for creating sequences and how to use them to ensure unique values are inserted into the rows of a table are outlined in detail in Chapter 7.

Views

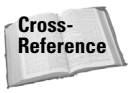
A view is a database object that enables you to present data from one or more tables in a single rowset (or recordset). It is created by specifying a name for the view and assigning that name to the SELECT statement that defines the view. When a user retrieves data from the view, the view appears as if it were a table. In fact, a user cannot tell the difference from the output when using an SQL SELECT statement to retrieve data from a view or a table — the output appears the same with column headings and rows returned as expected.



The syntax of the SELECT statement is discussed in detail in Chapters 2 through 4.

Views are useful to make complex queries easier to write by only having to do it once. By creating a view whose definition is the complex SQL SELECT statement for the query, users are now able to SELECT from the view and do not have to repeat the complex SQL syntax each time they want the same result.

Views are also useful in restricting access to only certain rows or columns in a database. For example, if you want to create a phone list for your organization with everyone's name, email address, phone number, and office location, you can create a table to hold that information or, preferably, create a view that extracts the necessary columns from the Employees table. In this way, the data is only stored once in the Employees table, and those columns in the Employees table that users should not be allowed to see (salary, bonus, last review results, and so on) are not available through the view.



The syntax and options for creating views are outlined in detail in Chapter 7.

Indexes

Database users always want to be able to retrieve data in the fastest possible way. In large databases, scanning the entire table to locate a particular value for a row can take a long time. Indexes can be created to make data retrieval quicker.

Indexes store, by default, the value of the column or columns of a table being indexed (also known as the *key*) in the index, as well as a pointer to the physical location of the row or rows that hold the value (the *rowid*). By issuing a query against the table where the index is created, Oracle may decide to scan the index, which is always organized in either ascending or descending order according to the key, and when it finds the entries with the required key value, use the corresponding key's rowid to return only those rows with the appropriate value.

Indexes can also be used to enforce uniqueness in a table. It is possible to create a unique index on a table (and one is created for you if it does not already exist) when you specify a UNIQUE or PRIMARY KEY constraint on a table. Oracle uses the index to ensure that no duplicate value exists.

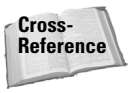
You should note that indexes can also increase the time it takes to perform inserts or updates of data on a table on which they are defined. This is because as rows are inserted or updated in a table, Oracle may have to insert the data as well as create or update any index keys for that row. Having too many indexes on a table can be detrimental to the speed of data modifications; not having enough indexes can be detrimental to data retrieval speed.

The syntax and options for creating and managing indexes are outlined in detail in Chapter 7.

Synonyms

If you took an English course in school (and most of us had to), you are already familiar with what a synonym is: a word that means the same as another word, such as *car* and *automobile*. Synonyms in Oracle do the exact same thing: They enable you to refer to an object in the database by another name.

Synonyms are created for other objects, such as tables and views. They can be used as a form of shorthand for tables with long names (for example, "Emps" for the Employees table or "Teachers" for Instructors). They can also be created to refer to objects that are owned by another user without having to specify the fully qualified name of the object. They are, in short, a form of shorthand.



The syntax and options for creating and managing synonyms are outlined in detail in Chapter 7.

User-defined datatypes

Starting with Oracle 8, you are able to create your own user-defined datatypes. These datatypes must be based on one of the scalar datatypes outlined in Table 1-2, or on another datatype already created. The reasons for doing so primarily deal with ensuring consistency across the database.

For example, let's say you wanted to ensure that address information for students, instructors, and employees included the following information:

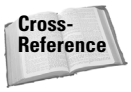
- ✦ Street address
- ✦ City
- ✦ State or province
- ✦ Postal or zip code
- ✦ Country

To guarantee that all tables have the same information, you can manually review that the column structure of each table has the necessary columns of the appropriate datatype and size, or you can create a user-defined datatype AddressType with these attributes. If you create a user-defined datatype, you can then create an Address column in each of the required tables whose datatype is the AddressType. In this way, all tables that have AddressType as the datatype of the Address column will have a uniform structure.

The "Introduction to Oracle: SQL & PL/SQL" exam does not test your knowledge of how to create and use user-defined datatypes. You only need to be aware that they exist and why they are useful.

Program units

Oracle 8*i* also enables you to create database objects that are actually programs. The objects that you can create are stored procedures, functions, triggers, and packages. Each of these objects can be written in either PL/SQL or Java and is used to manipulate data, enforce business rules, and perform database actions.



The syntax and options for creating stored procedures, functions, triggers, and packages are outlined in detail in Chapter 13.

The "Introduction to Oracle: SQL & PL/SQL" exam does not test your knowledge of stored programs or how to create or maintain them. However, this information is useful in creating a working database that can be easily maintained.

Stored procedures

Stored procedures are database objects that contain PL/SQL or Java code. They are created in the database and may have parameters passed to them (IN parameters), pass values back to the calling program or statement (OUT parameters), or have parameters do both (IN OUT parameters). They are used to perform further manipulation of data prior to it being committed to the database or to simplify complex actions that may involve several tables. Stored procedures can perform any action that is available through PL/SQL or Java and can be as simple or as complex as required.

Stored procedures can be invoked from another stored procedure or from the command line using SQL*Plus. They can also be invoked from many client applications such as Oracle Forms or Procedure Builder.

User-defined functions

User-defined functions in Oracle are also named blocks of PL/SQL or Java code that perform a specific action. They too can have IN parameters but do not support OUT or IN OUT parameters. Functions, however, must return a value, and the datatype of this value is specified in the function definition.

User-defined functions in Oracle can be used anywhere Oracle's own functions are allowed (that is, in any expression), but if the user-defined function operates on data, it can operate on only one row at a time. This means that you cannot create user-defined group functions (that is, multi-row or aggregate functions).

Chapter 3, "Using Single- and Multi-Row Functions," outlines the built-in functions available in Oracle and how to use them. User-defined functions can be used anywhere single-row functions are allowed in Oracle.

Triggers

Triggers are a special kind of stored procedure that cannot be invoked interactively (that is, you cannot call them from another stored procedure, or invoke them from the SQL*Plus command line or from a client application). Triggers are tied to a specific table or view, and an action being performed on the data in that table or view. The actions that can invoke a trigger include INSERT, UPDATE, and DELETE.

When a trigger is defined on a table, and the action for which it is defined takes place, the code in the trigger (which can be PL/SQL or Java) is run. If the trigger code runs smoothly and the changes satisfy the trigger conditions, no message is displayed, and the action is allowed to proceed. If the trigger code has a condition that is not satisfied, an error is raised in the trigger and returned to the calling program, in which case, the statement is rolled back, and the calling program or application must take corrective action.

Packages

In many situations, it is quite likely that a series of functions and stored procedures is run frequently to perform a particular series of actions. For example, when enrolling a student in a course, you may need to insert data into a number of tables and perform various verification checks to ensure that the student is eligible to take the course, that the course has room, that the course is being offered, that an instructor has been assigned, and so on. Each of these actions can be performed through a stored procedure, and some user-defined functions may be used in performing these actions. You can create each stored procedure and function by itself, or you can group them all into a package.

A package makes it easy to reference and load into memory (for speed) all the components that make up the package (such as stored procedures, functions, user-defined datatypes) by using a single name. Furthermore, because a package consists of a specification (which is the list of procedures and functions that is visible to anyone looking for information on the package) and a package body (which is the actual code to be run), it is possible to "hide" some of the code actually run by creating other procedures or functions that exist only in the package.

For more information on creating packages, and public and private procedures and functions within a package, refer to Chapter 13.

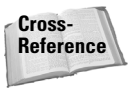
The Oracle Data Dictionary

As you may well imagine, a database may contain hundreds and even thousands of objects. Keeping track of all this information is the job of the Oracle data dictionary. A data dictionary in any database contains metadata information. Metadata is "data about data" or a set of tables and other database objects that store information about your own tables and database objects.

The data dictionary in Oracle is a set of tables, called *base tables*, that contain the most basic information about user-created database objects. These base tables are owned by an Oracle user called SYS, which is created when the database itself is created. The base tables are never accessed directly because their names are cryptic by design to discourage users from querying and modifying them. To make it easier to access the data dictionary and get information on objects in the database, a series of views are created during the database creation process. These views are commonly referred to as *data dictionary views*.

Oracle has three sets of data dictionary views:

- **USER_ views:** These views enable users to get information on objects that are in their schema (that is, objects that they have created and own).
- **ALL_ views:** These views enable users to get information on objects that they own or that they have been given access to. ALL_ views contains a superset of the information presented in USER_ views and enables users to find out what other objects they are allowed to reference or manipulate in the database.
- **DBA_ views:** The DBA_ views are designed to be used by the database administrator (DBA) and provide a full set of information for objects in the database (that is, any object created by any user). Normal users do not have access to these views because special privileges are needed to SELECT from them.



A list of USER_ views available in Oracle 8*i* databases and their structure is included in Appendix F, "Data Dictionary Views."

The Oracle data dictionary contains many USER_ views that enable an Oracle user to have a complete set of information on the objects that they own. Some of the more common views are listed in Table 1-4.

Table 1-4
Commonly Referenced USER_ Views in Oracle 8*i*

View	*Description*
USER_CATALOG	Lists all tables, sequences, and views in the user's schema.
USER_CONSTRAINTS	Displays information about all constraints created by the user in any schema.
USER_CONS_COLUMNS	Displays information about the columns on which the user has created constraints.
USER_ERRORS	Lists errors for PL/SQL blocks executed by the user. This includes anonymous PL/SQL blocks as well as stored programs like procedures, functions, and packages.
USER_INDEXES	Lists information on all indexes created by the user on objects in the database.

View	*Description*
USER_IND_COLUMNS	Lists the columns and the position (in a composite index) of objects in the database on which the user created indexes.
USER_IND_EXPRESSIONS	For function-based indexes, lists the column, its position in the index, and expression applied for the index created by the user.
USER_OBJECTS	Displays information about all objects in the user's schema.
USER_SEGMENTS	Displays storage properties of segments created by the user. Segments may include tables, indexes, materialized views, partitions, and LOBs.
USER_SEQUENCES	Displays information on sequences created in the user's schema.
USER_SYNONYMS	Lists synonyms in the user's schema and the objects to which they refer.
USER_TABLES	Displays information on tables created in the user's schema and, if an ANALYZE has been performed, statistics for them.
USER_TAB_PRIVS	Lists all permissions that were granted on tables to others by the user, granted on tables by others to the user, or granted on tables in the user's schema. This includes tables owned by the user, those owned by the others to which the user has been given permissions, or those tables to which the user has granted others permissions.
USER_TAB_PRIVS_MADE	Lists permissions granted to others by the user for tables in the user's schema.
USER_TAB_PRIVS_RECD	Lists permissions granted to the user by others for tables not in the user's schema.
USER_USERS	Displays information on the current user such as default and temporary tablespace settings and password expiration date.
USER_VIEWS	Displays information on all views created in the user's schema.

Key Point Summary

In understanding Oracle's place in the development of database management systems and its features and functionality, please keep these points in mind:

- ✦ Oracle is a database management system that is based upon the relational model developed by E.F. Codd in 1970. This makes Oracle an RDBMS.
- ✦ Oracle 8 and later versions also have features supporting the use of objects in the database. This makes Oracle an ORDBMS.
- ✦ The data dictionary stores information about all objects created in a database.

- ✦ USER_, ALL_, and DBA_ views enable you to query the data dictionary in order to retrieve information about the objects that you own or to which you have access.
- ✦ All database objects in Oracle are owned by a user. The collection of all database objects that a user owns is called a *schema*.
- ✦ Tables are database objects that store information about things or entities. Tables are composed of columns and rows.
- ✦ Columns must have at least two attributes — a name and a datatype.
- ✦ Oracle supports columns of character (CHAR, NCHAR), variable character (VARCHAR2, NVARCHAR2), numeric (NUMBER), date (DATE), large object (BLOB, CLOB, NCLOB, BFILE, LONG, and LONG RAW), binary (RAW), and physical location (ROWID, UROWID) datatypes.
- ✦ Indexes are created to speed data retrieval and to enforce uniqueness of data. They require additional storage and typically consist of a key value and a rowid for each row in the table on whose columns they are created.
- ✦ Constraints are used to enforce business rules and guarantee data integrity. Constraints supported by Oracle 8*i* include NOT NULL, PRIMARY KEY, FOREIGN KEY, UNIQUE, DEFAULT, and CHECK.
- ✦ A view is a SQL SELECT statement that has been given a name and been created in the database. Views are used to hide complexity from the user and also provide a security mechanism by allowing access to some data in a table without requiring that access be granted to the whole table.
- ✦ Sequences are used to generate primary key values and ensure that the values generated are not repeated.
- ✦ Synonyms enable you to reference objects in your schema, or another user's schema, using a shorthand name.
- ✦ User-defined datatypes enable you to ensure that data in tables follows a consistent pattern and structure. The use of user-defined datatypes requires that the SQL syntax used to insert, update, and delete rows is somewhat modified.
- ✦ Subprograms consist of stored procedures, user-defined functions, triggers, and packages.
- ✦ The main difference between a function and a stored procedure is that a function must return a value and a stored procedure does not have to.
- ✦ A trigger cannot be called interactively and is invoked when only the action for which the trigger is created takes place on the table on which the trigger is defined.
- ✦ Packages enable you to group stored procedures and functions commonly used together under a single name. All components of a package are loaded into memory when any one part of it is accessed, which helps improve performance.

✦ ✦ ✦

STUDY GUIDE

This section will enhance your understanding of the material presented in this chapter. Answer the questions and then work through the lab exercise in order to feel more comfortable with the material.

Assessment Questions

1. Which of the following are valid database datatypes in Oracle 8*i*? (Choose all correct answers.)

A. BOOLEAN

B. NUMBER

C. CHAR

D. NBLOB

E BFILE

F. RECORD

2. For which of the following types of information stored in a database is the use of a sequence appropriate? (Choose two correct answers.)

A. Invoice line item

B. Invoice number

C. Employee name

D. Atomic element

E. Customer identifier

3. Under what conditions may Oracle automatically create an index? (Choose two correct answers.)

A. A primary key is added to a table.

B. A foreign key is added to a table.

C. A default is added to a column.

D. A check constraint is added to a table.

E. A unique constraint is added to a table

4. If not specified explicitly, which of the following is automatically assumed to be true for a column in a table? (Choose the best answer.)

A. Datatype is varchar2

B. Default is ''

C. NOT NULL

D. NULL

E. The column is named "Col" plus a unique number.

5. How many columns of datatype LONG RAW are allowed in a table? (Choose the best answer.)

A. 1

B. 5

C. 10

D. 1,000

E. 0

6. What two components do typical Oracle indexes store for each row of a table being indexed? (Choose two correct answers.)

A. File number

B. ROWID

C. Key value

D. Block number

E. Entire row

7. How much data may be stored in a an external file specified in a column of datatype BFILE? (Choose the best answer.)

A. 2GB

B. 64KB

C. 4MB

D. 4GB

E. 2TB

8. What is a key benefit to making use of user-defined datatypes in Oracle? (Choose the best answer.)

A. Ability to rename Oracle built-in datatypes

B. Consistency of similar data structures across multiple tables

C. Polymorphism

D. Inheritance

E. Easier maintenance of databases

9. Which data dictionary views can you query to get information on the tables in your database? (Choose three correct answers.)

   A. USER_SEGMENTS

   B. USER_USERS

   C. USER_OBJECTS

   D. USER_TABLES

   E. USER_CONSTRAINTS

10. In which of the following situations would it be beneficial to use a view? (Choose all correct answers.)

    A. You generate a sales report on a weekly basis that joins several tables and performs calculations to return a single set of data.

    B. You want to enable users to see only the retail price of products that you sell.

    C. You need to move data from one table to another.

    D. You want to ensure that primary key values are not being repeated.

    E. You want to enable users to create their own tables.

Scenario

Your small company is considering implementing an enterprise resource planning application that uses Oracle so that you are more easily able to manage your business as the 200 percent annual growth rate you predict for the next three years comes into force. Currently you have the following information tracking taking place:

1. A billing system that stores customer invoice and payment information.
2. A hard copy of all invoices and purchase orders in a filing cabinet.
3. A spreadsheet-based application that tracks receipt of goods from suppliers and reports on those items received so that they can be matched to purchase orders.
4. A Microsoft SQL Server-based application that enables sales and customer service representatives to track communications and interactions with your clients.

You are looking to consolidate all of this into a single system based on Oracle.

A. How many databases do you currently have in place?

B. How many of those are relational database management systems (RDBMSes)?

C. What is the most important factor in moving all of your data to a single Oracle-based system?

D. What language will your client applications use to extract and update data in the Oracle database?

Lab Exercise

This chapter does not have any preset lab exercises. However, because the rest of the labs require that you have Oracle installed and the database objects outlined in Appendix E created, this is a good time to install Oracle and run the scripts outlined in Appendix E.

Lab 1.1 Setting Up Oracle and Creating Tables

1. The first step to installing Oracle is to acquire the software. Oracle 8.1.6 Enterprise Edition or later is required for the successful completion of the labs. Although there is no reason why you cannot use Oracle 8i Standard Edition, or Personal Oracle 8i, Oracle's "Introduction to Oracle: SQL & PL/SQL" exam assumes that you have been working with Enterprise Edition.

   The operating system may be Windows NT, Windows 2000, or any Unix variant including Linux. If you do not own the software, you may be able to download it from Oracle's Web site by joining the Oracle Technology Network (OTN). More information is available on Oracle's Web site at `http://otn.oracle.com`.

2. Once you have acquired the software, read the documentation that came with it. If you did not get paper documentation, view any online documentation for information on the requirements to successfully install it on your computer.

3. Make sure that your computer meets the minimum system requirements outlined in Appendix A of this book, "What's on the CD-ROM." As a minimum, your computer should be a Pentium 266 or better with 128MB of RAM, a CD-ROM drive, an SVGA video card, and a 10GB hard disk with a minimum of 2GB free disk space.

4. Install the Oracle software as per the instructions provided by Oracle. When running the Oracle Universal Installer to install the software and asked to create a starter database, have the Universal Installer create it with the default settings on the hard disk with the most free space.

5. You may need to restart your computer before proceeding. If you are not sure, restart your computer anyway.

6. Create the tables and other structures required for the rest of this book by following the directions provided in Appendix E. If you want to create the Oracle components on a drive other than C, or you are not running Windows NT or Windows 2000, you must change the scripts to point to the new location or to properly create the files (for example, for Linux, change the backslash (\) to a forward slash (/) for directory paths.)

Answers to Chapter Questions

Chapter Pre-Test

1. A relational database management system (RDBMS) is based upon the relational model developed by E.F. Codd in 1970. This model states that all data is a collection of relations (tables) with only logic links between them. This means that to find data in one table related to data in another table, you do not need to know the physical location of data in the each table. Only the key value that is common to both, regardless of the physical location of the data, is needed to find the data in the other table. This is different from flat file, network, or hierarchical systems where the physical location of the related data must be known.
2. Every column must have a name that is unique within the table, and a datatype, and size. Without these, a column definition is not complete.
3. You specify a NOT NULL constraint for a column when you want to ensure that data is always entered into the column whenever a row is inserted or updated. NOT NULL specifies that the value of the column must always be known.
4. Oracle is an Object Relational Database Management System because it supports object enhancement to the relational model, including the ability to create user-defined datatypes and support for large objects. Oracle supports large objects in columns of BLOB, NCLOB, CLOB, and BFILE datatypes.
5. A PRIMARY KEY constraint specifies that data in the column or columns is unique within the table and cannot be null. A UNIQUE constraint specifies that data in the column or columns is unique within the table or NULL. Both a PRIMARY KEY and UNIQUE constraint creates indexes to enforce the uniqueness, if they do not already exist. You are allowed to have only one PRIMARY KEY constraint on a table but may have any number of UNIQUE constraints.
6. You may define as many FOREIGN KEY constraints on a table as makes sense to enforce your business rules. There is no maximum, but you are limited to 1,000 columns per table and 255 indexes per table. These may become limitations before the number of FOREIGN KEYs on the table.

7. Sequences enable you to ensure that PRIMARY KEY or UNIQUE constraints always have a numerical incremental value assigned to the columns in the appropriate table, regardless of the number of users in a database. This prevents you from having to manually keep track of the next incremental value and enables you to have Oracle do it for you.

8. In Oracle you can create four types of stored subprograms. They are: stored procedures, triggers, user-defined functions, and packages. Packages are considered a fourth type of subprogram but really are made up of stored procedures or user-define functions.

   You can also create user-defined datatypes, which may have methods defined for them, and these may also be considered subprograms, although they are really object-oriented features.

9. Views enable you to take complex SQL statements and give them a logical name in the database. In this way, instead of typing the whole statement again, when you select from the view, the SELECT statement defining the view is run.

   Views also enable you to hide complexity from the user by having a multi-table join assigned a logical name. Furthermore, you can also enable users to have access to parts of the database that they would not have permission to by creating a view that selects only those columns that a user should be allowed to see, such as an employee's name, email address, and phone number for a phone list based on the Employees table.

10. Indexes have a positive effect on database queries — they can speed them up and return the data to the user much quicker than by scanning the table as a whole to find the relevant data. Indexes have a negative effect on data updates because they require that both the data and index entries be updated whenever a new row is inserted into a table, a key value is updated, or a row is deleted from the table. The ideal for indexes is to have enough of them to speed up queries but not too many to slow down data changes too much.

11. A constraint is used to enforce simple business rules, such as each student must have a first and last name, and to ensure relational integrity. This way, when using constraints, it is not possible for a user to delete a row from a parent table if corresponding child table rows exist, or to enter a duplicate student ID if a PRIMARY KEY constraint exists on a table.

    Triggers are a stored subprogram written in PL/SQL or Java that enable you to include conditional logic and perform checking of more sophisticated business rules, such as a student may not be enrolled in two courses that run at the same time, or an instructor cannot teach two courses without at least a 30-minute break between them.

Assessment Questions

1. **B, C, E** — Valid datatypes for columns in Oracle 8*i* include CHAR, NCHAR, VARCHAR2, NVARCHAR2, NUMBER, DATE, ROWID, UROWID, BLOB, BFILE, CLOB, NCLOB, LONG, LONG RAW, and RAW. RECORD and BOOLEAN are valid datatypes for variables in PL/SQL but cannot be used as a datatype for a column in the database. There is no NBLOB datatype.

2. **B, E** — Because invoice number and customer ID are unique ways to identify a specific invoice or customer, they are good candidates for the use of a sequence. Because most databases are accessed by several users at the same time, the use of sequences enables the developer to ensure that each new invoice or customer added is assigned an incremental numeric value. That way, they eliminate the need to manually keep track of this information in the application.

   Although at first glance, an invoice line item might also seem an appropriate candidate for the use of a sequence, because it is most likely that the line item number would be reset to 1 for each invoice created, it means that a sequence would have to be created for each invoice — an almost impossible task to manage.

   Employee name requires a character datatype, and sequences must be numerical, so one cannot be used here. Atomic elements already have a unique identifier — their symbol and atomic weight — so a sequence is not appropriate here.

3. **A, E** — Oracle automatically creates an index to correspond to the definition of a PRIMARY KEY or UNIQUE constraint, if an index does not already exist when the constraint is created. When any other type of constraint is added to a table, including a FOREIGN KEY constraint, no indexes are created.

4. **D** — When defining a column of a table, you must specify the column name, datatype, and size. Failing to do so causes Oracle to generate an error, and the table creation fails. However, if you do not specify NULL or NOT NULL, Oracle always assumes that a column should support NULLs. Therefore, it is always a good idea to explicitly specify whether or not a column should support NULLs so that you are able to determine it by looking at the table create statement. Furthermore, not all RDBMSes support NULLs by default, so if you want your table definition to be portable between systems, you should also explicitly specify whether or not a column supports NULLs.

5. **A** — Oracle supports the use of columns of LONG and LONG RAW datatypes in Oracle 8*i* for backward compatibility only. This means that the rules that are applied in previous versions of Oracle for these datatypes still apply. For this reason, you are allowed to create a table with only one column of LONG or LONG RAW datatype. Furthermore, the table in which this type of column is being defined cannot have any columns based upon objects (that is, LOBs, VARRAYs, nested tables, or user-defined datatypes).

6. **B, C** — Oracle's default index type of B*Tree always stores the value (that is, the key value) found in the table for the column or columns being indexed and the location of a row (that is, the rowid) that contains the value being indexed. This ensures that when a user performs a query where the key value found in the index is used, Oracle can scan the index and retrieve the rowids of all rows that have the value and then very quickly return the rest of the data for the rows by going directly to the location specified in the rowid. The rowid contains the file number, block number, and slot for the row, which is its physical location on disk.
7. **C** — A BFILE datatype can store up to 4GB of information. This is the same as any other datatype supporting large objects such as BLOB, CLOB, and NCLOB.
8. **B** — When creating a user-defined datatype and using it in your database, you are ensuring that whenever that user-defined datatype is used in the definition of a table, it will be the same in every case. This enables you to ensure consistency of similar data structures across multiple tables. If you answered **E**, you should get partial points as well because, in the long run, the database will be easier to maintain.
9. **A, C, D** — The USER_SEGMENTS view stores information about the physical storage of a table including the tablespace it is created on, and other storage parameters specified at creation time or modified since then. The USER_OBJECTS view stores information about all objects that a user owns, including tables, indexes, and user-defined datatypes. The USER_TABLES view stores information about the table including storage parameters and statistical information populated through the ANALYZE command. The USER_CONSTRAINTS views has information about constraints in the database, including the table on which the constraint is defined, so answering **E** gives you partial points because the main purpose of the view is not to provide information on the table but the constraint. The USER_USERS view has information about the currently logged in user and does not contain any information about tables.
10. **A, B** — Generating a sales report on a regular basis using a complex query is a good candidate for a view because you are able to write the SQL statement once to define the view and then SELECT from the view to get the results. Should the requirement change, you only need to redefine the view to match the new criteria. Any application using the view, assuming it returns the same number and types of columns, does not have to be changed.

    Projecting only the columns you want, such as enabling customers to see only the retail price for a product, is also a good candidate for a view, because you do not need to assign permissions to the underlying tables in order for users to see the information they need. This helps in securing your data.

    Moving data from one table to another cannot be done through a view. Enabling users to create their own tables requires you to assign them the permissions to do so. Ensuring that primary key values are unique for a table is a good candidate for the use of a sequence.

Scenario

A. Based upon the description presented, you currently have four database management systems in place. This includes the billing system holding customer invoice and payment information, the spreadsheet-based application tracking receipt of goods from suppliers, the Microsoft SQL Server-based customer relationship management system, and the filing cabinets with the hard copies of the invoices and purchase orders. It is important to remember that filing cabinets, and even shoeboxes, are database management systems because they allow for the quick retrieval and organized storage of data.

B. Only one of the systems that you currently have is a relational database management system (RDBMS). This is the customer relationship management application that is using Microsoft SQL Server. This will be replaced by your Oracle-based enterprise resource planning software.

C. The most important factor in moving your data to an Oracle-based system is the design of the tables and other database objects that are needed to hold the information. In designing a database, the business requirements must be mapped to a logical structure. In performing this task, you should not rush its completion, and you should ensure that all requirements are met.

D. RDBMSes, such as Oracle, use the Structured Query Language (SQL) to retrieve and change data. This is the language that you will use once you migrate to Oracle. You can also make use of PL/SQL to write subprograms and client-side application logic.

CHAPTER 2

Retrieving Data Using Basic SQL Statements

EXAM OBJECTIVES

- Writing basic SQL statements
  - List the capabilities of SQL SELECT statements
  - Execute a basic SELECT statement
  - Differentiate between SQL statements and SQL*Plus statements
- Restricting and sorting data
  - Limit the rows retrieved by a query
  - Sort the rows retrieved by a query

CHAPTER PRE-TEST

1. What two syntax elements are required in *all* SQL queries?
2. How do you find only one occurrence of a repeating value in a table?
3. Which is evaluated first, the multiplication or addition operator?
4. What happens if you subtract two date values from each other?
5. If you have a column that stores order dates, how do you query the database for orders placed in a quarter?
6. How do you find rows that are not in a range?
7. How can you override the order of precedence for any SQL operation?
8. What happens if you have a NULL in an arithmetic expression?
9. What operator do you use to find a NULL field in a WHERE clause?
10. What is the default column name for any column in a SELECT list that is derived from an expression?

✦ Answers to these questions can be found at the end of the chapter. ✦

This chapter examines one the most important aspects of the Structured Query Language (SQL) — the SELECT statement, or SQL query. Having a relational database isn't useful if you cannot effectively access the data it contains. The basic SQL query is how you access your data. If you do not understand the SELECT statement and all of its options, you cannot retrieve your data. The basic SELECT statement has a number of key clauses and operators, and in order to take full advantage of Oracle's querying capabilities, you must completely understand these language elements. In this chapter, you will learn how to use the various elements of the SELECT statement to access data from one or more tables and to limit your queries so that they return only the exact data that you require. You will also examine the various functions in Oracle that enable you to manipulate and format data. Finally, you will look at ways in which you can use SQL queries to derive summary information from your data using grouping functions.

As a Relational Database Management System (RDBMS), the Oracle engine controls all access to the database. Any activity that deals with data must be done through SQL. In order to work effectively with data in the database, you must have a full understanding of how Oracle's implementation of SQL works. The primary focus of this chapter is the basic query language of SQL. Queries in SQL all start with the SELECT statement. A number of key clauses and operators make up the basic SELECT statement, and for the exam, you will be expected to be familiar with all of them. Before we look at the basic SELECT statement in SQL, however, let's first examine the various elements that make up the SQL language.

A Quick SQL Overview

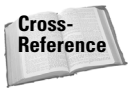
In Chapter 1, we introduce the concept of the Relational Database Management System (RDBMS). The RDBMS controls all access to the data contained in the database. When you want to change the storage structure of the database (for example, creating or modifying a table), adding or modify data, or just retrieving data from the database, you must do so using Oracle's implementation of SQL. Even when you are connecting to the database through a client application (such as an Oracle Forms application), you still send SQL statements to the RDBMS to perform actions on the data.

Oracle's implementation of SQL is based on the entry-level American National Standards Institute (ANSI) SQL standard. However, as is the case with most commercial database systems, Oracle has modified the ANSI standard (often referred to as "ANSI -92") to fit the needs of its internal systems. As a result, Oracle's SQL is unique, and any script written for an Oracle database may not run correctly on any other database system. Therefore, if you are going to be working with Oracle, you need to learn its rules.

SQL can be broken down into three basic sections or "languages": Data Definition Language (DDL), Data Control Language (DCL), and the Data Manipulation Language (DML).

Data Definition Language statements

DDL statements do not deal with data directly, rather they deal with the objects that hold and provide access to data. When you want to add, modify, or remove any object in the database schema (such as a table, a view, or an index), you must do so using a DDL statement. DDL includes, among others, all CREATE, ALTER, and DROP statements. Database creation begins with DDL statements; they build the entire framework of tables and constraints that will become the structure of the database. Before data can be added, you must have some place to add it. Your users, normally, will not issue DDL statements; your database administrators (DBAs) and database developers will perform them.



DDL statements are discussed in detail in Chapter 7, "Creating and Managing Oracle Database Objects."

Data Control Language statements

Oracle databases often contain sensitive information, and controlling access to data is essential. Database access is controlled by the Data Control Language. Once a login account is created for a user, that user can be given privileges on the database. Privileges are given using the GRANT statement and is taken away using the REVOKE statement. These two statements form the core of the DCL.

Two types of privileges can be granted and revoked: system and object . System privileges enable a user to perform action on the database. System privileges are required to perform such activities as creating a table or index, or backing up the database.

Object privileges are applied to particular objects in the database. The privileges available may change depending on the type of object. For example, it is possible to grant a user the SELECT, INSERT, or UPDATE privilege on a table or view, but one of the EXECUTE privileges can be granted on a stored procedure. DCL statements are, for the most part, the job of the DBA. It is his or her job to create user accounts for every user and to assign those accounts the correct permissions.

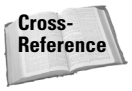
DCL statements are covered in Chapter 8, "Configuring Security in Oracle Databases."

Data Manipulation Language statements

DML statements are the level of the SQL language that deals directly with data. Any manipulation or retrieval of data requires a DML statement. The key elements of the DML are SELECT, INSERT, UPDATE, and DELETE. These commands enable you to retrieve data from a table, add new data to a table, modify existing data, and delete rows from a table.

An additional sublanguage is contained within the DML — the transactional control language (COMMIT, ROLLBACK, and SAVEPOINT). These language elements help control the execution of DML statements by grouping them together into transactions.

Transactions are one or more physical operations that are logically grouped into a single operation called a *logical unit of work*.



Transactions and DML statements are discussed in Chapter 5, "Adding, Updating, and Deleting Data."

DML statements are the most common type of interaction with the database. It is the level of SQL that your users will be working in almost exclusively. The DML includes the SELECT statement, and this is where we will begin this chapter.

General rules and conventions for all SQL statements

When you write any SQL statement, you should be aware of some general rules that apply to all of them:

- ✦ SQL commands and keywords cannot be abbreviated. If you attempt to truncate any SQL keyword, you will receive a syntax error.
- ✦ SQL statements are not case sensitive. In other words, you can use any mixture of uppercase and lowercase characters in your SQL statements and get the same results.

Oracle is case sensitive in its treatment of data. If you refer to a literal string value (that is, any value enclosed in single quotes), you must use the correct case. For example, Oracle treats the names "McLean", "Mclean", and "mclean" as three different values. This topic is discussed in detail later in this chapter.

Oracle also ignores indents, tabs, and carriage returns in SQL statements. This means that you can format the text of your code to make it more readable. Consider the following two scripts:

```
select instructorID, lastname, firstname, address1, address2,
city, state, country, postalcode from  instructors where
(firstname, lastname) in (select  firstname, lastname from
students);
```

or

```
SELECT InstructorID, LastName, FirstName, Address1
       Address2, City, State, PostalCode
FROM   instructors
WHERE (Firstname, LastName) IN (SELECT FirstName, LastName
                                FROM students);
```

SQL*PLus vs. SQL

SQL*Plus is a SQL editor tool that ships with Oracle. It allows you to submit SQL statements against the database. SQL*Plus is not a language, but it contains its own commands. SQL*Plus commands are used primarily to format output and to manipulate statements within the editor.

Oracle is not able to process SQL*Plus commands. When a script is executed that contains both SQL and SQL*Plus elements, SQL*Plus strips away the SQL*Plus commands and sends the SQL statements to Oracle. When the result set is returned, SQL*Plus applies its commands and returns the modified output to the user.

SQL*Plus commands can be abbreviated, but SQL commands cannot. For example, if you want to edit a script in SQL*Plus, you can use the command EDIT or simply type ED.

For more information on SQL*Plus see Chapter 6, "The SQL*Plus Environment."

To Oracle, both of these examples appear the same. It will parse and execute both statements exactly the same. However, the second query, from a user perspective, is clearly easier to read (and by extension, easier to edit and correct). The standard convention for SQL statements is to place keywords, functions, and operators in capital letters and to place object names (such as table and column names) in lowercase. It is also common to place the key elements on the left of the script and indent other elements to the right (as you can see in the second query). These are, of course, only guidelines. You will note, for example, that the second query capitalizes the initial character of all column names. The code was written this way to follow the manner in which these column names are expressed in the database. It also makes the point that rules are made to be broken! Ultimately, you must find a format you are comfortable with that provides legibility and clarity.

The Basic SELECT Statement

Objective

- ✦ List the capabilities of SQL SELECT statements
- ✦ Execute a basic SELECT statement

All SQL queries begin with the SELECT statement. This statement enables you to retrieve all data or only certain columns and rows in a table. It can also return data from more than one table. It enables you not only to retrieve data, but also to perform calculations on existing data and return the results of these calculations. The basic SELECT statement requires two elements: the SELECT list and a FROM clause. These clauses specify what columns to retrieve and from where. Here is the basic format:

```
SELECT [DISTINCT] {* | column,[expression], . . . }
FROM table;
```

The select list can contain either the asterisk or a list of column names or expressions. The select list dictates what is returned by the query. The asterisk is simply a shorthand, meaning all columns. For example, when you want to see all columns in the Courses table, you can use either of the following queries:

```
SELECT *
FROM Courses;
```

or

```
SELECT CourseNumber, CourseName, ReplacesCourse, Description,
RetailPrice
FROM Courses;
```

In both examples, the query returns all rows and all columns from the Courses table. One query, however, clearly takes a lot less typing! Note that all of the columns must be separated by commas; however, make sure that you do not put a comma after the last column. When the last column is followed by a comma, it indicates to Oracle that there is another column in the list to consider, and you will receive the following error:

```
ERROR at line 2:
ORA-00936: missing expression
```

The select list does not have to return all columns and does not have to follow the order of the columns in the table. For example, if you want only the retail price and name of a course, you can execute the following query:

```
SELECT RetailPrice, CourseName
FROM Courses;
```

Oracle returns all of the columns in the select list in the order they are referenced. You must always separate columns with a comma. You can also reference columns in an order that differs from the order in the data dictionary for the table.

Arithmetic operations

The select list is not limited to returning values that are actually stored in columns. It is also possible to return values that are derived from data that is stored in the tables. You may want to combine values, project changes in prices or salaries, and create "what if" projections. One way is to use arithmetic operations. The four arithmetic operators in SQL are listed in Table 2-1.

Table 2-1
Arithmetic Operators in Oracle

Operation	*Operator*
Multiply	*
Divide	/
Add	+
Subtract	-

You can use the operators in the select list with either literal values or columns to derive values for the select list. For example, suppose you want to generate a list that shows how much each instructor charges for a five-day course. You can use the following query:

```
SELECT InstructorID, PerDiemCost, PerDiemCost * 5
FROM   Instructors;
```

which returns the following result set:

```
INSTRUCTORID PERDIEMCOST PERDIEMCOST*5
------------ ----------- -------------
         300         500          2500
         310         450          2250
         100         600          3000
         110         500          2500
         200         750          3750
         210         400          2000
```

The instructors table doesn't contain a PerDiemCost*5 column. This column has been generated by Oracle and is returned in the result set as if it really existed. You do not need to use literal values with arithmetic operators. For example, when instructors must travel to teach a course, their actual per diem cost includes both their Cost and their Expenses. If you want to see the actual per diem for an instructor who is on the road, you can use the following query:

```
SELECT InstructorID, PerDiemCost + PerDiemExpenses
FROM Instructors;
```

The result set for this query looks like this:

```
INSTRUCTORID PERDIEMCOST+PERDIEMEXPENSES
------------ ---------------------------
         300                         700
         310                         650
         100                         800
         110                         700
         200                        1000
         210
```

Note that although you have referenced two columns, the arithmetic operation returns only one row of data.

Using arithmetic operators with date values

Oracle also enables you to perform arithmetic operations on data values. When you add or subtract a number from a date, you return a data value that is that many days before or after the date that you are dealing with. For example, suppose you want to calculate the date when a course ends. The ScheduleClasses table contains a StartDate column and a DaysDuration numeric column but does not store an end date. You can, however, generate this date using an arithmetic operator:

```
SELECT CourseNumber, StartDate, DaysDuration, StartDate +
DaysDuration
FROM ScheduledClasses;

COURSENUMBER STARTDATE DAYSDURATION STARTDATE + DAYSDURATION
------------  --------- ------------ -----------------------
         100  06-JAN-01            4 10-JAN-01
         200  13-JAN-01            5 18-JAN-01
         100  14-FEB-01            4 18-FEB-01
```

As you can see, the final column accurately represents the projected end date of each course based on the number of days. This works because Oracle stores date data in an internal numeric format that counts the number of days between a certain begin and end point. When you subtract a number from a date, Oracle simply adds or subtracts that number from its internal numeric value and recalculates the date.

You cannot use the multiplication or division operators with dates. Nor can you add two dates together. However, it is possible to subtract two date values. When you subtract two dates, Oracle provides a numeric value that is the difference (in days) between the two dates. If you take the STARTDATE and STARTDATE + DAYSDURATION columns from the previous example, you can reconstruct the DAYSDURATION values:

```
SELECT CourseNumber, (StartDate + DaysDuration) - StartDate
FROM ScheduledClasses

COURSENUMBER (STARTDATE+DAYSDURATION)-STARTDATE
------------ ------------------------------------
         100                                    4
         200                                    5
         100                                    4
```

If you alter the order of the two values in the equation so that the earlier date is referenced first, Oracle gives you a negative integer with the same numeric value (that is, –4, –5, and –4).

Understanding order of precedence with multiple arithmetic operators

At times you want to use more than one arithmetic operator in a select list. In order to work with multiple operators effectively, you must understand the order of precedence for these operators. The order of precedence is, simply, the order in which Oracle applies these operators. The order of precedence is governed by three rules:

- ✦ The multiplication and division operators have the same level of precedence and execute before addition and subtraction operators.
- ✦ Oracle processes all operators with the same level of precedence from left to right.
- ✦ You can override the rules of precedence with the use of parentheses.

You probably first came across these rules in fourth or fifth grade. The rules have not changed since elementary school; however, if you forget them in your SELECT statements, you may end up with inaccurate data.

For example, in the previous section you calculated the price that an instructor charges for a week and how much an instructor on the road charges. Suppose you want to know how much an instructor charges for one week on the road. If you used the following query to obtain this information, you would receive incorrect data:

```
SELECT InstructorID, PerDiemCost + PerDiemExpenses * 5
FROM Instructors;
```

With this query, Oracle executes the multiplication operation first and then the addition operation. In other words, you get each instructor's per diem cost added to five times the per diem expense. To correct the query, you have to use parentheses. Applying the third rule governing the order of precedence, parentheses override the natural order of precedence. To obtain the correct information, you use the following query:

```
SELECT InstructorID, (PerDiemCost + PerDiemExpenses) * 5
FROM Instructors;
```

In this query, the addition operation takes place before the multiplication and produces the values that you require.

You can use your arithmetic operations to derive complex calculations. However, as the complexity increases, you must be more aware of the rules of precedence. Consider the following query:

```
SELECT InstructorID,
       ((((PerDiemCost + PerDiemExpenses) * 5) -
       (PerDiemCost * 5)) / ((PerDiemCost + PerDiemExpenses) *
       5)) * 100
FROM Instructors
```

This query calculates the difference between the cost of an instructor with and without travel expenses for a week and then divides that amount by the cost of an instructor teaching on the road for a week. It then multiplies that amount by 100, providing the percentage increase of cost when the instructor is teaching a course that requires travel. As you can see, it is possible to nest expressions in parentheses. SQL processes the innermost expressions in parentheses first and then passes those values out to evaluate the outermost expressions. If you are missing any opening or closing parentheses, Oracle returns an error. For example, if you omit the final closing parenthesis on the preceding query, you receive the following error:

```
ERROR at line 2:
ORA-00907: missing right parenthesis
```

When writing queries like the preceding, it is best to start with the innermost expressions and work out. This makes it easier to keep your parentheses in the right order and to make sure you do not forget any closing parentheses.

Concatenating columns

In SQL it is also possible to use expressions on character data. You may want to display data stored in multiple columns as a single value in the result set. For example, first and last names are often stored in separate columns, but you may want a single "name" value that contains both the first and last name. This is done using the concatenation operator. In Oracle's implementation of SQL, the concatenation operator is represented by the double pipe (||) character. The pipe character is the vertical line above the backslash key (Shift+\). With this operator, it is possible to join together two or more columns into a single string expression. For example, when you execute the following query:

```
SELECT FirstName || LastName
FROM Instructors;
```

you receive the following output:

```
FIRSTNAME||LASTNAME
------------------------------------------------------------
MichaelHarrison
SusanKeele
DavidUngar
KyleJamieson
LisaCross
GeoffWilliams

6 rows selected.
```

Notice that only one column is returned and that the two columns are joined together without any intervening spaces. If you want a space between the two values, you must include a string-literal in the select list. A string-literal is simply a text string that is repeated for reach row in the result set. You include a string-literal by

including the text in the select list enclosed in single quotes. To place a space between the first and last name in the previous example, you concatenate a blank space between the two columns:

```
SELECT FirstName || ' ' || LastName
FROM Instructors;
```

The result set is as follows:

```
FIRSTNAME|| ' ' || LASTNAME
------------------------------------------------------------
Michael Harrison
Susan Keele
David Ungar
Kyle Jamieson
Lisa Cross
Geoff Williams

6 rows selected.
```

You can place any string value in the select list, and it is repeated in every line. For example, if you want to generate a list of products each instructor teaches, you can use the following statement:

```
SELECT FirstName || ' ' || LastName || ' teaches ' ||
       InstructorType
FROM Instructors;
```

The result set for this query looks like this:

```
FIRSTNAME||''||LASTNAME||'TEACHES'||INSTRUCTORTYPE
------------------------------------------------------------
Michael Harrison teaches ORACLE
Susan Keele teaches UNIX
David Ungar teaches ORACLE
Kyle Jamieson teaches ORACLE
Lisa Cross teaches UNIX
Geoff Williams teaches UNIX

6 rows selected.
```

Because a space is embedded on either side of the text string “teaches”, the spaces are included in the output. You can combine any number of columns and text strings in this manner. The output of the concatenation is always a string value. If Oracle encounters nonstring data, it implicitly converts this data into a character datatype. Consider this query:

```
SELECT 'Instructor ' || InstructorID || ' charges $'||
       PerDiemCost
 FROM Instructors;
```

In this case, the PerDiemCost column has a number datatype. However, Oracle converts the values in this column to a character datatype before including them in the result set. The result set is as follows:

```
   'INSTRUCTOR'||INSTRUCTORID||'CHARGES$'||PERDIEMCOST
-------------------------------------------------------------
Instructor 300 charges $500
Instructor 310 charges $450
Instructor 100 charges $600
Instructor 110 charges $500
Instructor 200 charges $750
Instructor 210 charges $400

6 rows selected.
```

This works fine as long as Oracle is able to make the conversion. However, consider this query:

```
SELECT 'Instructor ' || InstructorID || ' charges $'||
       PerDiemCost + PerDiemExpenses
 FROM Instructors;
```

When you execute this query, you receive this error:

```
SELECT 'Instructor ' || InstructorID || ' charges $'||
                                                    *
ERROR at line 1:
ORA-01722: invalid number
```

The problem with this query is that it has been asked to execute both an arithmetic operation and a string operation at the same time. To correct this, you must explicitly separate the two operations. In this case, you can solve the problem by placing parentheses around the arithmetic expression.

```
SELECT 'Instructor ' || InstructorID || ' charges $'||
       (PerDiemCost + PerDiemExpenses)
 FROM Instructors;
```

This forces Oracle to complete this operation first and then convert the resulting value to a string. Sometimes you are forced to explicitly convert values using a conversion function.

Conversion and other functions are covered in Chapter 3, "Using Single- and Multi-Row Functions."

Adding column aliases

In the previous examples, you may have noticed that when you execute a SELECT statement, the column headings in the result set are the same as in the select list.

This is fine when you are selecting column data, but it becomes more difficult when you use arithmetic or concatenation operations. Consider one of the previous examples:

```
SELECT InstructorID,
       ((((PerDiemCost + PerDiemExpenses) * 5) -
       (PerDiemCost * 5)) / ((PerDiemCost + PerDiemExpenses) *
       5)) * 100
FROM Instructors;
```

The output for this query looks something like this:

```
INSTRUCTORID
------------
((((PERDIEMCOST+PERDIEMEXPENSES)*5)-
(PERDIEMCOST*5))/((PERDIEMCOST+PERDIEMEXPENSES)*5))*100
-------------------------------------------------------------------------------
-----------
         300

28.571429

         310

30.769231

         100

25

         110

28.571429

         200

25

         210


6 rows selected.
```

Clearly this does not make for very readable output. SQL, however, provides a better way of handling column names through the use of aliases. An *alias* is simply a column-heading name that replaces the value from the select list.

There are two ways to add an alias to SQL SELECT statement. The first way is simply to place the alias after the column name in the select list. The alias and column name must be separated by a space. Make sure not to put a comma between the alias and column names. If you do, Oracle will interpret the alias as another column name. The second way to add an alias is to include the AS operator. Although the AS keyword is not necessary, it makes the code easier to read and should be included. When you rewrite the previous example with an alias, it looks like this:

```
 SELECT InstructorID,
      ((((PerDiemCost + PerDiemExpenses) * 5) -
      (PerDiemCost * 5)) / ((PerDiemCost + PerDiemExpenses) *
      5)) * 100 Price_difference;
FROM Instructors
```

When you use the AS operator, it looks like this:

```
SELECT InstructorID,
     ((((PerDiemCost + PerDiemExpenses) * 5) -
     (PerDiemCost * 5)) / ((PerDiemCost + PerDiemExpenses) *
     5)) * 100 AS Price_difference;
FROM Instructors
```

In both cases, the output of this query looks like this:

```
INSTRUCTORID PRICE_DIFFERENCE
------------ ----------------
         300        28.571429
         310        30.769231
         100               25
         110        28.571429
         200               25
         210

6 rows selected.
```

When you compare this to the previous output example, you can clearly see the advantage of using an alias. You will also notice that the second query is easier to read because it is immediately clear in this case what the value Price_difference represents in the query.

Aliases can also be used with concatenation expressions. For example, the following statement:

```
SELECT FirstName || ' ' || LastName || ' teaches ' ||
       InstructorType AS Instructors_by_Course
FROM Instructors;
```

returns the following result set:

```
INSTRUCTORS_BY_COURSE
------------------------------------------------------------
Michael Harrison teaches ORACLE
Susan Keele teaches UNIX
David Ungar teaches ORACLE
Kyle Jamieson teaches ORACLE
Lisa Cross teaches UNIX
Geoff Williams teaches UNIX

6 rows selected.
```

In all of these examples, the alias name cannot include an embedded space. The reason is that when Oracle parses the query, it interprets the first value after the column name as the alias and then does not know what to do with the second element. For example, the following query:

```
SELECT InstructorID,
       ((((PerDiemCost + PerDiemExpenses) * 5) -
       (PerDiemCost * 5)) / ((PerDiemCost + PerDiemExpenses) *
        5)) * 100 AS Price difference
FROM Instructors;
```

returns the following error:

```
ERROR at line 2:
ORA-00923: FROM keyword not found where expected
```

In this query, Oracle treats the equation as the column source, the word "Price" as the alias name, and then, because there are no more commas, expects the FROM clause. However, instead it finds the word "difference". This is the source of the error.

You can run into a similar problem if you include SQL keywords as alias names. For example, the word "Order" is a SQL keyword (as one-half of the ORDER BY clause). If you use "Order" as an alias name:

```
SELECT ClassID, StudentNumber, (Price * 1.1) AS Order
FROM ClassEnrollement;
```

you receive the same error. In this case, Oracle encounters the Order keyword where it does not expect to find it (because it should be at the end of the query followed by a BY <column name>).

The way to correct these problems is to enclose the entire alias name in double quotes ("). This becomes known as a *delimited identifier*. The double quotes tell Oracle that everything between the two quotes should be considered part of the same object. One other effect of using double quotes is that the alias preserves its case. Looking at the first alias example, although the alias was entered as

Price_difference, it appears in the result set as PRICE_DIFFERENCE. When you reference an alias without using double quotes, SQL automatically returns the alias in capitals (in the same way that it returns the column names and expressions in capitals).

Testing this against the earlier example, you can see the full effect of the double quotes on aliases. The following query:

```
SELECT InstructorID,
       ((((PerDiemCost + PerDiemExpenses) * 5) -
       (PerDiemCost * 5)) / ((PerDiemCost + PerDiemExpenses) *
       5)) * 100 AS "Price Difference"
FROM Instructors;
```

returns the following result set:

```
INSTRUCTORID Price Difference
------------ ----------------
         300        28.571429
         310        30.769231
         100               25
         110        28.571429
         200               25
         210

6 rows selected.
```

As you can see, the alias name preserves both the embedded space and the capitalization in the query.

Aliases are not restricted to expressions. You can create an alias for any column in the SELECT list. In the course of database design, columns are sometimes given names that are not easily interpreted. If this is the case, you can use aliases to make your output more readable. For example, this query:

```
SELECT FirstName || ' ' || Lastname AS "Name",
       InstructorType AS "Specialty",
       PerDiemCost AS "Daily Rate"
FROM Instructors;
```

produces the following output:

```
Name                      Specialty  Daily Rate
------------------------- ---------- ----------
Michael Harrison          ORACLE            500
Susan Keele               UNIX              450
David Ungar               ORACLE            600
Kyle Jamieson             ORACLE            500
Lisa Cross                UNIX              750
Geoff Williams            UNIX              400

6 rows selected.
```

In this result, each column is represented by its alias, and because each alias is enclosed in single quotes, each one retains its case and allows for embedded spaces. This makes for much more readable output.

The effect of NULL values on arithmetic and concatenation operations

The presence of NULLs in your tables can have an effect on both arithmetic and concatenation operations. A NULL is not a value. It is not considered a zero or a blank space; rather, it is the absence of a value. A NULL represents a value that is unknown or cannot be determined. If a row is inserted into a table, but a value is not assigned for a particular column, that column is determined to be null. In simple terms, a null is Oracle's way of say "I don't know." The presence of NULLs in a column can have an effect on a variety of operations including arithmetic and concatenation operator expressions.

When Oracle encounters a null in any element of an arithmetic operation, the result of the expression is always null. Because one of the elements is unknown, the result of that element added, subtracted, multiplied, or divided by another value is still unknown (that is, still null). Consider the result of the following query:

```
SELECT InstructorID, PerDiemCost, PerDiemExpenses,
       PerDiemCost + PerDiemExpenses AS "Daily Cost"
FROM Instructors;

INSTRUCTORID PERDIEMCOST PERDIEMEXPENSES Daily Cost
------------ ----------- --------------- ----------
         300         500             200        700
         310         450             200        650
         100         600             200        800
         110         500             200        700
         200         750             250       1000
         210         400

6 rows selected.
```

In this example, instructor 210 has a NULL in the PerDiemExpenses column and, as a result, also has a NULL value under Daily Cost. If we consider the null as an unknown, this makes perfect sense because 400 plus an unknown value is still an unknown value.

With the concatenation operators, the effects of nulls are slightly different. When Oracle encounters a NULL in a concatenated string, it converts the null into a blank text string. It still, however, returns the rest of the string. Consider the following query and result:

```
SELECT InstructorID || ' charges $' || PerDiemExpenses ||
      ' per day for expenses.' AS "Travel Expenses"
FROM Instructors;
```

```
Travel Expenses
-----------------------------------------------------
300 charges $200 per day for expenses.

310 charges $200 per day for expenses.

100 charges $200 per day for expenses.

110 charges $200 per day for expenses.

200 charges $250 per day for expenses.

210 charges $ per day for expenses.

6 rows selected.
```

In this example, the per diem expenses for instructor 210 are simply left blank, and the rest of the string is concatenated after the blank text string.

Eliminating duplication in the result set

When you issue a SELECT statement, Oracle returns all rows that match the query. For example, if you issue the following query:

```
SELECT City
FROM Students;
```

you receive the following:

```
CITY
------------------------------
Victoria
New York
New York
Toronto
Ottawa
New York
Dallas
San Francisco
Toronto
San Francisco
San Francisco

11 rows selected.
```

This list contains several students from the same city. What if, however, you wanted a list only of all the cities from which the company had enrolled students? To create such a list, you have to use the DISTINCT keyword. When you include this keyword,

Oracle sorts the result set and then returns only the first occurrence of each value returned by the query. To use the DISTINCT keyword, you place it in the SELECT list after the SELECT keyword. For example, to display the distinct cities in the student list, you use the following:

```
SELECT DISTINCT City
FROM Students;
```

which returns:

```
CITY
------------------------------
Dallas
New York
Ottawa
San Francisco
Toronto
Victoria

6 rows selected.
```

With this keyword, only a single occurrence of every city is returned in the result set. Notice as well that Oracle returns the data in sorted order. Oracle sorts the data to group together identical values so that duplicates can be removed.

Remember that the DISTINCT operator applies to the entire select list. If you include multiple columns in the select list, DISTINCT sorts by the unique occurrence of *all* columns. Consider the following example and result:

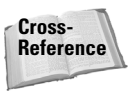
```
SELECT DISTINCT Country, City
FROM Students;

COUNTRY                        CITY
------------------------------ ------------------------------
Canada                         Ottawa
Canada                         Toronto
Canada                         Victoria
USA                            Dallas
USA                            New York
USA                            San Francisco

6 rows selected.
```

In this case, the value of Country is repeated even though it is next to the DISTINCT keyword. However, if you look at the rows for Toronto or San Francisco, you will notice that there is only one row for each of these cities. Oracle treats the two columns as a single value and returns the unique combination of the two. This can

become a problem when you include a column in the select list that does not have any duplicate values. This is the case when you add, for example, StudentNumber to the previous query:

```
SELECT DISTINCT Country, City, StudentNumber
FROM Students;

COUNTRY              CITY                 STUDENTNUMBER
-------------------- -------------------- -------------
Canada               Ottawa               1004
Canada               Toronto              1003
Canada               Toronto              1008
Canada               Victoria             1000
USA                  Dallas               1006
USA                  New York             1001
USA                  New York             1002
USA                  New York             1005
USA                  San Francisco        1007
USA                  San Francisco        1009
USA                  San Francisco        1010

11 rows selected.
```

In this example, Oracle returns the unique occurrence of Country, City, *and* StudentNumber. StudentNumber is the primary key for the table. A Primary Key is a constraint that allows you to identify unique rows by forcing all values in the column to be unique. This means that each row has a unique student number and the query will return all rows in the table. Note, however, that the DISTINCT keyword is still forcing the query to be sorted.



For details on the primary key, see Chapter 7, "Creating and Managing Oracle Database Objects."

Ordering data in the SELECT statement

✦ Sort the rows retrieved by a query

When you issue a SELECT statement in SQL, you receive a result set that contains all data in the table or tables that match the query. However, the order in which that data is returned is completely random. It is conceivably possible to issue the same query ten times and receive the same data back in ten different orders. When you want the data returned in a specific order, you must use the ORDER BY clause. With this statement, you specify the column or columns that you want as the basis of your ordering. You can also specify whether you want the data sorted in ascending order (lowest to highest) or descending order (highest to lowest). You can order by a column of any datatype except for the large object datatypes (that is, CLOB, NCLOB, BLOB, BFILE, RAW, LONGRAW, and LONG). The default for character columns is to sort alphabetically.

Suppose you wanted to list all of your instructors and their per diem costs, listing them from least to the most expensive. You could use the following query:

```
SELECT InstructorID, PerDiemCost
FROM Instructors
ORDER BY PerDiemCost ASC;
```

which returns:

```
INSTRUCTORID PERDIEMCOST
------------ -----------
         210         400
         310         450
         300         500
         110         500
         100         600
         200         750

6 rows selected.
```

The default order is ascending, and you can omit the ASC when you want the rows sorted in this manner. When you are interested in ordering data from highest cost to lowest cost, you use DESC instead of ASC.

It is also possible to sort by more than one column. When Oracle encounters an ORDER BY clause with more than one column, it orders by the first column, and within that column, orders by the second. For example, suppose you wanted to list all of the courses that you have scheduled, their locations, and their starting dates. You want the list ordered by location; however, you also want it ordered by course number for each location. You can use the following query:

```
SELECT LocationID, CourseNumber, StartDate
FROM ScheduledClasses
ORDER BY LocationID, CourseNumber;
```

which returns a result set that looks like this:

```
LOCATIONID COURSENUMBER STARTDATE
---------- ------------ ---------
       100          100 06-JAN-01
       300          100 14-FEB-01
       300          200 13-JAN-01
```

In this example, two courses are scheduled at location 300, and they are also presented in the result set in ascending order.

When you reference multiple columns, you can specify ascending or descending for each column. For example, you can issue the following statement:

```
SELECT LocationID, CourseNumber, StartDate
FROM ScheduledClasses
ORDER BY LocationID ASC, CourseNumber DESC;
```

which returns:

```
LOCATIONID COURSENUMBER STARTDATE
---------- ------------ ---------
       100          100 06-JAN-01
       300          200 13-JAN-01
       300          100 14-FEB-01
```

In this case, the result set is still sorted in ascending order by LocationID but is now sorted in descending order by CourseNumber.

When executing a SQL query, the ORDER BY statement is always executed last. For this reason, the ORDER BY clause must always be the last clause in the SELECT statement. When you put a clause after the ORDER BY, you receive an error.

Limiting Rows Using the WHERE Clause

✦ Limit the rows retrieved by a query

In all of the examples so far, the queries have limited which columns have been returned but have returned all rows in the table. In most cases, however, when you work with the data in you database, you are interested in only certain data. For example, order entry personnel will rarely want to know the prices of all products simultaneously but, more likely, will want to know the price of widget X (especially when they have a client on the phone inquiring about the price of this product). In SQL, you limit the number of rows returned with the use of a WHERE clause. In a SELECT statement, the WHERE clause is always placed after the FROM clause:

```
SELECT [DISTINCT] {* | column,[expression], . . . }
FROM table;
[WHERE condition1 [{AND | OR  [NOT]} conditon2 . . .] ]
```

When SQL processes the WHERE clause, it tests the value of each row in a column (or the result of each row in an expression) against a particular value. The query includes only rows that meet the condition in the WHERE clause. Each condition is set with the use of an *operator.* An operator is a keyword or symbol that instructs Oracle to perform an action. You have already seen two sets of operators in this chapter (the arithmetic and concatenation operators). You can also link multiple conditions using another set of operators: the logical operators, which are covered later in this chapter. You will require a complete understanding of these operators to complete the exam successfully (and to work effectively with SQL).

Comparison operators

The most common set of operators used in WHERE clauses are the comparison operators. As their name suggests, these operators compare the values in a column or column expression to a particular value. Each comparison operator can compare a column to *only one* value. The comparison operators are listed in Table 2-2.

Table 2-2
The Comparison Operators in SQL

Operator	*Meaning*
=	Equal to
<> or !=	Not equal to
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to

Currently both the <> and != symbols can be used for "not equal to." However, != was borrowed from C and was never adopted into the SQL-92 standard. As Oracle further adopts the standard (considering that SQL-99 is beginning to be implemented), it is possible that this symbol may be removed from a later version of Oracle. For that reason, it is best to stay with the ANSI "not equal to" operator (<>).

These operators are used in the following manner:

```
WHERE <column | expression> operator <value>
```

It is conventional to show the column on the left and the comparison value on the right.

For example, when you want a list of all instructors who charge more than $500 per day, you can use the following query:

```
SELECT InstructorID, PerDiemCost
FROM Instructors
WHERE PerDiemCost > 500;
```

which returns the following result:

```
INSTRUCTORID PERDIEMCOST
------------ -----------
         100         600
         200         750
```

The result set returns only those rows that meet the condition set in the WHERE clause.

Using comparison operators with character and date data

When you refer to character and date values in a WHERE clause, you must enclose them in single quotes. Numeric data does not require single quotes. If you omit the single quotes, Oracle attempts to interpret the value as a schema object, and you receive an error.

For example, when you issue this statement:

```
Select InstructorID, FirstName, LastName
FROM Instructors
WHERE Firstname = Susan;
```

you receive the following error:

```
ERROR at line 3:
ORA-00904: invalid column name
```

When Oracle executes this statement, it assumes that "Susan" is the name of another column and then looks in the table referenced in the FROM clause for a column called "Susan". Because the Instructors table doesn't contain a "Susan" column, Oracle returns the preceding message, explaining that the column name is invalid.

If you issue a query such as:

```
Select InstructorID, FirstName, LastName
FROM Instructors
WHERE Firstname = LastName;
```

no syntax error occurs. Oracle simply looks in the FirstName and LastName columns and returns any row that has the same value in both columns. In this particular table, you get a "no rows selected" message. This is not a syntax error. Rather, Oracle is informing you that no rows matched your WHERE clause condition.

When you place single quotes around these values, they become literal values. This has an important effect on how Oracle processes these statements. At the beginning of this chapter, we mentioned that SQL statements are not case sensitive. However, when you deal with literal values, case becomes an issue. Let's look at the effect of single quotes on character data and then at its impact on date data.

When Oracle stores character data in tables, it stores that data in whatever case it was inserted. When you want to reference this data, you must, therefore, reference it in the same case that it is stored. For example, when you want a list of the instructors from Toronto, you might try the following query:

```
SELECT Instructorid, LastName, City
FROM Instructors
WHERE City = 'toronto';
```

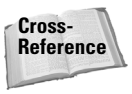
You receive the following message:

```
no rows selected
```

You receive this message because, in the table, data in the city column is stored with initial capital letters, as you can see from the following statement:

```
SELECT Instructorid, LastName, City
FROM Instructors
WHERE City = 'Toronto';

INSTRUCTORID LASTNAME                       CITY
------------ ------------------------ ----------------------
         300 Harrison                 Toronto
         310 Keele                    Toronto
         210 Williams                 Toronto
```

In this case, Oracle is able to find data that matched the character string in the WHERE clause. If you are not sure of the case your data is stored in, you can use certain character functions to modify case.



Character and other functions are discussed in Chapter 3, "Using Single- and Multi-Row Functions."

You can use any of the comparison operators with character data. When you use the greater than or less than sign, Oracle returns all values that are alphabetically greater or less than a particular value.

Consider the following example:

```
SELECT FirstName
FROM Instructors
WHERE FirstName < 'Lisa';

FIRSTNAME
------------------------------
David
Kyle
Geoff
```

In this case, all of the instructor names that are alphabetically less than Lisa are returned. The two names that are not returned are Michael and Susan, both alphabetically greater than Lisa.

There is an issue with date data in the WHERE clause that has to do with how Oracle handles date data. As was mentioned earlier in this chapter, Oracle stores date data in an internal numeric format. However, as human beings, we are not as adept at mentally converting numeric values in days, months, and years. For this reason, Oracle performs the conversion for us. The format that it uses is determined by the configuration of the National Language Support (NLS) date format. In a default installation, the NLS_DATE_FORMAT value is set to "DD-MON-YY" — that is, the day, the first three letters of the month, and a two-digit year, each separated by a hyphen.

If you are not sure what your NLS date format is, you can determine it by querying a system view: V$NLS_Parameters. Use the following query:

```
SELECT value
FROM V$NLS_Parameters
WHERE parameter = 'NLS_DATE_FORMAT';
```

When you reference date data in the WHERE clause of a query, Oracle compares the date string against the NLS_DATE_FORMAT value to convert it into its internal format. If it cannot interpret the date string from this format, it returns an error. For example, suppose you want to know which course is scheduled to start on January 6, 2001 (assuming your Oracle database is using the default data parameter setting). The following query returns data:

```
SELECT CourseNumber
FROM ScheduledClasses
WHERE StartDate = '06-JAN-01';
```

whereas, this query returns the following error:

```
SELECT CourseNumber
FROM ScheduledClasses
WHERE StartDate = 'January 6, 01';

ERROR at line 3:
ORA-01858: a non-numeric character was found where a numeric
was expected
```

The "non-numeric character" in this example is the word "January" where Oracle expected to find the two-digit day value (DD-MON-YY). Because Oracle is unable to convert this string into its internal date format, it is unable to process the WHERE condition and returns the error.

One interesting point about date data used in a WHERE clause is that, unlike character data, it is not case sensitive. For example, Oracle resolves all of these WHERE clauses the same, to the same internal date value:

```
WHERE StartDate = '06-JAN-01'
WHERE StartDate = '06-Jan-01'
WHERE StartDate = '06-jan-01'
```

Using comparison operators with expressions

The comparison operators in SQL are not limited to comparing column values. They can also be used to compare the results of a derived column. In an earlier example, the weekly cost of each instructor was calculated using the following query:

```
SELECT InstructorID, (PerDiemCost + PerDiemExpenses) * 5 As
Weekly_Cost
FROM Instructors;
```

What if, however, you want only those instructors whose weekly cost is greater than $3,500? To find this information, you simply place the expression in the WHERE clause:

```
SELECT InstructorID, (PerDiemCost + PerDiemExpenses) * 5 As
Weekly_Cost
FROM Instructors
WHERE (PerDiemCost + PerDiemExpenses) * 5 > 3500;

INSTRUCTORID WEEKLY_COST
------------ -----------
         100        4000
         200        5000
```

When you perform queries with an expression in the WHERE clause, you must include the entire expression. You cannot place the alias name in its place.

For example, the following query returns an error:

```
SELECT InstructorID, (PerDiemCost + PerDiemExpenses) * 5 As
Weekly_Cost
FROM Instructors
WHERE Weekly_Cost > 3500;

ERROR at line 3:
ORA-00904: invalid column name
```

This error is generated because the alias is not applied until the result set is generated, and the result set is not generated until *after* the WHERE clause has been evaluated. By placing the alias name in the WHERE clause, you are referencing an object that does not yet exist.

You can, however, place aliases in the ORDER BY statement because the result set must be generated before it can be ordered, and thus, the alias exists before the ORDER BY is evaluated.

Using logical operators in WHERE clauses

It is possible to have more than one condition in a WHERE clause. However, to combine conditions, you must use a logical operator. There are three logical operators in SQL (see Table 2-3), and each has a different effect on the outcome of a WHERE clause.

Table 2-3
SQL Logical Operators

Operator	*Meaning*
AND	Returns a row when *both* conditions are TRUE.
OR	Returns a row when *either* condition is TRUE.
NOT	Returns a row when the condition is FALSE.

The AND operator

The AND operator is used to join two or more conditions in one query. In order for a row to be returned in this condition, all of the conditions linked together must return TRUE. Consider the following query:

```
SELECT InstructorID, City, PerDiemCost
FROM Instructors
WHERE City = 'New York' AND PerDiemCost > 500;

INSTRUCTORID CITY                          PERDIEMCOST
------------ ------------------------------ -----------
         100 New York                               600
```

In this table, three instructors are from New York, and two instructors charge more than $500 per day. However, only one instructor, instructor 100, satisfies *both* conditions. The other instructors either charge less than $500 per day or live somewhere other than New York. Note that when either condition returns a NULL, no rows are returned. In the Instructors table, instructor 210 is the only instructor with a PerDiemCost of $400. He has a NULL PerDiemExpense. If you were to execute the following query:

```
SELECT InstructorID, PerDiemCost, PerDiemExpenses
FROM Instructors
WHERE PerDiemCost = 400 AND PerDiemExpenses >= 200;
```

no rows are returned. Because a NULL is returned for the PerDiemExpenses, the second condition does not evaluate to TRUE, and therefore this row is not returned.

The OR operator

When you combine two or more conditions with the OR operator, Oracle returns any row that meets either condition. If you take the first example for the AND operator and use the OR operator instead, you notice a significant difference between the two:

```
SELECT InstructorID, City, PerDiemCost
FROM Instructors
WHERE City = 'New York' OR PerDiemCost > 500;

INSTRUCTORID CITY                           PERDIEMCOST
------------ ------------------------------ -----------
         100 New York                               600
         110 New York                               500
         200 Palo Alto                              750
         410 New York                               400
```

In this case, Oracle returns the two instructors from New York who charge less than $500 dollars per day and the one instructor who is not from New York but charges more than $500. You may have several conditions linked by the OR operator, and a row is returned as long as one of the conditions returns TRUE.

For this reason, columns that return a NULL value do not have the same effect on OR queries as they do with the AND operator. With the OR, as long as one of the other conditions is met, the row is returned even when the other condition returns a NULL. When you execute the second example in the previous section with an OR instead of an AND, you see the difference:

```
SELECT InstructorID, PerDiemCost, PerDiemExpenses
FROM Instructors
WHERE PerDiemCost = 400 OR PerDiemExpenses > 100;

INSTRUCTORID PERDIEMCOST PERDIEMEXPENSES
------------ ----------- ---------------
         300         500             200
         310         450             200
         100         600             200
         110         500             200
         200         750             250
         210         400

7 rows selected.
```

In this case, instructor 210 is the only instructor who charges $400 per day, but six other instructors charge more than $100 for daily expenses.

The NOT operator

Unlike the AND and OR operators, the NOT operator is not used to set multiple conditions on a WHERE clause. Instead, it is a negative operator. It negates a condition so that it returns all rows that do not meet the condition. The NOT operator cannot be used with the standard comparison operators, but it is used with some of the other WHERE clause operators, which are discussed later in this chapter.

It is possible to select by negative conditions using the NOT operator; however, it should be avoided when possible. Positive queries tend to be more efficient than negative queries because they can take better advantage of indexes. Negative queries tend to result in more table scans. If you find yourself writing a negative query, check to see if it is possible to rewrite it as a positive query. If not, the negative is syntactically acceptable.

The rules of precedence for logical operators

Just as with the arithmetic operators, rules of precedence govern the operators. The order is shown in Table 2-4.

Table 2-4
Order of Precedence for Logical Operators

Operator	*Order of Evaluation*
Parentheses	1
All comparison operators	2
NOT	3
AND	4
OR	5

As with the arithmetic operators, the order of precedence can be overridden with parentheses. It is important to keep these rules in mind; otherwise, you can run into serious problems.

Consider the following situation. Cust_list is a master customer table for your company that contains over 1,000,000 customers from across the continent. You are interested in getting the address and phone number of everyone named "Smith" that lives in either New York or Chicago, and you issue the following query:

```
SELECT Name, Address, City, State, Postal, Telephone
FROM Cust_list
WHERE Name = 'Smith'
      AND CITY = 'New York' or City = 'Chicago';
```

When you execute this query, you are in for a surprise. Why? Consider the order of precedence. When Oracle parses this query, it parses the AND condition first (Name = 'Smith' AND City = 'New York') and then applies the OR condition (City = 'Chicago'). In other words, this query returns all customers named "Smith" living in New York and every customer living in Chicago! This problem can be resolved with the use of parentheses:

```
SELECT Name, Address, City, State, Postal, Telephone
FROM Cust_list
WHERE Name = 'Smith'
      AND (CITY = 'New York' or City = 'Chicago');
```

In this example, Oracle resolves the OR condition first (generating a list of all customers in New York and Chicago) and then applies the AND operator to the result, linking it to the first WHERE clause condition.

For this exam, you are expected to fully understand the order of precedence for the logical operators. You may be presented with complex scenarios involving logical operators in nested parentheses. Make sure you practice reading and interpreting these types of queries. See question 6 for an example of this in the Assessment Questions at the end of the chapter.

Additional comparison operators in the WHERE clause

Other comparison operators are available in the WHERE clause (see Table 2-5). Some of them are shortcuts that automatically combine several conditions; others help you overcome some inherent problems in working with the WHERE clause.

Table 2-5
Addition Comparison Operators

Operator	*Meaning*
BETWEEN ... AND	Returns all rows between two values.
IN	Returns any row matching a list of values.
LIKE	Allows two rows based on a character pattern.
IS NULL	Returns all rows where value is NULL.

You should be familiar with each of these operators.

Using the BETWEEN ... AND operator

When limiting rows, you may sometimes want to limit by a range of values. For example, suppose you have a list of orders stored in an Order_Info table. For each row in this orders table, you have an order date column. If you wanted a list of all the sales for the first calendar quarter, how would you retrieve the data? You would probably write a query like this:

```
SELECT *
FROM Order_Info
WHERE orderdate >= '01-JAN-01' AND orderdate <= '30-MAR-01'
```

In this case, you have to use >= and <= rather than simply > or <; otherwise, you would not return any orders placed on the first of the last day of the quarter. SQL provides a shortcut for this type of query: the BETWEEN ... AND operator.

The BETWEEN operator enables you to write a query that returns a range of values in a single statement. For example, you can rewrite the last query as follows:

```
SELECT *
FROM Order_Info
WHERE orderdate BETWEEN '01-JAN-01' AND '30-MAR-01'
```

It is important to remember that the BETWEEN operator returns any value between and including the upper and lower limits. For example, when you want a list of all instructors who charge more than $400 per day but less than $700, you cannot use the following query because it includes those who make exactly $400:

```
SELECT InstructorID, PerDiemCost
FROM Instructors
WHERE PerDiemCost BETWEEN 400 and 700;

INSTRUCTORID PERDIEMCOST
------------ -----------
         300         500
         310         450
         100         600
         110         500
         210         400
```

In order to eliminate instructor 210, you have to set your WHERE condition to BETWEEN $401 AND $700.

You also want to make sure that you place the lower range value first in the BETWEEN statement. For example, the following query returns no rows:

```
SELECT InstructorID, PerDiemCost
FROM Instructors
WHERE PerDiemCost BETWEEN 700 and 400;
```

The query returns no rows because Oracle interprets the WHERE clause as follows:

```
WHERE PerDiemCost >= 700 AND PerDiemCost <= 400
```

Because there cannot be a condition where a value can simultaneously be greater that $700 and less than $400, no rows in the query match this condition.

When you want to exclude a range, you can do so with the NOT operator. For example, the following query:

```
SELECT InstructorID, PerDiemCost
FROM Instructors
WHERE PerDiemCost NOT BETWEEN 400 and 700;
```

returns all instructors who have a per diem cost below $400 or above $700.

Using the IN operator

Sometimes you may want to retrieve data based on a list of values from the same column. For example, suppose you want to find instructors from Toronto or New York. Comparison operators accept only one value so you have to use an OR operator. The query looks something like this:

```
SELECT InstructorID, City
FROM Instructors
WHERE city = 'Toronto' OR City = 'New York';
```

This is a perfectly valid query; however, what happens when the list contains 10 or 20 items? This may lead to a great deal of typing. To accommodate this situation, SQL includes the IN operator. The IN operator enables you to compare a column or expression to a range of values. Oracle implicitly adds an OR operator between each element in the list.

If you rewrite the previous query with an IN operator, it looks like this:

```
SELECT InstructorID, City
FROM Instructors
WHERE city IN ('Toronto', 'New York');
```

You must include the list in parentheses and separate each element in the list with a comma.

You can also exclude all elements in the list using the NOT operator. For example, if you want all instructors except those from Toronto or New York, you can use the following query:

```
SELECT InstructorID, City
FROM Instructors
WHERE city NOT IN ('Toronto', 'New York');
```

In this query, Oracle interprets the WHERE condition as:

```
WHERE city <> 'Toronto' OR City <> 'New York'
```

and returns all rows that do not match either condition.

Using the LIKE operator

When dealing with large text strings, you may want to search by a part of the string rather than the full string. This kind of search cannot be performed with the comparison operators. A comparison operator can be used only when you include the *entire* string. For example, the Instructors table includes comments for each instructor. Suppose you want to know which instructors have knowledge of C++ programming; the following query does not return any data:

```
SELECT InstructorID, Comments
FROM Instructors
WHERE comments = 'C++'
```

Comments for two instructors mention C++, but in neither case is C++ the entire text string. The "equals to" sign returns rows only where the contents of the comments column match the *entire* string in single quotes. If you want to match only part of a string, you must use the LIKE operator.

The LIKE operator enables you to use wild card characters to match a portion of a character string. A wild card is simply a single character that can represent a number of different characters. The two wild cards in SQL are shown in Table 2-6.

Table 2-6
Wild Card Operators in SQL

Symbol	*Meaning*
%	Any number of characters (including none)
_	Any one character

The string value, including the wild cards, must be placed inside single quotes. You can use wild card characters on either side of the character string. You use the underscore (_) to represent a single character. For example, when you are not sure whether a name was entered in a table as LaGrand or LeGrand, you can find either value with the following condition:

```
WHERE name LIKE 'L_Grand'
```

Remember that the rest of the string is case sensitive, but the wild card value is not. When you are unsure of more than one character, you can use multiple wild cards. For example, when you are unsure whether a name is stored as Hardy or Hurdy, you can use the following:

```
WHERE name LIKE 'H__dy'
```

You can also use the percent (%) wild card, but it also returns strings that have more than or less than two characters. Compare the output of these two queries:

```
SELECT FirstName
FROM Instructors
WHERE FirstName LIKE '_a%';

FIRSTNAME
------------------------------
David
Lana

SELECT FirstName
FROM Instructors
WHERE FirstName LIKE '%a%';

FIRSTNAME
------------------------------
Michael
Susan
David
Lisa
Lana
```

As you can see, the first query returned only those names that have a lowercase "a" as the second letter, whereas the second returned all names that contain an "a".

Remember that the % wild card also includes cases where there is no character. For example:

```
SELECT FirstName
FROM Instructors
WHERE FirstName LIKE '%A%';
```

returns:

```
FIRSTNAME
------------------------------
Adele
```

This also demonstrates the case-sensitive nature of the rest of the string. The percent wild card is used most often when you are looking for one word in sentences stored in the database. Going back to the original example in this section, if you want to find instructors in the company that have C++ mentioned in their comments, you can use the following query:

```
SELECT InstructorID, Comments
FROM Instructors
WHERE comments LIKE '%C++%';
```

This query returns the following instructors:

```
INSTRUCTORID  COMMENTS
------------  --------------------------------------------
210           Has extensive shell scripting experience.  Has
211           also programmed with C and C++

410           Has 10 years experience with C and C++.
```

Just as with IN and BETWEEN, you can search on a negative condition using a NOT operator:

```
SELECT InstructorID, Comments
FROM Instructors
WHERE comments NOT LIKE '%C++%';
```

When using the LIKE and NOT LIKE operator, you should try to avoid using wildcards at the beginning of the string. Placing a wildcard at the beginning of a character string forces Oracle to ignore any indexes.

Using the IS NULL operator

NULL fields present a particular problem when working with comparison operators. Because a NULL is an unknown value, it is logically impossible to compare it to any other values using a comparison operator. Whenever you use a comparison operator with NULL, the result is always NULL. In a query, this means that no rows are returned. At times, however, you want to limit the number of rows by a NULL in a column. You may want to search for clients who do not have a fax number listed in their client record or for employees who do not have Social Security numbers or department IDs. Each one of these examples requires you to find rows where the column is NULL.

To get around this problem, SQL includes the IS NULL operator. The IS NULL operator tests each value in a column and determines whether the cell has a value. When it does not, the row is returned. When the field has a value, it is ignored, and Oracle moves to the next row. This is the only way to retrieve rows based on the presence of a NULL.

Here is an example. The PerDiemExpenses column for some instructors has a NULL. When you want a list of these instructors, you can use the following query:

```
SELECT InstructorID, PerDiemExpenses
FROM Instructors
WHERE PerDiemExpenses IS NULL;

INSTRUCTORID PERDIEMEXPENSES
------------ ---------------
         210
         410
```

As with the other operators in this chapter, you can also find all rows that do not contain NULL values by using the NOT operator:

```
SELECT InstructorID, PerDiemExpenses
FROM Instructors
WHERE PerDiemExpenses IS NOT NULL;

INSTRUCTORID PERDIEMEXPENSES
------------ ---------------
         300             200
         310             200
         100             200
         110             200
         200             250
         450             200

6 rows selected.
```

Including the ROWNUM pseudo-column in the WHERE clause

Aside from column data and expressions, Oracle includes a number of pseudo-columns that you can reference as part of the select list. A pseudo-column acts as if it were a column in a table but does not actually exist within the table schema, and the values produced by the pseudo-column are not stored in the database. You can reference a pseudo-column in conjunction with any table. One of these pseudo-columns is the ROWNUM column. ROWNUM returns a numerical value that indicates the order in which the data was returned. For example, the following statement:

```
SELECT ROWNUM, InstructorID, City
FROM Instructors;
```

returns a result set that looks something like this:

```
   ROWNUM INSTRUCTORID CITY
--------- ------------ --------------------------
        1          300 Toronto
        2          310 Toronto
        3          100 New York
        4          110 New York
        5          200 Palo Alto
        6          210 Toronto

6 rows selected.
```

The value of the pseudo-column is not physically stored anywhere; rather, it is generated as the result set is generated. If you change the rows that are returned, the values of the ROWNUM column change with them. For example, compare the result of the previous query with this query and result set:

```
SELECT ROWNUM, InstructorID, City
FROM Instructors
WHERE City = 'Toronto';

   ROWNUM INSTRUCTORID CITY
--------- ------------ --------------------------
        1          300 Toronto
        2          310 Toronto
        3          210 Toronto
```

Notice that the ROWNUM value for instructor 210 changes from 6 to 3 in this case. The reason for this value change is simple: The row containing instructor 210 is now the third row in the result set. Instructor 210 is still the sixth row in the table, but the ROWNUM pseudo-column is concerned only with the rows in the result set.

You can use this behavior to your advantage when writing SELECT statements. For example, if you want only the first four instructors listed, you can use the following statement:

```
SELECT ROWNUM, InstructorID, City
FROM Instructors
WHERE ROWNUM < 5;
```

which returns the following:

```
   ROWNUM INSTRUCTORID CITY
--------- ------------ ------------------------------
        1          300 Toronto
        2          310 Toronto
        3          100 New York
        4          110 New York
```

As each row is returned, it is assigned a ROWNUM. That ROWNUM is then tested against the WHERE clause. When a row is rejected, the next row is given the same ROWNUM as the row before it. It too is rejected, and on it goes until you reach the end of the result set.

This behavior has one side effect. You cannot select a row number that is *greater than* a particular number. For example, if you want all but the first two rows and you execute the following statement:

```
SELECT ROWNUM, InstructorID, City
FROM Instructors
WHERE ROWNUM > 2;
```

you receive the following message:

```
no rows selected
```

The reason is that when the first row is returned, it is given the ROWNUM value of 1. It is tested against the WHERE clause and then rejected. The next row is then given the ROWNUM value of 1. It is tested and also rejected. This continues until no more rows are in the table, and no rows are returned in the result set.

There is one final effect of the order of operation with the ROWNUM column: It is evaluated *before* the ORDER BY clause. This order can be seen in the result to the following query:

```
SELECT ROWNUM, InstructorID, PerDiemCost
FROM Instructors
ORDER BY PerDiemCost DESC;

   ROWNUM INSTRUCTORID PERDIEMCOST
--------- ------------ -----------
        5          200         750
        3          100         600
        1          300         500
        4          110         500
        2          310         450
        6          210         400

6 rows selected.
```

In this case, the order of the ROWNUM column demonstrates that the values for this column are generated before the order by was applied; otherwise, they would be listed in ascending order. This means that this statement cannot be used to find the three highest paid employees:

```
SELECT ROWNUM, InstructorID, PerDiemCost
FROM Instructors
WHERE ROWNUM < 4
ORDER BY PerDiemCost DESC;
```

```
    ROWNUM INSTRUCTORID PERDIEMCOST
---------- ------------ -----------
         3          100         600
         1          300         500
         2          310         450
```

When you compare this result set to the previous result, you can see that what is returned is not the three most expensive instructors, but rather the first three instructors encountered, ordered by their per diem costs. To solve this problem, you have to either determine the value programmatically (using, for example, PL/SQL) or use aTop-N query with a subquery in the FROM clause (this topic is examined in Chapter 4, "Advanced SELECT Statements").

Key Point Summary

When you want to retrieve data from your database, you must query the database using a SQL SELECT statement. The SELECT statement is made up of multiple parts or clauses but must always contain a SELECT list and a FROM clause.

The SELECT list can contain multiple elements:

- ✦ The SELECT list contains a list of the columns you want returned.
- ✦ When you want to return all columns, use the asterisk (*).
- ✦ When you want to remove duplicates, use the DISTINCT keyword.
- ✦ Values in the select list can also be derived using arithmetic expressions or concatenation operators.
- ✦ Arithmetic operators can be used with numeric or date data. When you add or subtract a number from a date, the result is a date. When you subtract two dates, the result is the number of days between the two dates.
- ✦ When Oracle calculates an arithmetic expression, it evaluates parentheses, then multiplication and division, and then addition and subtraction.
- ✦ A column created by an expression has a column heading that is the same as the expression.
- ✦ You can change any column heading by using a column alias.

A basic SELECT statement returns all rows in a table. It returns this data in a random order. When you want to order the data returned by the query, you must include an ORDER BY clause. When using an ORDER BY clause, remember:

- ✦ The ORDER BY clause must always be the last clause in the SQL statement.
- ✦ You can order by one or more columns.
- ✦ You can specify ascending (ASC) or descending (DESC) sorted order.

When you want to limit the number of rows returned, you must use a WHERE clause. The WHERE clause uses a number of operators to set conditions for which rows a query returns:

- ✦ The WHERE clause must follow the FROM clause.
- ✦ WHERE clauses compare a column to a condition (in the form WHERE <col-name> <condition>.
- ✦ The condition can use any of the legal comparison operators.
- ✦ All rows in the table are tested against the condition. Only those rows that satisfy the condition are returned in the result set.
- ✦ Character and date data must be enclosed in single quotes in a WHERE clause.
- ✦ Character data enclosed in single quotes is treated as case sensitive.
- ✦ Date data must match the NLS date format in order for Oracle to interpret it correctly without the use of functions.
- ✦ You can link more than one condition in a WHERE with a logical operator (OR, AND).
- ✦ In the case of multiple logical operators, Oracle evaluates the AND conditions first and then the OR conditions.
- ✦ Parentheses can be used to override the behavior described in the preceding bullet.
- ✦ NULL values cannot be tested for with a conditional operator. You must use the IS NULL to test for NULLS.

✦ ✦ ✦

STUDY GUIDE

In this chapter, we have looked at all of the elements of the basic SELECT statement. The exam tests very heavily on the SQL query language, and you will be required to have a firm grasp of the elements of the SELECT statement to successfully complete this exam. You can test your understanding with the sample questions, scenarios, and exercises that follow.

Assessment Questions

1. You issue the following query against the Instructors table:

```
SQL> SELECT InstructorID, FirstName || ' ' || LastName AS
2    Employee Name
3    FROM Instructors
4    WHERE city = 'new york';
```

Which line in this statement will cause an error?

A. 1

B. 2

C. 3

D. 4

2. For which of the following tasks would you use the ROWNUM function in a SELECT list. Select two.

A. Return the four highest salaries determined with an ORDER BY clause.

B. Return the first ten rows of the result set.

C. Number all of the rows in a result set.

D. Return all rows except the first ten.

3. Which of the following WHERE clauses would you use to find all courses running after January 1, 2001 but before March 31, 2001? Choose all that apply.

A. WHERE CourseDate BETWEEN '01-JAN-01' and '31-MAR-01'

B. WHERE CourseDate > ='01-JAN-01' AND CourseDate <= '31-MAR-01'

C. WHERE CourseDate BETWEEN 'JAN-01-01' and 'MAR-31-01'

D. WHERE CourseDate > '01-JAN-01' AND CourseDate < '31-MAR-01'

4. Which expression will be evaluated first in the following query?

```
SELECT ColA * (ColB /(ColC + (ColD * ColE/2)) / ColF)
FROM TestTable;
```

A. ColD * ColE

B. ColE/2

C. <exp> / ColF

D. ColC + <exp>

5. You issue the following query against the Instructors table:

```
SQL>SELECT InstructorID, Firstname || Lastname
2   "Instructor Name",
3 (PerDiemCost + PerDiemExpenses) x 5
4 FROM Instructors
5 WHERE City = 'New York';
```

Which line in this statement will cause an error?

A. 1

B. 2

C. 3

D. 4

6. Consider the following query:

```
SELECT FirstName, LastName, City
FROM Instructors
WHERE FirstName Like 'K%' OR LastName LIKE 'C%' AND Firstname
LIKE 'L%' AND City = 'Toronto' OR city = 'New York';
```

Which of the following rows would be returned by this query? Choose all that apply.

A. Lana, Chiu, New York

B. Lisa, Cross, Palo Alto

C. David, Unger, New York

D. Kyle, Jamieson, New York

7. Which of the following WHERE clauses will return only rows that have a NULL in the PerDiemExpenses column?

A. WHERE PerDiemExpenses <> *

B. WHERE PerDiemExpenses IS NULL

C. WHERE PerDiemExpenses = NULL

D. WHERE PerDiemExpenses NOT IN (*)

8. You issue the following query:

```
SELECT FirstName
FROM StaffList
WHERE FirstName LIKE '_A%'
```

Which names would be returned by this query? Choose all that apply.

A. Allen

B. CLARK

C. JACKSON

D. David

9. You execute the following query:

```
SELECT (PerDiemCost + PerDiemExpense) * 5
FROM Instructors
WHERE InstructorID =  210;
```

If instructor 210 has a PerDiemCost of $400 and a PerDiemExpenses that is NULL, what will be the outcome of the arithmetic expression?

A. 0

B. NULL

C. 2000

D. The query will return an error.

10. You execute the following query:

```
SQL>SELECT InstructorID AS "Instructor Number",
2          FirstName || ' ' || LastName AS
3          "Employee", City, PerDiemCost * 5 AS
4          "Weekly Charge"
5    FROM Instructors
6    WHERE PerDiemCost = $400
7    ORDER BY City ASC, "Weekly Charge" DESC
```

Which line will return an error?

A. 2

B. 5

C. 6

D. 7

Scenarios

1. You are writing a report on the courses that you have scheduled for your various centers. The majority of data required for this report is held in the ScheduledClasses table.
   - **A.** How can you find the information for only the New York office?
   - **B.** The ScheduledClasses table contains the start date for each course and the number of days for each course. However, for the purpose of the report, you want both the start and end date for each course. How would you write a query that generated the ending date for each course?
   - **C.** For the report, you want the title of the new column to be EndDate. What is the default column name, and how would you change the column name?
   - **D.** What would be the value of the End Date column if one of the courses had a NULL in the StartDate column?
2. You have been asked to generate a report on your company's various locations.
   - **A.** How would you write a query to list only the U.S. sites?
   - **B.** Some of the locations are near the subway. How would you find those locations?
   - **C.** How would you find only U.S. sites that are close to the subway?
   - **D.** How could you list all of your locations ordered alphabetically by city?

Lab Exercises

Lab 2-1 Working with basic queries

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.
2. Write a query that returns all rows and columns from the Students table.
3. How many rows are returned? _____
4. Modify your query so that it returns only the first name, last name, and city for each student.
5. Write a second query that only returns each city only once.
6. Do you need to order this list with an ORDER BY clause?

Lab 2-2a Using expressions in SQL queries: the concatenation operator

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.
2. Rewrite the first query from the pervious exercise so that it returns only two columns, one that contains both the first and last name in the following format <LastName, FirstName> and one that contains the city information.
3. What is the name of the new column? _____
4. Rewrite the query so that the two columns are called Student Name and Home City.
5. Rewrite the query so that it lists the students alphabetically by last name.

Lab 2-2b Using expressions in SQL queries: the arithmetic operator

1. Using the Instructors table, write a query that lists each instructor by ID and last name and includes a third column that is each instructor's per diem cost times five days. Name this third column weekly cost.
2. Modify this query to add a fourth column that adds the per diem cost and the per diem expense and multiply this value by 5. Name the fourth column travel cost.
3. Rewrite this query so that it lists the instructors from the highest travel cost to the lowest.

Lab 2-3 Limiting the rows returned with a WHERE clause

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.
2. Write a query that returns all instructors who have a per diem cost higher than $400 but lower than $600.
3. Write a query that lists only instructors who live in Toronto or New York.
4. Rewrite this query so that it lists all instructors who have a per diem cost higher than $400 but lower than $600 and who live in Toronto or New York.
5. Write a query that lists the InstuctorID for instructors that do not have a NULL in their per diem expenses column.

Lab 2-4 Using the ROWNUM pseudo-column

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.
2. Write a query that returns the first and last name of each instructor, the per diem cost, and gives each of them a number.
3. Rewrite this query to return only the first four instructors.
4. Rewrite the query to return all instructors and order them by their per diem cost.
5. Check to see if the generated row numbers are still in numeric order.

Answers to Chapter Questions

Chapter Pre-Test

1. All SQL queries must have a SELECT list and a FROM clause. That is, you must supply *what* you are selecting and from *where*.
2. You can find one occurrence of a repeating value by using the DISTINCT keyword as part of the SELECT list. For example, if you want a list of the dates when courses are scheduled, you can use the following query:

   ```
   SELECT DISTINCT StartDate
   FROM ScheduledClasses;
   ```

3. In an arithmetic operation, multiplication and division operators are evaluated before addition and subtraction operators.
4. When you subtract two dates from each other, the result is a number that represents the number of days between the two dates. If you subtract the earlier date from the later date, the result is a negative number.
5. You can find date data by quarter one of two ways. Either you can have two conditions linked by an AND:

   ```
   WHERE order_date >= '01-JAN-01' AND order_date <= '31-Mar-01'
   ```

   or you can use the BETWEEN operator:

   ```
   WHERE order_date BETWEEN '01-JAN-01' AND '31-Mar-01'
   ```

   The BETWEEN clause is simply a shorthand so you don't have to rewrite the column name. When Oracle evaluates the BETWEEN clause, it converts it back to the Boolean comparison before it executes the query.

6. You can find rows that are not in a range by using the NOT operator with the BETWEEN operator. If you take the previous example, to find all rows not in the first quarter of 2001, you can use the following query:

   ```
   WHERE order_date NOT BETWEEN '01-JAN-01' AND '31-Mar-01'
   ```

   The NOT operator returns all rows that do not meet the BETWEEN condition.

7. You can override the order of precedence for any operator by using parentheses. You can also nest parentheses. In this case, the innermost operators are executed first.

8. If you have a NULL anywhere in an arithmetic expression, the result is NULL. NULL is an unknown value, and any number treated arithmetically with this value also is unknown.

9. To find a NULL field with a WHERE clause, you must use the IS NULL operator. All of the other WHERE clause operators ignore NULL fields.

10. By default, the name of any column derived from an expression is the actual expression itself. You can change this behavior through the use of a column alias.

Assessment Questions

1. **B** — The error in this query occurs at line 2. Because an embedded space is in the alias name, you must enclose it in double quotes. If you do not, Oracle assumes that the first element of the alias name is the alias and then is unable to process the second element. Spacing does not matter in any SQL statement, and the fact that the alias is on a different line from the column name has no bearing on the query as long as commas enclose the two. The fact that New York is in all lowercase characters is not in itself a syntax error, but it may cause no rows to be returned.

2. **B** and **C** — The ROWNUM function numbers each row in the result set as it is returned. Therefore, it can be used to number each row. It can also be used in a WHERE clause to limit the number of rows returned. However, the ROWNUM value is generated before the ORDER BY clause is applied so you cannot use it to find the four highest salaries, only the salaries of the first four rows returned by the query. You also cannot limit ROWNUM by the lower range. This results in no rows being returned because each row is assigned a ROWNUM value of 1 and then rejected by the WHERE clause. For more information, see Chapter 4, “Advanced SELECT Statements.”

3. **D** — Only answer D is correct. Both A and B are syntactically correct; however, both return courses running *on* both January 1 and March 31. The question does not ask for courses running on these two dates. Answer C uses an incorrect NLS date format and returns a syntax error.

4. **A —** Arithmetic expressions are evaluated from the innermost parentheses outward. However, in this example, two operations in the innermost parentheses have the same level of precedence (ColD * ColE /2). In this case, Oracle processes the elements from left to right, and therefore, the multiplication is evaluated first (ColD * ColE). This result is divided by 2. The query then works its way outward until it has a final value.

5. **C —** The error in this query is in line 3. The multiplication operator in SQL is the asterisk (*), not an "x". In this example, Oracle assumes "x" is the alias for the column and then is unable to process the 5.

6. **A, C, D —** Oracle evaluates AND conditions first and then OR conditions. Therefore, Oracle evaluates this query based on three criteria:

```
WHERE LastName LIKE 'C%' AND Firstname LIKE 'L%' AND City =
'Toronto'

OR WHERE FirstName Like 'K%'
OR WHERE City = 'New York'
```

   Both Lana Chiu and Lisa Cross have last names starting with "C" and first names starting with "L"; however, neither one is from Toronto. Therefore, none of them meet the first condition. Lana Chiu is returned only because she lives in New York and meets the final condition. Remember that with an AND operator, *all* conditions must be TRUE for a row to be returned, but with the OR operator, only one condition must return TRUE to return the row. For more information, see the section "Using logical operators in WHERE clauses," earlier in this chapter.

7. **B —** Only answer B returns rows that contain a NULL. Because a NULL is an unknown value, you cannot use a comparison operator to find a NULL (that is, you cannot look for rows that equal an unknown value). For this reason, SQL includes the IS NULL operator. You also cannot use the asterisk in WHERE clauses to mean all values. It can be used only in the select list to mean all columns. For more information, see the section "Using the IS NULL Operator," earlier in this chapter.

8. **C —** Two wildcards are used with the LIKE operator. The underscore (_) stands for any one character of any case, and the percent sign (%) stands for any number of characters of any case including none. Because this string starts with an underscore rather than a percent sign, it won't return Allen or Clark because they represent zero and two characters before the "A". If the LIKE string had been "%A%", both of these values would have been returned. David was not returned because all non-wild card characters are case sensitive. Therefore, only strings with an uppercase "A" as their second letter are returned. For more information, see the section "Using the LIKE Operator," earlier in this chapter.

9. **B —** Whenever a NULL value is part of an arithmetic expression, the result is always NULL. A NULL is an unknown value, not a zero. This does not register as a syntax error because the query is syntactically correct. The NULL is the expected result. For more information, see the section "The Effect of NULL Values on Arithmetic and Concatenation Operations," earlier in this chapter.

10. **C —** The error in this query is on line 6. Specifically, the author of the query has put a dollar sign next to the per diem cost value. Oracle is expecting a numeric value for this column. Instead, it has received a string value that is not enclosed in single quotes. Line 7 does not contain an error. You should remember that you can use aliases in the ORDER BY clause but not in a WHERE clause because the aliases are generated after the WHERE clause is evaluated but before the ORDER BY clause is evaluated.

Scenarios

1. There is no city column in this table; however, the ScheduledClasses table does list courses by location ID. In order to list courses in the New York office, you would simply have to use a WHERE clause to limit the row returned to rows with a locationID of 100 (the ID for the New York office).

   To find the end date for a course, you simply have to add the start date and the duration. Remember that when you add or subtract a number to or from a date, the resulting value is a date that is that many days away from the original date (either forward or backward in time, depending on whether you add or subtract the number).

   The default name of the column is the expression itself. You have to use a column alias to change the name of the column. Because you want to use an alias name that has an embedded space, you also have to enclose the alias name in double quotes.

   Just as with other arithmetic expressions, when the start date field is NULL for a particular row, the resulting end date value is also NULL. The entire query would look like this:

   ```
   SELECT CourseNumber, StartDate, Startdate + DaysDuration AS
   "End Date"
   FROM ScheduledClasses
   WHERE locationID = 100
   ```

2. Because you have offices in only two countries, you can find either all rows where the country equals the U.S. or all rows where the country does not equal Canada. However, it is always best to use positive conditions rather than negative conditions. This also saves your having to rewrite the query when you open your first office in Mexico.

   Information about subways is stored in the Description column. This is a large text field; therefore, you have to use the LIKE operator to find the rows you are interested in. Also, because you do not know where in the text field the word “subway” occurs, you have to use wild cards with the LIKE operator. The code looks like this,

   ```
   SELECT LocationID, LocationName
   FROM Locations
   WHERE Description LIKE '%subway%';
   ```

You have to be careful with this code. If the description information is stored in all capital letters or an initial capital is in the word "subway" for any of the descriptions, no rows are returned for those locations.

To find only American locations near the subway, you simply add a second condition in the WHERE clause using the AND operator. If you use the OR operator, you get Canadian locations near subways and all U.S. locations (whether or not they are near a subway). The query should look like this:

```
SELECT LocationID, LocationName
FROM Locations
WHERE Description LIKE '%subway%' AND Country =
      'USA';
```

Finally, to list all locations alphabetically by city, you simply include the following ORDER BY clause:

```
ORDER BY city
```

The ORDER BY clause orders text columns alphabetically in either ascending or descending order.

Lab Exercises

Lab 2-1 Working with basic queries

2.

```
SELECT *
FROM Students;
```

3. Eleven rows should be returned. You may see more or fewer rows if you added or deleted data outside of the labs.

4.

```
SELECT FirstName, LastName, City
FROM Students;
```

5.

```
SELECT DISTINCT City
FROM Students;
```

6. No. The Distinct keyword automatically sorts all data in ascending order. However, if you want the data sorted in descending order, you have to use an ORDER BY clause.

Lab 2-2a Using expressions in SQL queries: the concatenation operator

2.

```
SELECT LastName || ', ' || FirstName, City
FROM Students;
```

3. The new column has the same name as the expression (LastName || ', ' || FirstName).

4.
```
SELECT LastName || ', ' || FirstName AS "Student Name", City
AS "Home City"
FROM Students
```

5.
```
SELECT LastName || ', ' || FirstName AS "Student Name", City
AS "Home City" "
FROM Students
ORDER BY "Student Name"
```

Lab 2-2b Using expressions in SQL queries: the arithmetic operator

1.
```
SELECT InstructorID, LastName, PerDiemCost * 5 AS "weekly
cost"
FROM Instructors;
```

2.
```
SELECT InstructorID, LastName,
PerDiemCost * 5 AS "weekly cost",
(PerDiemCost + PerDiemExpenses) * 5 AS "travel cost"
FROM Instructors
```

3.
```
SELECT InstructorID, LastName,
PerDiemCost * 5 AS "weekly cost",
(PerDiemCost + PerDiemExpenses) * 5 AS "travel cost"
FROM Instructors
ORDER BY "travel cost" DESC
```

Lab 2-3 Limiting the rows returned with a WHERE clause

2.
```
SELECT InstructorID, PerDiemCost
FROM Instructors
WHERE PerDiemCost > 400 AND PerDiemCost < 600;
```

Note: If you use the condition "WHERE PerDiemCost BETWEEN 400 AND 600", your answer is incorrect because this condition includes those who make exactly $400 and exactly $600. The question asks for those who make more than $400 and less than $600.

3.

```
SELECT InstructorID, City
FROM Instructors
WHERE city IN ('Toronto', 'New York');
```

4.

```
SELECT InstructorID, PerDiemCost, City
FROM Instructors
WHERE PerDiemCost > 400 AND PerDiemCost < 600
AND city IN ('Toronto', 'New York');
```

5.

```
SELECT InstructorID
WHERE PerDiemExpenses IS NOT NULL;
```

Lab 2-4 Using the ROWNUM pseudo-column

2.

```
SELECT FirstName, LastName, PerDiemCost, ROWNUM
 FROM Instructors;
```

3.

```
SELECT FirstName, LastName, PerDiemCost, ROWNUM
FROM Instructors
WHERE ROWNUM <=4;
```

4.

```
SELECT FirstName, LastName, PerDiemCost, ROWNUM
FROM Instructors
ORDER BY PerDiemCost;
```

5. No. The ORDER BY statement is processed after the ROWNUM values have been assigned and changes the order in which the rows are presented.

CHAPTER 3

Using Single- and Multi-Row Functions

EXAM OBJECTIVES

- Single-Row Functions
  - Describe various types of functions available in SQL
  - Use character, number, and date functions in SELECT statements
  - Describe the use of conversion functions
- Aggregating Data Using Group Functions
  - Identify the available group functions
  - Describe the use of group functions
  - Group data using the GROUP BY clause
  - Include or exclude grouped rows by using the HAVING clause

CHAPTER PRE-TEST

1. What is the primary difference between single-row and multi-row functions?
2. When do you use a case-conversion function in a WHERE clause?
3. How do you format a date to include an ordinal number (for example, January 12th)?
4. What data types are accepted by the MAX() function?
5. What is the difference between COUNT(column) and COUNT(*)?
6. What function do you use to return the current date?
7. Can you write a query that calculates a value without drawing data from a specific table?
8. Do you have to include all nonaggregate values in the SELECT list in the GROUP BY clause?
9. Can you use group functions in the WHERE clause?
10. How do you write a query that returns a different value depending on the value of another column?

✦ Answers to these questions can be found at the end of the chapter. ✦

This chapter discusses Oracle's built-in functions. It examines both group and single-row functions. A function is a named programming element that takes one or more input values and returns an output value that has been generated by the code that defines the function. Oracle includes a number of functions. These functions can be used to modify, or generate values in the SELECT list of a SQL query. For the exam, you are expected to have a full understanding of the various functions in Oracle and how they are used. This chapter starts by looking at single-row functions. A single-row function returns one value for each value passed into it. Next, this chapter examines the use of group functions to generate aggregate data. It also examines how the GROUP BY and HAVING clauses can be used with group functions to generate subtotal data.

Having a solid understanding of how Oracle's built-in queries work enables you to greatly increase the power of the SQL language. Without these functions, you are limited to returning the data as it appears in the tables. However, the information you require often is not in the table but *derived from* data in the table. To derive this data you can use Oracle's functions. These functions, for example, enable you to format your output more effectively and to derive values from your tables that would be otherwise unavailable. Group functions also enable you to aggregate or summarize data in your tables to return values based on the data in your tables. This chapter starts by looking at single-row functions and then proceeds to an examination of group functions.

Single-Row Functions

- ✦ Describe various types of functions available in SQL
- ✦ Use character, number, and date functions in SELECT statements
- ✦ Describe the use of conversion functions

All single-row functions return one output value for each input value. For example, if you place a single-row function in a SELECT statement that returns ten rows, you receive one output value for each of the ten rows (that is, ten output values). Single-row functions are used to manipulate table values or expressions. They can work with different datatypes and can even be used to change the datatype of value. These functions can be broken down into four general categories: conversion functions, character functions, number functions, and date functions.

Conversion functions

As you saw in Chapter 2, Oracle is, in some cases, capable of implicitly converting data from one datatype to another. For example, if you concatenate a character column and a numeric column using the concatenation operator `||`, Oracle implicitly

converts the data in the numeric column into character data. However, in some situations, Oracle is unable to perform these conversions unassisted. For these situations, you need to use a conversion function. Oracle includes three conversion functions: TO_NUMBER, TO_CHAR, and TO_DATE.

When writing SQL statements, it is often better to explicitly convert values using a conversion function rather than rely on implicit conversion. This makes the code more readable and also avoids unexpected errors that result when Oracle is, for whatever reason, unable to make an implicit conversion.

The TO_NUMBER function

The TO_NUMBER function is used to convert character data, or strings, into numeric data. The syntax for this function is:

```
TO_NUMBER (string, [format])
```

This function is particularly useful when data has been stored in a character format to allow for thousand separators and decimal points. Consider the following example. A table called Sales_records contains two columns with a variable character datatype (varchar2(10)) called Asking_price and Sale_price. If the data in these columns looks like this:

```
Asking_price  Sale_price
------------  ------------
2000          1000
3000          1500
```

Oracle can execute the following query using implicit conversion:

```
SELECT Asking_price - Sale_price AS "Price Difference"
FROM Sales_records;

Price Difference
------------------
1000
1500
```

In this example, Oracle is able to implicitly convert the varchar2 values into numeric values and perform the arithmetic expression.

However, if the data in these columns looks like this:

```
Asking_price  Sale_price
------------  ------------
2,000.00      1,000.00
3,000.00      1,500.00
```

The previous query returns the following error:

```
ERROR at line 1:
ORA-01722: invalid number
```

You get this error because non-numeric characters in the data prevent Oracle from performing the implicit conversion. To avoid this error you can use the TO_NUMBER function. It would look like this:

```
SELECT TO_NUMBER(Asking_price, '9,9999.99') -
TO_NUMBER(Sale_price,  '99,999.99') AS "Price Difference"
FROM Sales_records;
```

This query works because the TO_NUMBER returns a numeric value. The character 9 in the formatting code represents any numeric value. By including the thousands separator and the decimal point, you are instructing the function on how it is to interpret the value passed into the function.

The last example in this section does not work properly when the column has a fixed-length character datatype. The reason is that the column contains padding characters to pad the value up to the width of the column. The TO_NUMBER function is unable to deal with these padding characters. This can be dealt with by using the TRIM function inside the TO_NUMBER function. If we consider the previous example, it looks like this:

```
SELECT TO_NUMBER(TRIM(Asking_price), '9,9999.99') -
TO_NUMBER(TRIM(Sale_price),  '99,999.99') AS "Price
Difference"
FROM Sales_records;
```

Using the TO_CHAR function

The TO_CHAR function is used to convert either date or numeric data to character data. It is most often used to format the appearance of date and numeric data using non-numeric characters to improve readability. In both cases, the syntax of the function is as follows:

```
TO_CHAR({numeric data | date data}, ['format'])
```

The format characters depend on what type of conversion you are attempting.

Using TO_CHAR with numeric data

In several situations, you can use the TO_CHAR function to convert numeric into character data. You might want to include a thousands separator or a decimal point (if the value is not stored in the table with decimal values). Also, if the numeric data represents currency, you might also want to include a currency symbol.

To indicate how you want the data formatted, you must include format elements with the function. Some of the more common format elements are listed in Table 3-1.

NLS currency settings

If you are working in a multilingual environment, you should use the "L" rather than the "$" as the formatting character to represent the currency symbol. The $ character always displays a dollar sign; however, when you use "L", it displays the currency symbol specified in the NLS_currency parameter.

For example, if your database's NLS language is set to American, the following query returns a value with the American currency symbol:

```
SELECT InstructorID,
  TO_CHAR(PerDiemCost * 5, 'L99,999.99') AS "Weekly Cost"
FROM Instructors;

INSTRUCTORID Weekly Cost
------------ -----------
         300   $2,500.00
         310   $2,250.00
         100   $3,000.00
         110   $2,500.00
         200   $3,750.00
         210   $2,000.00
         410   $2,000.00
         450   $2,250.00
```

In this case, the NLS_currency parameter value is the dollar sign. If you change the NLS_Territory value to Italy, you receive a different out put from the same query:

```
SELECT InstructorID,
  TO_CHAR(PerDiemCost * 5, 'L99,999.99') AS "Weekly Cost"
FROM Instructors;

INSTRUCTORID Weekly Cost
------------ --------------------
         300           L.2,500.00
         310           L.2,250.00
         100           L.3,000.00
         110           L.2,500.00
         200           L.3,750.00
         210           L.2,000.00
         410           L.2,000.00
         450           L.2,250.00
```

In this case, the NLS_currency value has changed from the dollar sign to the lira symbol (L.). Using the floating local currency symbol makes your scripts portable among language settings.

If you are not sure what your current NLS_currency value is, you can find it using the following query:

```
SELECT value FROM V$NLS_parameters
WHERE Parameter = 'NLS_CURRENCY';
```

Table 3-1
Number Format Elements with the TO_CHAR Function

Symbol	*Significance*
0	Displays a leading zero.
9	Represents any number. The number of nines determines width of output but does not display leading zeros.
,	Thousand separator.
.	Decimal point.
L	Floating currency symbol.
$	Floating dollar sign.
MI	Places minus sign to right of value if the value is negative.

These formatting elements describe how values are to be displayed. For example, suppose you want to list the weekly cost for each instructor and you want to include a dollar sign, you can use the following query:

```
SELECT FirstName, LastName,
  TO_CHAR(PerDiemCost * 5, '$99,999.99') AS "Weekly Cost"
FROM Instructors;
```

which produces the following output:

```
FIRSTNAME       LASTNAME             Weekly Cost
--------------- -------------------- -----------
Michael         Harrison             $2,500.00
Susan           Keele                $2,250.00
David           Ungar                $3,000.00
Kyle            Jamieson             $2,500.00
Lisa            Cross                $3,750.00
Geoff           Williams             $2,000.00
Lana            Chiu                 $2,000.00
Adele           LaPoint              $2,250.00
```

Note that in this example, the dollar sign is floating — that is, it appears next to the first numeral even though the format shows a five-digit number. If, however, you do not include enough number symbols, Oracle is unable to return any output. Consider the following example:

```
SELECT InstructorID,
  TO_CHAR(PerDiemCost * 5, '$999.99') AS "Weekly Cost"
FROM Instructors;

INSTRUCTORID Weekly C
------------ --------
```

```
300 ########
310 ########
100 ########
110 ########
200 ########
210 ########
410 ########
450 ########
```

In this example, the data returned exceeds the width of the formatting, and Oracle is unable to display the result set.

Show an example using zeros (0) in the format model, e.g. '$000.99', & perhaps tell why this might be used over just 9s.

Using the TO_CHAR function with date value

Just as with currency, Oracle looks to the NLS settings when it displays date data. When Oracle stores data using the date datatype, it stores the date in an internal numeric format. It must convert this numeric value to character data before it returns the value. When it makes this conversion, it uses the NLS_date_format value to determine how the date information will be presented. If you want a date format other than the NLS format, you must use the TO_CHAR function. TO_CHAR includes a number of format elements that are specific to date. A partial list of these format elements is provided in Table 3-2.

Table 3-2
TO_CHAR Date Format Elements

Symbol	*Significance*
DD	Two-digit day value
MM	Two-digit month value
Mon	Abbreviated month value
Month	The full month spelled out
YYYY	Four-digit year value
Year	The full year spelled out
Day	The day of the week
DY	Three-letter abbreviation for the day of the week
Q	The quarter

These formatting elements are used the same way as the number format elements. In addition to these elements, you can include spelling, punctuation, and literal values (enclosed in double quotes).

For example, the following query displays the start date for all classes, including the day of the week and the month spelled out:

```
SELECT ClassID, TO_CHAR(StartDate, 'DAY MONTH DD, YYYY')
    AS "Start Date"
FROM ScheduledClasses;

  CLASSID Start Date
--------- ----------------------------
       50 SATURDAY  JANUARY   06, 2001
       51 SATURDAY  JANUARY   13, 2001
       53 WEDNESDAY FEBRUARY  14, 2001
```

You should note a few points about this result set. First, the values for DAY and MONTH are all uppercase. This is because Oracle sets the capitalization of these values in the same manner as you list them in the format elements. If you had submitted the format values "DAY" and "MONTH" as "Day" and "Month", the output also would be in initial capital letters:

```
 SELECT ClassID, TO_CHAR(StartDate, 'Day Month DD, YYYY')
    AS "Start Date"
FROM ScheduledClasses;

  CLASSID Start Date
--------- ----------------------------
       50 Saturday  January   06, 2001
       51 Saturday  January   13, 2001
       53 Wednesday February  14, 2001
```

The second point to note is that the punctuation in the format elements is included in the result set. Finally, notice that spaces are embedded between the various date elements. These embedded spaces can be removed by placing the "fm" prefix before the elements in the format elements. If we include this prefix in the previous query, we get a different result:

```
SELECT ClassID, TO_CHAR(StartDate, 'fmDay Month fmDD, YYYY')
    AS "Start Date"
FROM ScheduledClasses

  CLASSID Start Date
--------- ----------------------------
       50 Saturday January 06, 2001
       51 Saturday January 13, 2001
       53 Wednesday February 14, 2001
```

In this query, the result set does not contain the same embedded spaces that were in the previous query.

You can include two suffixes in the format elements. The suffix "sp" is used to spell out a numeric value (for example, the format element DDsp spells out the day rather than returning a numeral). You can also include the "th" suffix to add an ordinal ending to a numeric format element. Consider the following query:

```
SELECT ClassID,
TO_CHAR(StartDate, 'fmDAY "the" DDth "of"  Month DD, YYYYsp')
    AS "Start Date"
FROM ScheduledClasses

CLASSID Start Date
------- ------------------------------------------------------
      50 SATURDAY the 6TH of  January 06, TWO THOUSAND ONE
      51 SATURDAY the 13TH of  January 13, TWO THOUSAND ONE
      53 WEDNESDAY the 14TH of  February 14, TWO THOUSAND ONE
```

The TO_DATE function

The TO_DATE function is used to help Oracle convert string values into its internal date format. It uses the same formatting values that the TO_CHAR function uses with dates. The syntax for this function is:

```
TO_DATE(string, 'format')
```

When Oracle encounters a character string that it needs to treat as a date, it uses the NLS_date_format value to attempt to interpret the string. If the date string does not conform to this format, Oracle returns an error. Consider the following query:

```
SELECT ClassID
FROM ScheduledClasses
WHERE StartDate = 'January 6, 2001';

WHERE StartDate = 'January 6, 2001'
                  *
ERROR at line 3:
ORA-01858: a non-numeric character was found where a numeric
was expected
```

The reason Oracle was expecting a numeric value is the NLS_date_format for this server was not changed from the default "DD-MON-YY". It was looking for a numeric value for the day of the month and, instead, encountered a character value. In order to make this query work, you have to use the TO_DATE function to show Oracle how to interpret the string:

```
SELECT ClassID
FROM ScheduledClasses
WHERE StartDate = TO_DATE('January 6, 2001', 'Month DD, YYYY');

  CLASSID
---------
       50
```

The DUAL Table

All SQL queries contain two mandatory clauses: the SELECT list and the FROM clause. At times, however, you may want to issue a query that derives all of its data expressions within the query which do not rely on any table. If you were to issue a query without a FROM clause, for example:

```
SELECT UPPER('Hello There') AS UPPER,
       LOWER('HOW ARE') AS LOWER,
       INITCAP('YOU TODAY') AS INITCAP
```

you would receive the following error:

```
ERROR at line 3:
ORA-00923: FROM keyword not found where expected
```

You receive this message because Oracle cannot process a query without a FROM clause. To solve this problem, Oracle includes the DUAL table.

The DUAL table is a very small public table. When you use the SQL*Plus DESC command to view this table, it appears as follows:

```
SQL> DESC dual;
 Name                           Null?    Type
 ------------------------------ -------- -----------------
 DUMMY                                   VARCHAR2(1)
```

The Dummy column contains a single row with a value of X. In the example above, because you are not referencing any column, you could have put any table in your schema in the FROM clause. However, this query would evaluate one for each row in the table. Because the DUAL table only contains one row, the select expression is only evaluated once, and the lookup on the table is much faster.

Character functions

Character functions are used to manipulate case and appearance of character data. Character functions can be divided into two general categories: case-conversion functions and character-manipulation functions.

Case-conversion functions

Case-conversion functions are used to alter the case of character data in the result set. The three case-conversion functions are described in Table 3-3:

Table 3-3
Case-Conversion Functions

Function	*Description*
LOWER	Returns all alphabetic characters in lowercase.
UPPER	Returns all alphabetic characters in uppercase.
INITCAP	Capitalizes the first letter of each word and sets the remaining characters in lowercase.

You can see the results of all three functions in the following query:

```
SELECT UPPER('Hello There') AS UPPER,
       LOWER('HOW ARE') AS LOWER,
       INITCAP('YOU TODAY') AS INITCAP
FROM Dual

UPPER         LOWER     INITCAP
-----------   -------   ---------
HELLO THERE   how are   You Today
```

Case-conversion functions are also quite useful in the WHERE clause. If you do not know the exact case of a character value as it is stored in the table, it is very difficult to retrieve data using that string in a WHERE clause. For example, consider the outcome of the following query:

```
SELECT InstructorId, City
FROM Instructors
WHERE City = 'TORONTO';

no rows selected
```

There are instructors in this table from Toronto; however, in the table, the string is stored in initial capital letters. Because the character strings in the WHERE clause are case sensitive, values must match case as well as spelling. You can avoid this problem by using one of the case-conversion functions:

```
SELECT InstructorId, City
FROM Instructors
WHERE UPPER(City) = 'TORONTO';

INSTRUCTORID CITY
------------ --------------------------
         300 Toronto
         310 Toronto
         210 Toronto
```

In this query, Oracle takes each value in the City column, forces it into uppercase, and then compares it to the string. This allows the query to return the three Toronto instructors even though the city value is not stored in all capital letters.

Character-manipulation functions

Character-manipulation functions are used to modify the appearance of character data or to manipulate the strings to provide information about them. Several character-manipulation functions are listed in Table 3-4.

Table 3-4
Character-Manipulation Functions

Function	*Description*
SUBSTR	Returns a portion of a string.
CONCAT	Combines two string values.
LPAD	Pads a string with a defined character to a defined length.
TRIM	Trims leading or ending characters.
LENGTH	Returns the length of a string value.
INSTR	Returns the position of a particular character within a string.

SUBSTR

The SUBSTR function is used to return a portion of a string value. The portion that the function returns is determined with parameters in the function itself. The syntax for the function is:

```
SUBSTR(string, <start position>, <number of characters>)
```

Both the start position and number of characters are numeric values indicating the position of the characters within the string that you want returned. For example, if you want to return the first initial and last name of all instructors, you can use the following query:

```
SELECT SUBSTR(FirstName, 1, 1) , Lastname
FROM Instructors;

S LASTNAME
- --------------------------
M Harrison
S Keele
D Ungar
```

```
K Jamieson
L Cross
G Williams
L Chiu
A LaPoint

8 rows selected.
```

In this case, the SUBSTR function instructs Oracle to return a portion of FirstName, starting at the first character and returning only one character. The output of this function also includes any spaces included in the string. Consider the output of the following query:

```
SELECT SUBSTR(CourseName, 5, 10) AS SUBSTRING
FROM Courses

SUBSTRING
----------
c SQL
nced SQL
ormance Tu
base Perfo
base Admin
ing up you
c PL/SQL
nced PL/SQ
g your PL/
```

In the preceding example, spaces are maintained. If the number of characters required is greater than the number of characters in the string, Oracle simply leaves the remainder blank (as is the case with the first row "Basic SQL" which has only nine characters total).

CONCAT

The CONCAT function works much like the concatenation operators (`||`) that you saw in Chapter 2. The function takes any two string values and combines them into a single value. The syntax for this function is:

```
CONCAT(string1,string2)
```

For example:

```
SELECT CONCAT('Oracle', 'Server') AS Concat
FROM Dual;

CONCAT
------------
OracleServer
```

With the CONCAT function, you are limited to joining only two values. This might seem less effective than using the concatenation operators; however, the advantage of this function is that it can be nested in other functions (whereas the concatenation operator cannot be used within all SQL function).

LPAD

The LPAD function is used to place characters to the left of a string whose size will always be as specified in the parameters passed to the function. This function is used to format output and make it uniform. The syntax for this function is:

```
LPAD(char, size of string, 'padding character')
```

For example:

```
SELECT LPAD(LastName, 10, '*') AS LPAD
FROM Students;

LPAD
----------
*****Smith
*****Jones
****Massey
*****Smith
*****Hogan
*******Hee
****Andrew
***Holland
*****Jones
****Colter
*Patterson
```

In this example, the result set returns ten characters per row. The value in each row returns a string with the last name and pads the values with the padding character up to ten characters in length.

There is also an RPAD function that pads right-justified. Its syntax is the same as the LPAD function. If you do not supply a padding character of the LPAD and RPAD functions, they will pad with a blank space by default.

TRIM

The TRIM function removes leading and trailing characters from a string. The syntax for this function is:

```
TRIM([{LEADING | TRAILING | BOTH}]char FROM string)
```

If you do not specify a character to be trimmed, Oracle trims blank spaces by default. If you do not specify leading or trailing, the default behavior is to trim both leading and trailing values. For example, the following query removes the character "S" from a string:

```
SELECT TRIM( 'S' FROM 'STEVENS') AS TRIM
FROM Dual;

TRIM
------
TEVEN
```

If you want only one trimmed, you must specify which string you want trimmed. For example, the query:

```
SELECT TRIM(TRAILING 'S' FROM 'STEVENS') AS TRIM
FROM Dual;
```

returns:

```
TRIM
-------
STEVEN
```

An additional point you must remember with this function is that it is case sensitive. The characters you choose to trim must match the case of the characters in the string. Consider the following example:

```
SELECT TRIM(BOTH 'S' FROM 'Stevens')AS TRIM
FROM Dual

TRIM
------
tevens
```

Because the trailing "s" is lowercase, it is ignored by the TRIM function.

Finally, you cannot include multiple characters in the list of characters to be trimmed. For example, the following query returns an error

```
SELECT TRIM('ST' FROM 'STEVENS') AS TRIM
FROM Dual;

ERROR at line 1:
ORA-30001: trim set should have only one character
```

The TRIM function is new to Oracle 8*i* and can be used in place of both the LTRIM and RTRIM functions (although these still work and can be used). The LTRIM function removes leading characters, whereas the RTRIM function removes trailing characters.

LENGTH

The LENGTH function is used to return the length of a string value. It returns it as a numeric value.

For example:

```
SELECT Firstname, LENGTH(Firstname) AS LENGTH
FROM Instructors;

FIRSTNAME                      LENGTH
------------------------------ ---------
Michael                             7
Susan                               5
David                               5
Kyle                                4
Lisa                                4
Geoff                               5
Lana                                4
Adele                               5
```

INSTR

The INSTR function returns the position of first occurrence of a particular character within a string. Like the LENGTH function, it returns a numeric value. If it does not find the character value, it returns a 0. For example:

```
SELECT Firstname, INSTR(Firstname, 'a') AS INSTR
FROM Instructors

FIRSTNAME                       INSTR
------------------------------ ---------
Michael                             5
Susan                               4
David                               2
Kyle                                0
Lisa                                4
Geoff                               0
Lana                                2
Adele                               0
```

Note that the string value in the function is case sensitive. Adele shows an INSTR value of 0 because the function is looking only for the lowercase “a” character.

Number functions

Number functions enable you to manipulate numeric values. There are many number functions, but you are responsible for knowing the following functions: ROUND, TRUNC, and MOD.

ROUND

The ROUND function is used to round numeric values to a decimal place specified as a parameter of the functions. The syntax for this function is:

```
ROUND(number, n)
```

where n is the number of decimal places to which you want the value rounded. If n is a negative number, the ROUND function rounds to the left of the decimal place. If n is zero, Oracle rounds to the nearest whole number. If you do not supply a value for n, Oracle rounds the numeric value to the nearest whole number.

Consider the following example:

```
SELECT ROUND(44.647, 2) AS POSITIVE,
       ROUND(44.647, 0) AS ZERO,
       ROUND(44.647, -1) AS NEGATIVE
FROM Dual;

POSITIVE      ZERO  NEGATIVE
--------- --------- ---------
    44.65        45        40
```

In this example, the value in the POSITIVE column has been rounded to the nearest hundred (two decimal places), the value of ZERO has been rounded to the nearest whole number, and the value for NEGATIVE has been rounded to the nearest ten.

TRUNC

The TRUNC function works much like the ROUND function except that rather than rounding values, it simply truncates the value at the specified decimal place. The syntax for this function is:

```
ROUND(number, n)
```

Just as with the ROUND function, if n is a positive number, the function truncates values to the right of the decimal place; if the value is 0, it truncates to the nearest whole number; and if it is negative, it truncates to the left of the decimal point. Consider the following example:

```
SELECT TRUNC(44.647, 2) AS POSITIVE,
       TRUNC(44.647, 0) AS ZERO,
       TRUNC(44.647, -1) AS NEGATIVE
FROM Dual;

POSITIVE      ZERO  NEGATIVE
--------- --------- ---------
    44.64        44        40
```

As you can see from this example, there is no rounding of the values; instead, anything outside the scope of the truncation is ignored.

MOD

The MOD function returns the remainder of two values. The syntax for this function is:

```
MOD(value1, value2)
```

When this function is called, it divides value1 by value2 and returns only the remainder. If there is no remainder, the function returns a zero.

Consider the following example:

```
SELECT MOD(2200, 300) AS ODD, MOD(2100, 300) AS EVEN
FROM Dual

      ODD       EVEN
--------- ----------
      100          0
```

In the ODD column, 300 goes into 2,200 three times with a remainder of 100. In the EVEN column, 2,100 can be evenly divided by 300, leaving no remainder. In Oracle, when the second value is a 0, the MOD function returns the first value (which deviates a bit from the rules of mathematics). For example, the query:

```
SELECT MOD(1000, 0)AS ZERO_DIVIDE
FROM Dual;
```

should return a mathematical error (because values cannot be mathematically divided by zero). However, in Oracle this function returns the following:

```
ZERO_DIVIDE
-----------
       1000
```

When either element passed into the MOD function is NULL, the result is NULL.

Date functions

Oracle includes a number of functions that enable you to modify and manipulate date data. Most of these functions enable you to perform calculations on dates. As mentioned in Chapter 2, when you add a number to or subtract a number from a date, the result is the date that many days forward or backward in time. However, what happens if you want to add two months? Clearly you cannot add 60 days, because not all months are 30 days in length. It is these kinds of problems that the majority of the date functions were designed to deal with. The date functions are listed in Table 3-5.

Table 3-5
Oracle Date Functions

Function	*Description*
ADD_MONTHS	Adds a number of months to date value and returns a new date.
MONTHS_BETWEEN	Determines the number of months between two dates.
NEXT_DAY	Determines the next occurrence of a particular day of the week.
LAST_DAY	Determines the last day of the month.
ROUND	Rounds to the nearest date part.
TRUNC	Truncates to the nearest date part.

ADD_MONTHS

The ADD_MONTHS function is used to add or subtract a number of months to or from a date value. The syntax for this function is:

```
ADD_MONTHS(date, number)
```

For example, if you want to schedule a course six months after the current start date for the courses in the ScheduledClasses table and need to know the date, you can use the following query:

```
SELECT StartDate, ADD_MONTHS(StartDate, 6) AS 6_MONTHS
FROM ScheduledClasses;

STARTDATE 6_MONTHS
--------- ---------
06-JAN-01 06-JUL-01
13-JAN-01 13-JUL-01
14-FEB-01 14-AUG-01
```

Internally, the function is able to calculate the number of days that are in the six-month span defined by the StartDate value and add that many days to the start date value.

MONTHS_BETWEEN

The MONTHS_BETWEEN function also enables you to derive values that cannot easily be calculated mathematically. Remember that when you subtract two dates, the result is the number of days between the two dates. If you need the number of months, you need to use this function. The syntax for the function is:

```
MONTHS_BETWEEN(date1, date2)
```

For example:

```
SELECT MONTHS_BETWEEN('01-JAN-01', '01-AUG-01') AS
MONTHS_BETWEEN
FROM Dual;

MONTHS_BETWEEN
---------------
             -7
```

The result of this function returns a negative value because in this example, you are subtracting a higher value from a lower number (because numeric values in Oracle's internal date format increase with time). If you put the later date first, the result is a positive number.

NEXT_DAY

The NEXT_DAY function returns the date for the next day of the week requested in the parameters of the function. The syntax for this function is:

```
NEXY_DAY(date, day)
```

For example, to find the date of the Friday after each course start date, you use the following query:

```
SELECT StartDate, NEXT_DAY(StartDate, 'FRIDAY') AS NEXT_DAY
FROM ScheduledClasses;


STARTDATE NEXT_DAY
--------- ---------
06-JAN-01 12-JAN-01
13-JAN-01 19-JAN-01
14-FEB-01 16-FEB-01
```

The advantage of using this function is that you do not have to know which day of the week the initial date falls on. In this example, the first course actually starts on a Friday, whereas the last course starts on a Wednesday.

LAST_DAY

The LAST_DAY function returns the last day of the month that contains the date specified in the function's parameters. The syntax for this function is:

```
LAST_DAY(date)
```

For example, suppose you pay your instructors on the last day of each month. You can calculate the payment date with the following query:

```
SELECT ClassID, StartDate, LAST_DAY(StartDate) AS Payment
FROM ScheduledClasses;

  CLASSID STARTDATE PAYMENT
--------- --------- ---------
       50 06-JAN-01 31-JAN-01
       51 13-JAN-01 31-JAN-01
       53 14-FEB-01 28-FEB-01
```

Oracle determines the month in which each date value exists and then determines the last day for that month.

ROUND and TRUNC

You can also use the ROUND and TRUNC functions with date values. In this case, you specify the part of the date you want to round or truncate values on rather than a decimal position. For example, if you want to round a date to a particular month or year, you can use the following query:

```
  SELECT ROUND(StartDate, 'MONTH') AS MONTH,
         ROUND(StartDate, 'YEAR') AS YEAR
  FROM ScheduledClasses

MONTH     YEAR
--------- ---------
01-JAN-01 01-JAN-01
01-JAN-01 01-JAN-01
01-FEB-01 01-JAN-01
```

Additional functions

Oracle also contains a few single-row functions that do not fit into any of the above-mentioned categories. Three main functions in this category are: NVL, SYSDATE, and DECODE.

Use the NVL function

The NVL function is used to provide an actual value in place of a NULL returned by a query. It can be used with any date, number, or character datatype. This value can then be treated as a literal value in the query. This function evaluates a specified column in each row returned. The syntax for this function is:

```
NVL(column, value)
```

As each row is passed into the function, it checks to see if the field for the specified column is NULL. If it is not, the NVL function simply forwards the value; however, if

it is NULL, the NVL function returns whatever value you specified in the function. For example, if you want to calculate the total weekly cost for each instructor, you can use a query like the following:

```
SELECT InstructorID,
       (PerDiemCost + PerDiemExpenses) * 5 AS "Weekly Cost"
FROM Instructors;

INSTRUCTORID Weekly Cost
------------ -----------
         300        3500
         310        3250
         100        4000
         110        3500
         200        5000
         210
         410
         450        3250
```

You remember from Chapter 2 that any arithmetic expression containing a NULL in one of its values always returns NULL. In this case, because instructors 210 and 410 have a NULL value in the PerDiemExpenses column, their total weekly cost also is NULL. To return a value for these two instructors, you must use the NVL function:

```
SELECT InstructorID,
       (PerDiemCost + NVL(PerDiemExpenses, 0)) * 5 AS "Weekly
Cost"
FROM Instructors;

INSTRUCTORID Weekly Cost
------------ -----------
         300        3500
         310        3250
         100        4000
         110        3500
         200        5000
         210        2000
         410        2000
         450        3250
```

In this example, the NVL function checks the PerDiemExpenses field for each row. If it is not NULL, the function returns whatever value was passed into it. If the value is NULL, the function returns a 0. Unlike the NULL, the zero is a literal value and allows the arithmetic expression to be evaluated correctly.

The value you include in the NVL function must match the datatype of the column being evaluated. For example, the following query returns an error:

```
SELECT InstructorID, NVL(PerDiemExpenses, 'No expenses')
FROM Instructors;
```

```
SELECT InstructorID, NVL(PerDiemExpenses, 'No expenses')
                                                  *
ERROR at line 1:
ORA-01722: invalid number
```

In this case, Oracle is being asked to place character date in a numeric column, which violates the datatype.

If you want to supply an NVL default that is of a different datatype, you must also include a conversion function in the query to convert the datatype of the column in the result set. The following query does not return an error:

```
SELECT InstructorID, NVL(TO_CHAR(PerDiemExpenses)
       'No  Expenses')
FROM Instructors
```

Using the SYSDATE function

The SYSDATE function is used to return the current date from the system. When this function is called, it consults the internal clock for the server and returns the current date. This function often is used in scripts that are temporal in nature. For example, if you want to write a query that returns a list of all courses that have already passed, you can use the following SQL statement:

```
SELECT CourseNumber, Startdate
FROM ScheduledClasses
WHERE Startdate < SYSDATE;
```

Clearly, in this statement, you do not want to hard-code the date in the condition because that fixes the result set. The SYSDATE value is recalculated every time the statement is run. Therefore, the query always returns courses that started previous to the current day.

Using the DECODE function

The DECODE function enables you to build some conditional logic into your queries. It functions much like an If-Then-Else statement. The syntax for this function is:

```
DECODE(column/expression, conditon1, result1
                         [,condition2, result2]
                         [, default])
```

The use of this function is best demonstrated in this scenario. You want to return each instructor and total daily cost. You also want to include a rank value based on the instructors' cost. You want to order the list by this rank value, with the highest-ranking instructors at the top of the list. You can do so with the following query:

```
SELECT InstructorID, PerDiemCost,
       DECODE(ROUND(PerDiemCost/100, 0),
                          4, 1,
                          5, 2,
                          6, 3,
                          7, 4,
                             5) AS Rank
FROM Instructors
ORDER BY Rank DESC;

INSTRUCTORID PERDIEMCOST      RANK
------------ ----------- ---------
         200         750         5
         100         600         3
         300         500         2
         110         500         2
         450         450         2
         310         450         2
         210         400         1
         410         400         1
```

In this query, the PerDiemCost value for each row is divided by 100 and then rounded to the nearest whole number. That whole number is then compared against the first value in the list (in this case, 4). If it equals this value, the substitute value is returned (if the value is 4, the substitute value is 1). If it is not equal, it evaluates against the next value in the list. This continues until either the generated value finds a match or it reaches the end of the list. If it reaches the end of the list (that is, the value is greater than 7), the final substitute value (5) is returned.

Nesting functions

In SQL, single-row functions can be nested within each other. When functions are nested, Oracle evaluates the innermost function first and passes its value to the outer function. Only the outermost function actually returns a value back to the result set. Consider the following example. In the Instructors table, the InstructorID column is numeric, but suppose you want an InstructorID that is the first three letters of each instructor's last name concatenated to the first two initials of the instructor's first name. You can do so with the following query:

```
SELECT CONCAT(SUBSTR(LastName, 1, 3), SUBSTR(FirstName, 1, 2))
AS CONCAT_ID
FROM Instructors;

CONCA
-----
HarMi
KeeSu
UngDa
JamKy
```

```
CroLi
WilGe
ChiLa
LaPAd
```

If you want to eliminate the possibility of NULL values and pad the column so that the full column name is visible, you can add further functions to the query:

```
SELECT CONCAT(SUBSTR(NVL(LastName, 'xxx'), 1, 3),
SUBSTR(NVL(FirstName, 'yy'), 1, 2)) AS CONCAT_ID
FROM Instructors;

CONCA
-----
HarMi
KeeSu
UngDa
JamKy
CroLi
WilGe
ChiLa
LaPAd
```

In this example, the two NVL functions are evaluated first. If either the first name or the last name is NULL, the NVL value is returned, otherwise the column value is passed to the SUBSTR functions (which are evaluated second). Finally, the results of the two SUBSTR functions are used to evaluate the outermost function (CONCAT), and it is that function that actually returns a value to the result set.

Group/Aggregate Functions

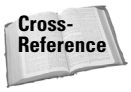
Group (sometimes referred to as *multi-row*) functions differ greatly from the single-row functions. They are used to derive aggregate values. All of these functions return one and only one value, regardless of how many values are passed into the function. The value that is returned by these functions is a result based on all of the values passed into the function. There are a number of group functions (the primary ones are listed in Table 3-6).

Table 3-6
Group Functions

Function	*Description*
AVG	Returns the average of all values inputted.
SUM	Returns the total of all values inputted.
COUNT	Counts the number of rows inputted.

Function	*Description*
MIN	Returns the lowest value inputted.
MAX	Returns the highest value inputted.

A SELECT list can contain one or more group functions; however, the SELECT list cannot contain both multi-row functions and nonaggregate column data unless it also contains a GROUP BY clause.



The GROUP BY clause is discussed later in this chapter in "Using the GROUP BY Clause."

AVG and SUM

The AVG and SUM functions accept only numeric datatype values. The AVG function returns the average of all NOT NULL values passed into the function, and the SUM function returns the total for all NOT NULL values. For example, if you want to know the average per diem cost for all instructors and the total of all per diem costs, you can use the following query:

```
SELECT AVG(PerDiemCost) AS AVERAGE, SUM(PerDiemCost) AS TOTAL
FROM Instructors;

  AVERAGE     TOTAL
--------- ---------
   506.25      4050
```

To determine the total value, Oracle adds the values from all rows passed into the function. To determine the average value, Oracle divides the sum by the number of columns that do not contain NULLs. This final point is important to remember and is discussed in detail at the end of this section.

COUNT(column) and COUNT(*)

The COUNT function is used to count the number of rows passed into the function. The difference between COUNT(column) and COUNT(*) is that COUNT(*)includes rows where there are NULL values, whereas COUNT(column) only counts those rows that are not NULL in the specified example. Consider the following example:

```
  SELECT COUNT(*) AS ALL_ROWS,
         COUNT(PerDiemExpenses) AS NOT_NULL
  FROM Instructors

  ALL_ROWS NOT_NULL
---------- ----------
         8          6
```

Eight rows are in the Instructors table, and the COUNT(*) function includes all of them; however, two instructors do not have a value in the PerDiemExpenses column. This is why the value in this column is two less than the COUNT(*) value. Each of these functions can be used with columns of any data type.

MIN and MAX

The MIN and MAX functions return the lowest and highest values in a column. The MIN and MAX functions also ignore NULLs. For example, if you want to know who has the highest and lowest per diem travel cost, you can use the following query:

```
SELECT MAX(PerDiemExpenses) AS HIGHEST,
       MIN(PerDiemExpenses) AS LOWEST
FROM Instructors;

  HIGHEST    LOWEST
--------- ---------
      250       200
```

As with all group functions, only one value is returned. If multiple rows all have the highest or lowest values, still, only one row is returned.

These functions work with any datatypes except for BLOB, CLOB, RAW, LONG, and LONG RAW. When you use MIN and MAX with dates, it gives you the oldest and most recent dates. For example, if you want to see the first and last days when there were enrollments, you can use the following query:

```
SELECT MAX(EnrollmentDate) AS Latest,
       MIN(EnrollmentDate) AS Earliest
FROM ClassEnrollment;

LATEST    EARLIEST
--------- ---------
05-JAN-01 02-DEC-00
```

When you use MIN and MAX with character data, Oracle retrieves the lowest or highest value alphabetically (that is, closest to “a” and closest to “z”). Consider the following query:

```
SELECT MIN(LastName) AS "Closest to A",
       MAX(LastName) AS "Closest to Z"
FROM Instructors;

Closest to A                   Closest to Z
------------------------------ ------------------------------
Chiu                           Williams
```

How Oracle determines character values

Oracle determines MIN and MAX character values based on the ASCII values for each letter. For this reason, uppercase letters (ASCII characters 65–90) are considered to be lower than lowercase letters (ASCII characters 97–122). Therefore, if you have a single column table called test with the following values:

```
TEST_VAl
---------------
Banana
apple
orange
Watermelon
```

and you apply the MIN and MAX functions, they consider lowercase letters higher than uppercase letters. Consider the following query:

```
SELECT MAX(TEST_VAL) AS HIGHEST,
      MIN(TEST_VAL) AS Lowest
FROM Test;

HIGHEST         LOWEST
--------------- ---------------
orange          Banana
```

If you consider the whole list, apple is alphabetically lower than Banana; however, the ASCII value for "a" (97) is higher than the ASCII value for "B" (66). In the same fashion, "o" (ASCII value 112) is higher than "W" (ASCII value 87).

Group functions and NULLs

It is important to be aware of the fact that, with the exception of COUNT(*), all group functions ignore NULL values. Because of this behavior, it is important to be aware of the existence of NULL fields in your tables (particularly when the COUNT(exp) and AVG functions are involved) because this behavior may result in unexpected results.

Suppose a two-column table called Wages looks like this:

```
Employee    Hourly_Wage
---------  -----------
Bob        15
Sue        10
Jane
Rich       5
```

You execute the following query against this table and receive a result:

```
SELECT AVG(Hourly_Wage) as AVG_SAL
FROM Wages;

AVG_SAL
-------
10
```

To process this query, Oracle adds all of the values and divides by all of the non-NULL columns. However, is this answer, strictly speaking, correct? The answer to this question is "it depends." It depends on why Jane has a NULL Hourly_Wage. If she is a seasonal contractor who works only on holidays and you want to know only the average wage for your regular employees, 10 is the correct value. However, if you simply haven't determined Jane's salary yet, this value is accurate only if Jane ends up earning $10 per hour. To include all of the employees, you would have to use the NVL function to replace any NULLs (which will be ignored by the function) with a value that will be read by the function. For example, you could rewrite the previous query to return a value of zero for any nulls:

```
SELECT AVG(NVL(Hourly_Wage,0)) as AVG_SAL
FROM Wages;

  AVG_SAL
---------
      7.5
```

Using the DISTINCT keyword with NULL

If you want to consider only unique rows, it is possible to use the DISTINCT operator inside a group function. For example, to find the average of all unique per diem expense rates in the Instructors table, you use the following query:

```
SELECT AVG(DISTINCT PerDiemExpenses) AS WITH_DISTINCT,
       AVG(PerDiemExpenses) AS NO_DISTINCT
FROM Instructors;

WITH_DISTINCT NO_DISTINCT
------------- -----------
          225   208.33333
```

In this example, the values are different because the NO_DISTINCT column adds up all duplicate values and then divides by all values, whereas the WITH_DISTINCT column adds only one instance of each value and divides by the number of unique instances. The DISTINCT keyword can be used with all group functions in the same manner. Since group functions return only one value, there is no advantage to using the DISTINCT keyword in the SELECT list — for example:

```
SELECT DISTINCT AVG(PerDiemExpenses)
FROM Instructors;
```

Using the WHERE clause with group functions

In any SQL query, you can use a WHERE clause to limit the number of rows returned in the result set. When you use a WHERE clause in a query that contains group functions, you limit the number of rows that are considered by the query. For example, the following query returns the AVG per diem cost for all instructors:

```
SELECT AVG(PerDiemCost) AS "AVG COST"
FROM Instructors

AVG COST
---------
  506.25
```

However, if you want the per diem cost for only those instructors in Toronto, you can limit the group function with a WHERE clause:

```
SELECT AVG(PerDiemCost) AS "AVG COST"
FROM Instructors
WHERE City = 'Toronto'

 AVG COST
---------
      450
```

The result for this second query is different from the first because the WHERE clause is evaluated by Oracle before the group function is evaluated. Only those rows that match the WHERE clause criteria are included in the calculation of the group function.

Using the GROUP BY Clause

- ✦ Identify the available group functions
- ✦ Describe the use of group functions
- ✦ Group data using the GROUP BY clause
- ✦ Include or exclude grouped rows by using the HAVING clause

When you use a group function, it returns only one value that considers all applicable rows in the table. However, you may not want a single value that includes all rows. For example, rather than the average per diem cost for all trainers, you might want to know the average for each location. You might also want to list the location in the result set; however, if you execute the following query, it raises an error:

```
SELECT City, AVG(PerDiemCost) AS "Average Cost"
FROM Instructors;

SELECT City, AVG(PerDiemCost)
```

```
       *
ERROR at line 1:
ORA-00937: not a single-group group function
```

The source of the error in this query is that Oracle is not able to evaluate a group function result and the result set of a nonaggregate column in the same query. Essentially, this query is asking Oracle to return one value and multiple values in the same result set. Oracle, clearly, is unable to satisfy this query.

This understanding, however, does not help you find the average salary by city. You can determine each value individually by limiting the rows considered by the AVG function with a WHERE clause, but this is impractical when you have several hundred (or several thousand) values. To enable you to find an aggregate based on a nonaggregate value, SQL includes the GROUP BY clause.

With the GROUP BY clause, you can return both aggregate and nonaggregate data in the same query. Consider the previous example with the GROUP BY clause added:

```
SELECT City, AVG(PerDiemCost) AS "Average Cost"
FROM Instructors
GROUP BY City;

CITY                           Average Cost
------------------------------ ------------
New York                              487.5
Palo Alto                               750
Toronto                                 450
```

In this query, the GROUP BY clause groups all rows based on the column(s) specified in the clause (in this case, City) and then returns a single aggregate value for each unique grouping value. In this example, three different cities are listed in the City column, with multiple instances of each value. The AVG function is applied to all rows that contain a particular city value, and only one row is returned per city. You do not have to put the nonaggregate column in the SELECT list; however, the output is more readable when you know what each Average Cost is referring to.

If you place a nonaggregate column in the SELECT list, you must include it in the GROUP BY clause. For example, suppose you want to know what the highest per diem is in each city . You also want the query to include the ID of the instructor who makes this amount. When you execute the following query, you receive an error:

```
SELECT InstructorID, City, MAX(PerDiemCost) AS "Highest Cost"
FROM Instructors;
GROUP BY City;

SELECT InstructorID, City, MAX(PerDiemCost) AS "Highest Cost"
       *
ERROR at line 1:
ORA-00979: not a GROUP BY expression
```

In this query, you receive an error because Oracle doesn't know how to deal with the additional nonaggregate column. Again, Oracle is being asked to return one row per group and multiple rows in the same operation. In order to avoid this error, you have to include InstructorID in the GROUP BY clause, but this adds a new set of problems. Consider the output of the query when you include the InstructorID column in the GROUP BY clause:

```
SELECT InstructorID, City, MAX(PerDiemCost) AS "Highest Cost"
FROM Instructors
GROUP BY InstructorID, City;

INSTRUCTORID CITY                           Highest Cost
------------ ------------------------------ ------------
         100 New York                                600
         110 New York                                500
         200 Palo Alto                               750
         210 Toronto                                 400
         300 Toronto                                 500
         310 Toronto                                 450
         410 New York                                400
         450 New York                                450

8 rows selected.
```

In this example, the query returns all the rows in the Instructors table. In fact, if you execute the following query, you receive the same results:

```
SELECT InstructorID, City, PerDiemCost AS "Highest Cost"
FROM Instructors;
```

Why do we get this result? It is because SQL groups values on the unique occurrence of *all* columns listed in the GROUP BY clause. Because InstructorID is the primary key for the table, by definition, each row is unique. This means that only one row value is being considered by the group function for each row. In this case, you only wanted to group by City, however, because you included the InstructorID column in the SELECT list, you were forced to place it in the GROUP BY clause to avoid a syntax error. Therefore, you must include only those columns in the SELECT list that you intend to use as grouping criteria in the GROUP BY clause.

You can also use a WHERE clause with a GROUP BY clause. The WHERE clause limits which rows are considered by the grouping function. For example, returning to the earlier average cost by city example, if you want a report for only U.S. centers, you can add the following WHERE clause:

```
SELECT City, AVG(PerDiemCost) AS "Average Cost"
FROM Instructors
WHERE Country = 'USA'
GROUP BY City;
```

```
CITY                           Average Cost
------------------------------ ------------
New York                              487.5
Palo Alto                               750
```

When writing this query, you must always place the WHERE clause before the GROUP BY clause. As with the other group functions, the WHERE clause is evaluated first when Oracle executes the query.

Using the HAVING Clause

As you just saw, it is possible to limit the rows considered by using a WHERE clause. But what happens when you want to limit the rows returned based on the result of a group function? Consider the previous example. Suppose, rather than limiting the result set to U.S. centers, you want to limit the result set to those centers with an average cost less than $500. How do you write this query? If you place the condition in the WHERE clause, you actually receive an error:

```
SELECT City, AVG(PerDiemCost) AS "Average Cost"
FROM Instructors
WHERE AVG(PerDiemCost) < 500
GROUP BY City;

WHERE AVG(PerDiemCost) < 500
      *
ERROR at line 3:
ORA-00934: group function is not allowed here
```

The source of this error is easy to understand if you remember one key fact: The WHERE clause is evaluated *before* the group function values are calculated. In this example, the WHERE clause must be evaluated before the AVG(PerDiemCost) function can be calculated; however, you cannot evaluate the WHERE clause *until* you calculate this value — catch -22. It is for this reason that SQL includes a HAVING clause.

The HAVING clause is essentially a second WHERE clause that is evaluated *after* Oracle has calculated the grouping values. To fix the previous query, you can write something like this:

```
SELECT City, AVG(PerDiemCost) AS "Average Cost"
FROM Instructors
GROUP BY City
HAVING AVG(PerDiemCost) < 500;

CITY                           Average Cost
------------------------------ ------------
New York                              487.5
Toronto                                 450
```

You can still include a WHERE clause, but any conditions based on the outcome of a group function must be placed in the HAVING clause. For example, if you want U.S. centers with an average per diem cost below $500, you can use the following query:

```
SELECT City, AVG(PerDiemCost) AS "Average Cost"
FROM Instructors
WHERE Country = 'USA'
GROUP BY City
HAVING AVG(PerDiemCost) < 500;

CITY                           Average Cost
------------------------------ ------------
New York                              487.5
```

The WHERE clause is evaluated first in this query. The rows returned by the WHERE clause are the criteria for the grouping function. Finally, the HAVING clause condition is applied to the result of the grouping function.

The execution order of a SQL statement is shaped partially by the order the various clauses are listed in the query. When the Oracle parser finds clauses where it does not expect them, it can raise an error. The standard order of all of the clauses in a SQL query is:

```
SELECT ...
FROM ...
WHERE ...
GROUP BY...
HAVING ...
ORDER BY...
```

Key Point Summary

Oracle contains a number of functions that can be used to manipulate and modify data in the result set. These functions can be divided into two basic groups: single-row functions and group or multi-row functions. With a single-row function, one value is returned for each value that is input into the function. With a group function, one and only one value is returned regardless of the number of values passed into the function.

✦ Single-row functions:

- You can alter the datatype of the result set using the TO_NUMBER, TO_DATE, and TO_CHAR functions.
- TO_CHAR can be used to convert both numeric and date data.
- TO_DATE is used to convert non-NLS_date_format date strings into Oracle's internal date format.

- The UPPER, LOWER, and INITCAP functions can be used to alter capitalization. They can be used in the WHERE clause when the case of character data in the table is unknown.
- You can use the SUBSTR, CONCAT, LPAD, and TRIM functions to alter the appearance of character data.
- You can use the LENGTH and INSTR functions to return information about string data.
- With numeric data, you can use the ROUND and TRUNC functions to deal with decimal values.
- The MOD function returns the remainder of two numeric values.
- MOD always returns the first value when the second value is 0.
- When you add a number to a date value, the result is another date value. However, if you want to add months, you must use the ADD_MONTH function.
- You can use the MONTHS_BETWEEN, NEXT_DAY, and LAST_DAY functions to calculate date values.
- You can also use the ROUND and TRUNC functions to manipulate date values to the nearest month or year.

Single-row functions can be nested within each other. When the query is executed, the innermost function is evaluated first, and the outermost function is evaluated last. Only the outermost function returns data for the result set.

✦ Group functions:

Group (or multi-row) functions are used to aggregate or summarize data. They take all rows passed into them and return a single value based on these rows. The main group functions are AVG, SUM, COUNT, MIN, and MAX.

- A WHERE clause can be used to limit the numbers of rows considered by a group function.
- You can have multiple group functions in a SELECT list.
- You cannot combine group functions and nonaggregate columns and expressions in a SELECT list without a GROUP BY clause.
- With the exception of COUNT(*), all group functions ignore NULLs.
- AVG and SUM return the average and total of all non-NULL rows. They accept only numeric datatypes.
- COUNT(*) returns the total number of rows in a table; COUNT(column) returns the number of non-NULL fields in a column.

- MIN and MAX return the highest and lowest values in a column. They can be used with character, numeric, and date datatypes.
- The DISTINCT keyword can be used within any group function to force the function to ignore duplicate values in a column.
- If you want a group function to include NULL fields, you must use the NVL function to assign a literal value to these fields.

✦ GROUP BY and HAVING

The GROUP BY clause is used to group data together and create subtotals with group functions. If you want aggregate and nonaggregate values in the same result set, you must use a GROUP BY clause. The HAVING clause is, basically, a second WHERE clause that is evaluated *after* the group functions are evaluated. This enables you to include group functions in conditions for the result set.

- You cannot have nonaggregate columns or expressions in the SELECT list unless they are also listed in the GROUP BY clause.
- SQL groups by the unique occurrence of all values listed in the GROUP BY clause.
- You cannot use group functions in a WHERE clause; you can use them in the HAVING clause.
- A query can contain both a WHERE and HAVING clause.

✦ ✦ ✦

STUDY GUIDE

In this chapter, we have looked at Oracle's use of single-row and group functions. For the exam, you are expected to be familiar with uses of the functions listed in this chapter and how they work. You also are expected to understand the purpose and correct use of the GROUP BY and HAVING clauses and to understand queries using all of these language elements.

Assessment Questions

1. Which of the following functions can be used only with numeric values? (Choose all that apply.)

A. AVG

B. MIN

C. LENGTH

D. SUM

E. ROUND

2. Which function do you use to find the position of the letter "x" in a given character string?

A. CONCAT

B. INSTR.

C. STRCOUNT

D. SUBSTR

3. Consider the following query:

```
SELECT CONCAT('THE STRING LENGTH IS: ',
TO_CHAR(ROUND(NVL(LENGTH('Hello There'), 0), 0)))
FROM Dual;
```

In what order will these queries be executed?

A. CONCAT, TO_CHAR, ROUND, NVL, LENGTH

B. CONCAT, LENGTH, NVL, ROUND, TO_CHAR

C. TO_CHAR, ROUND, NVL, LENGTH, CONCAT

D. LENGTH, NVL, ROUND, TO_CHAR

4. What result is generated when the following statement is executed?

```
SELECT MOD(1500, 0)
FROM Dual;
```

A. NULL

B. 0

C. 1500

D. An Oracle error message

5. You issue the following query against the Instructors table:

```
SQL> SELECT InstructorID, InstructorType,
  2  MAX(PerDiemCost) AS "Highest Sal"
  3  FROM Instructors
  4  WHERE Country <> 'Canada'
  5  GROUP BY InstructorType
  6  HAVING MAX(PerDiemCost) > 500
  7  ORDER BY "Highest Sal";
```

Which line in this statement causes an error?

A. 1

B. 5

C. 6

D. 7

6. Which function do you use to remove all padded characters to the right of a character value in a column with a char datatype?

A. RTRIM

B. RPAD

C. TRIM

D. TRUNC

7. Which statement do you use to eliminate padded spaces between the month and day values in a TO_CHAR function?

A. TO_CHAR(SYSDATE, 'Month TRIM(DD), YYYY')

B. TO_CHAR(SYSDATE, 'fmMonth DD, YYYY')

C. TO_CHAR(SYSDATE, 'Month spDD, YYYY'

D. TO_CHAR(SYSDATE, 'Month DDfm YYYY')

8. You issue the following query:

```
SELECT TO_CHAR(SYSDATE, 'Dy. "THE" DDspth "of" MONTH,
                YYYY')
FROM Dual;
```

Assuming the current date is 12-JAN-01, which of the following reflects the output of the query?

A. Fri. THE TWELTH of JANUARY, 2001

B. FRI the 12th of January, 2001

C. Friday THE TWELTH of JANUARY, 2001

D. Fri THE TWELTH of JANUARY, 20001

9. You want to write a query that returns the date of the Monday after each course so you can schedule followup letters to be sent after the course is over. Which query do you use to find this information?

A. SELECT StartDate + 7 AS Next_Monday FROM ScheduledClasses;

B. SELECT ADD_WEEK(StartDate, 1) AS Next_Monday FROM ScheduledClasses;

C. SELECT NEXT_DAY(StartDate, Monday) AS Next_Monday FROM ScheduledClasses;

D. SELECT NEXT_DAY(StartDate, 'Monday') AS Next_Monday FROM ScheduledClasses;

10. You execute the following query:

```
SQL>  SELECT InstructorType,
 2           AVG(PerDiemCost + NVL(PerDiemExpenses, 0)) AS
 3            AVG_SAL
 4    FROM Instructors
 5    GROUP BY InstructorType
 6    WHERE Country = 'USA'
 7    HAVING MAX(PerDiemCost) > 500
 8    ORDER BY AVG_SAL
```

Which line returns an error?

A. 2

B. 5

C. 6

D. 7

E. 8

Scenarios

1. You have been asked to create a report that evaluates the status of the various class enrollments. Working with the ClassEnrollment table:
   - **A.** How do you find the number of classes that currently have students in them?
   - **B.** Each registration has a status (confirmed, canceled, on hold). You have been asked to write a query that returns the number of students in each category. How do you write this query?
   - **C.** Your manager wants this query further broken down by ClassID. Is it possible to group by both ClassID and status?
   - **D.** How do you include both elements in the grouping?
   - **E.** Your manager also wants you to displace the price of each course. Can you also include the price but not include it in the grouping?
2. You have been asked to generate a report on the instructors your company uses.
   - **A.** You have been asked to generate a list of each instructor and his or her total weekly cost with and without travel expenses. You notice that when you execute the query, some of the instructors have NULLs in the two calculated columns. What is the cause of these NULLs, and how can you remove them?
   - **B.** Your manager wants the list formatted so that the values appear with a dollar sign, decimal point, and a thousands separator. How do you add these elements to your output?
   - **C.** This script will also be run at your offices in Brussels and Paris. Those offices want the output formatted with their local currency symbols. How do you alter your script so it returns the local currency symbol rather than the dollar sign (assumimg that each location is using its own NLS settings)?
   - **D.** Your manager has asked that the report also contain the average local and traveling costs. Can you add these values to your current query? If not, why?

Lab Exercises

Lab 3-1 Using single-row functions

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.

2. Write a query that generates a new student identifier by concatenating the first four characters of each student's first name and the second to fifth characters of their last names. Name the column Student_code. Include the students' first and last names in the query.
3. Attempt to modify the previous query to include their middle initials between the portions of the first and last names.
4. Was the query successful? If not, why?

Lab 3-2 Using conversion functions

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.
2. Write a query that returns the InstructorID and PerDiemCost for five days for all instructors. Format the cost value in the following manner $###,###.00. Call the column TOTAL_COST.
3. Rewrite this query to include the PerDiemExpenses column.
4. When you run the query, do all instructors have a value in their TOTAL_COST columns?
5. Rewrite the query so that all instructors have a value in the TOTAL_COST column.
6. Currently the following query returns an error.

```
SELECT CourseNumber, LocationID
FROM ScheduledClasses
WHERE StartDate = 'Saturday January 6, 2001'
```

   Without altering the date string, how can you rewrite this query to return a result set?

Lab 3-3 Using character functions

1. Write a query that graphically displays each instructor and his or her PerDiemCost in a single column with an asterisk (*) representing every $10 of the cost. Your output should look like this:

```
Instructor Cost Chart
--------------------------------------------------------------------------------
210  ************************************
410  ************************************
310  *******************************************
450  *******************************************
300  ************************************************
110  ************************************************
100  ************************************************************
200  ********************************************************************************
```

```
8 rows selected.
```

Order the data by PerDiemCost with the lowest first and the highest last.

Lab 3-4 Using group functions

1. Write a query that returns the highest and lowest PerDiemCost values and a third column that is the difference between the two. Give each column the appropriate name.
2. Rewrite the previous query so that it returns information only for Canadian instructors.
3. Write a query that returns each Instructor Type and the number of instructors of that type. The query should also return the total number of instructors. Your output should look like this:

```
   TOTAL    ORACLE      UNIX      PROG
-------- --------- --------- ---------
       8         3         3         2
```

Lab 3-5 Using GROUP BY and HAVING clauses

1. Write a query that returns the average and highest PerDiemCost for each Instructor Type.
2. Rewrite this query so that it does not include the "Prog" Instructor Type.
3. Rewrite the query so that it also excludes any Instructor Type with a maximum PerDiemCost greater than $700.

Answers to Chapter Questions

Chapter Pre-Test

1. The primary difference between single-row and multi-row or group functions is that a single-row function returns one value for each value passed into it, whereas a group function returns only one value that is a summary of all the values passed into it.
2. A case conversion is most useful in a WHERE clause when you don't know the case of a character value in a table. Character values in WHERE clauses are case sensitive, and if you set a string with the wrong case as search criteria, the value will not be found. For example, if a last name value is stored in a table as "Smith" and you issue the condition WHERE lastname = 'SMITH', no rows are returned. Whereas, if you issue the condition WHERE UPPER(lastname) = 'SMITH', the value is returned, regardless of its case in the table.

3. To format a date to include ordinal numbers, you need to include the "th" suffix to the day value in the format string of the TO_CHAR function (for example, TO_CHAR(SYSDATE, 'MONTH DDth, YYYY). The "th" suffix always returns the appropriate ordinal end (that is, 1st, 2nd, 3rd, 4th).
4. The MAX function accepts any character, number, or date datatype. It does not accept large object datatypes such as CLOB and BLOB.
5. The difference between COUNT(*) and COUNT(column) is that the COUNT(column) function counts all non NULL fields, whereas the COUNT(*) returns the total number of rows regardless of the presence of NULL fields.
6. To return the current date, you use the SYSDATE function.
7. All SELECT statements must have a value in the FROM clause; however, if you are calculating a value that does not draw data from a table source, you can use the dual table in the FROM clause.
8. All nonaggregate values in the SELECT list must also be included in the GROUP BY clause; otherwise, Oracle raises an error. When you have multiple columns in the GROUP BY clause, Oracle groups by the unique occurrence of all columns listed.
9. Group functions cannot be included in a WHERE clause because this clause must be evaluated before the group values are determined. If you want to limit the rows returned based on the result of a group function, you must use a HAVING clause.
10. To return a different value based on the value of another column, you have to use the DECODE function. DECODE works much like an IF-THEN-ELSE statement, enabling you to test every each row as it is passed into the function.

Assessment Questions

1. **A** and **D** — Only A and D are correct. The MIN function works with any character, numeric, or date datatype. The LENGTH function is a character function that returns the number of letters in a character value. The ROUND function works with both numeric and date values.
2. **B** — The INSTR function returns the first position of a specified letter within a character string. The CONCAT function concatenates two values. The SUBSTR function returns only a portion of a string. There is no STRCOUNT function in Oracle.
3. **D** — Only answer D is correct. When Oracle executes a query that contains nested functions, it ill executes the innermost function first and then works its way out to the outermost query. In this example, the LENGTH function passes a value to the NVL function. NVL in turn passes a value to ROUND and so on until it reaches the CONCAT function. Only CONCAT returns a value to the user.

4. **C** — Mathematically you would expect this function to return an error (because you cannot divide a value by 0). However, Oracle avoids the error by having the MOD function return the first value when the second value is 0. Therefore, this function returns 1500 (the initial value).
5. **A** — The error in this query is in line 1. The SELECT list for this query contains two nonaggregate columns (InstructorID and InstructorType); however, only one of these columns appears in the GROUP BY clause. In order for this query to execute successfully, all nonaggregate columns in the SELECT list must also appear in the GROUP BY clause.
6. **C** — The TRIM function is used to remove padded spaces. LTRIM and RTRIM functions were included in earlier versions of Oracle, but Oracle 8*i* has replaced them with a single TRIM function (although both are still included in the product for backwards compatability). The TRUNCATE function is used with only numeric and date columns. RPAD has the opposite function of TRIM; it is used to pad columns with additional characters.
7. **B** — To remove padded spaces, you use the "fm" prefix before the date element that contains the spaces. The TRIM function cannot be used in the format string of a TO_CHAR function. The "th" suffix is used to add ordinal endings to number values.
8. **A** — When you use the TO_CHAR function with dates, the capitalization and punctuation in the format string is preserved in the output. For example, the format element Dy. returns a day abbreviation with initial capital and a period at the end.
9. **D** — To find the next Monday after a particular date, you use the NEXT_DAY function. Simply adding seven days does not work unless the initial date is also a Monday. Within the NEXT_DAY function, you must include single quotes around the day parameter; otherwise, Oracle returns an error. There is no ADD_WEEK function in Oracle.
10. **C** — The error in this query is on line 6. Each clause individually is syntactically correct; however, the problem in this query is one of order. In a SQL query, the WHERE clause must appear before the GROUP BY clause. Because Oracle does not find the WHERE clause where it expected it, it returns an error. The proper order for SQL SELECT clauses is: SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY. Only the SELECT and FROM clause are mandatory.

Scenarios

1. Any course that shows up on the enrollment list, by definition, has students in it. (Otherwise, why is it listed in the ClassEnrollment table?) The problem is that some classes have more than one student enrolled in them, making a simple COUNT(*) impossible. To find this answer, you have to use the DISTINCT keyword in the COUNT function. For example:

```
SELECT COUNT(DISTINCT ClassID)
FROM ClassEnrollment;
```

To write a query that lists the number of students by status requires the presence of both aggregate and nonaggregate columns in the SELECT list. In order to have this mix, you must include a GROUP BY clause to group the results of you count by the Status Column.

It is possible to include ClassID and status in this query; however, you have some repetition because Oracle does not group on the unique combination of ClassID and Status. Both columns must appear in the GROUP BY clause and return output like this:

```
SELECT Status, ClassID, COUNT(StudentNumber) AS Count
FROM ClassEnrollment
GROUP BY Status, ClassID;

STATUS       CLASSID COUNT
---------- --------- --------------------
Cancelled         51                    1
Confirmed         50                    3
Confirmed         51                    2
Hold              53                    1
```

Notice that both the Confirmed status and Class 51 repeat, but the unique combination of the two columns does not.

You cannot, however, include the price for each course. Any nonaggregate column listed in the SELECT list must be included in the GROUP BY clause and thereby included in the grouping.

2. The NULLs that you are noticing are most likely the result of NULL values in the underlying columns. When you include a NULL in an arithmetic expression, the result is always NULL. You can deal with this problem by using the NVL function to replace the NULLs with a value of 0. The zero does not affect the accuracy of the expression, but because it is a literal value, it allows the expression to be calculated. It looks something like this:

```
SELECT InstructorID, NVL(PerDiemCost, 0) * 5 AS NO_TRAVEL,
       (NVL(PerDiemCost,0) + NVL(PerDiemExpenses, 0) * 5
       AS WITH_TRAVEL
FROM Instructors;
```

In order to format the output, you need to use the TO_CHAR function with the appropriate formatting options. The resulting query looks something like this:

```
SELECT InstructorID, TO_CHAR(NVL(PerDiemCost, 0) * 5,
       '$99,999.99') AS NO_TRAVEL,
       TO_CHAR((NVL(PerDiemCost,0) +
       NVL(PerDiemExpenses,0)* 5), '$99,999.99')
       AS WITH_TRAVEL
FROM Instructors;
```

In order to make the script portable to your European offices, you need to include a formatting character that uses the NLS currency symbol rather than the static dollar sign. The formatting character that does this is the “L”. This

character indicates a floating currency symbol but defers to the NLS parameters to determine which symbol to use. In this example, you can run the same script in Brussels and Paris, and each country gets its own currency symbol. The format string of the TO_CHAR function looks like this: 'L999,999.99'.

As with the previous scenario, you cannot use an aggregate function in this query. If you attempt to place an AVG function in the SELECT list, you have to group by the unique occurrence of the InstructorID (because it is the Primary KEY) and the results of both arithmetic expressions.

Lab Exercises

Lab 3-1 Using single-row functions

2.

```
SELECT CONCAT(SUBSTR(FirstName, 1, 4), SUBSTR(LastName,
2,4))AS Student_Code, FirstName, LastName
FROM Students;
```

3.

```
SELECT SUBSTR(FirstName, 1, 4) || MiddleInitial ||
SUBSTR(LastName, 2,4) AS Student_Code, FirstName, LastName
FROM Students
```

4. The query is successful only if you use the concatenation operators rather than the CONCAT function. This function accepts only two values, whereas you can link multiple values together with the operator. The concatenation operators cannot be embedded in all functions.

Lab 3-2 Using conversion functions

2.

```
SELECT InstructorID, TO_CHAR((PerDiemCost * 5),
 '$999,999.99') AS TOTAL_COST
FROM Instructors;
```

3.

```
SELECT InstructorID, TO_CHAR(((PerDiemCost + PerDiemExpenses)
  * 5), '$999,999.99') AS TOTAL_COST
FROM Instructors;
```

4. No. Some of the instructors have a NULL in the TOTAL_COST column. This is because two instructors have NULLs in the PerDiemExpenses column.

5.

```
SELECT InstructorID, TO_CHAR(((NVL(PerDiemCost, 0) +
NVL(PerDiemExpenses,0))
  * 5), '$999,999.99') AS TOTAL_COST
FROM Instructors;
```

6.

```
SELECT CourseNumber, LocationID
FROM ScheduledClasses
WHERE StartDate = TO_DATE('Saturday January 6, 2001', 'Day
Month DD, YYYY'
```

Lab 3-3 Using character functions

1.

```
SELECT InstructorID || ' ' || RPAD(' ', PerDiemCost/10, '*')
FROM Instructors
Order by PerDiemCost;
```

In this query, you need to use the RPAD function to add the asterisk (*) (RPAD is simply LPAD in the other direction). The way this query works is to add a blank space and pad the rest of the row with asterisks up to a width of the PerDiemCost value for each row divided by ten. Because RPAD is a single-row function, the value is recalculated for each row based on the current PerDiemExpenses column. Therefore, its width is different for each instructor.

Lab 3-4 Using group functions

1.

```
SELECT MAX(PerDiemCost) AS HIGHEST, MIN(PerDiemCost)
AS LOWEST, MAX(PerDiemCost)- MIN(PerDiemCost) AS
      DIFFERNCE
FROM Instructors;
```

2.

```
SELECT MAX(PerDiemCost) AS HIGHEST, MIN(PerDiemCost)
AS LOWEST, MAX(PerDiemCost)- MIN(PerDiemCost) AS
      DIFFERNCE
FROM Instructors
WHERE Country = 'Canada';
```

3.

```
SELECT COUNT(*) AS TOTAL, SUM(DECODE(TRIM(InstructorType),
'ORACLE', 1,0)) AS Oracle,
       SUM(DECODE(TRIM(InstructorType), 'UNIX', 1,0)) as
UNIX,
       SUM(DECODE(TRIM(InstructorType), 'Prog', 1,0)) as PROG
FROM Instructors
```

For each column (with the exception of Total), you need to test the value of InstructorType before deciding whether or not to count it. To do this test, you need the DECODE function. In the Oracle column, for example, the contents of the InstructorType column is evaluated for each row. If the value is Oracle, a 1 is passed to the SUM function; if it is not Oracle, a 0 is passed. The SUM

function then totals all of the 1s and 0s it has been passed and derives a value. Because the InstructorType column is a fixed-length column (a char(10), in this case), you must use the TRIM function to eliminate any of the hidden padding characters; otherwise, Oracle cannot match the values in the column to the strings in the DECODE function.

Lab 3-5 Using GROUP BY and HAVING clauses

```
SELECT InstructorType, AVG(PerDiemCost) AS AVERAGE,
Max(PerDiemCost) AS HIGHEST
FROM Instructors
GROUP BY InstructorType;
```

2.

```
SELECT InstructorType, AVG(PerDiemCost) AS AVERAGE,
Max(PerDiemCost) AS HIGHEST
FROM Instructors
WHERE InstructorType <> 'Prog'
GROUP BY InstructorType;
```

3.

```
SELECT InstructorType, AVG(PerDiemCost) AS AVERAGE,
Max(PerDiemCost) AS HIGHEST
FROM Instructors
WHERE InstructorType <> 'Prog'
GROUP BY InstructorType
HAVING MAX(PerDiemCost) > 700;
```

CHAPTER 4

Advanced SELECT Statements

EXAM OBJECTIVES

- Displaying Data from Multiple Tables
  - Write SELECT statements to access data from more than one table using equality and nonequality joins
  - View data that generally does not meet a join condition by using outer joins
  - Join a table to itself
- Producing Readable Output with SQL*Plus
  - Produce queries that require an input variable
- Subqueries
  - Describe the types of problems that subqueries can solve
  - Define subqueries
  - List the types of subqueries
  - Write single-row and multiple-row subqueries
- Multiple-Column Subqueries
  - Write multiple-column subqueries
  - Describe and explain the behavior of subqueries when NULL values are retrieved
  - Write subqueries in a FROM clause

CHAPTER PRE-TEST

1. What three syntax elements are required to write a query that joins multiple tables together?
2. What happens if you omit a join condition when referencing more than one table?
3. Can you join a table to itself?
4. When would you want to use an outer join?
5. How many columns can be returned by a subquery?
6. What happens when a subquery returns a NULL?
7. Which clauses of a query can contain a subquery?
8. When do you use a correlated subquery?
9. How can you control the execution of a SQL statement at runtime?
10. Does Oracle's implementation of SQL have a runtime variable?

✦ Answers to these questions can be found at the end of the chapter. ✦

This chapter expands upon the information in Chapter 2, "Retrieving Data Using Basic SQL Statements." In Chapter 2, you learned how to write a basic SQL query against a single table; in this chapter, you learn how to expand this query to consider two or more tables. In a properly normalized database, information is often stored in multiple tables linked by primary and foreign keys. For example, you may have a column containing a ClassID value in the ScheduledClasses table, but the actual name of that course is stored in the Courses table. In order to write a query with schedule information and the name of the course, you have to join these tables in the query. This chapter shows you how to write an Oracle join to combine data from multiple tables. In addition to joins, you also learn how you can use subqueries to draw information from one table, based on values drawn from the same or another table.

This chapter also looks at the SQL set operators. These operators enable you to combine multiple queries into a single result set. This chapter ends with some miscellaneous advanced query topics, which, although not necessarily covered on the exam, are worth knowing when you start working extensively with Oracle's SQL.

Oracle is a relational database management system (RDBMS). The implication of being relational is that data is often stored in more than one related table. Data is stored in more than one table for many reasons, and one is to avoid redundancy. Consider the registration of students. If one single table contains the student's information (such as name and phone number) in additon to all of the course information (name of class and start date), you have to repeat all of the student information every time a student enrolls in a different course. Normalizing the tables eliminates this redundancy; however, when querying data, it means that you often must go to two or more places to find all of the information that you require. There are different methods of accessing multiple tables in a SQL query: joins, subqueries, and set operators. This chapter examines all of these methods. It will start by examining one of the most common methods of combining tables (joins.

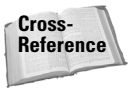
Working with Joins

- ✦ Write SELECT statements to access data from more than one table using equality and nonequality joins
- ✦ View data that generally does not meet a join condition by using outer joins
- ✦ Joining a table to itself

All SQL queries require two elements — a SELECT list (which specifies what to retrieve) and a FROM clause (which specifies where to retrieve the items in the SELECT list from). In the all of the examples in Chapter 2, only one table is referenced in the FROM clause, but it is possible to include more than one table in the FROM clause of a query. This increases the scope of columns that can be referenced in the SELECT list.

It is important to understand how to work with joins, not just because it is required for the exam, but because the majority of queries you will write in the real world will include joins. Consider the training company tables (Courses, ScheduledClasses, Instructors, etc.) that were added to your database in the labs for Chapter 1. If you want to write a query that looks up a particular class starting on a particular day, you can simply write a query against the ScheduledClasses table. However, if you want that query to include the name of the course, the name of the instructor that is teaching it, and the city it is running in, you have to include information from the Instructors, Courses, and Locations tables. To do this, you need to use a join.

Writing a join requires more than just a SELECT list and a FROM clause; it also requires a join condition. A join condition is a way to equate specific rows in one table with specific rows in another table. This condition is most often a primary key/foreign key pair and is expressed in the WHERE clause.



Primary keys and foreign keys are discussed in detail in Chapter 7, "Creating and Managing Oracle Database Objects."

The join condition is defined in the syntax of the join. The basic join syntax looks like this:

```
SELECT {* | [exp, col1][,col2 ...]}
FROM Table1, Table 2
WHERE Table1.col <cond> Table2.col
```

You need to be comfortable with four basic types of joins: equijoins, nonequijoins, outer joins, and self-joins.

Working with equijoins

An equijoin joins two tables based on the equality of values in those tables; that is, an equijoin returns all rows that are common to the two tables. Consider the following situation. You want to list a particular class and the name and cost of the instructor who will be teaching it. To get this information, you need to join the Instructors table and the ScheduledClasses table. It is possible to join these two tables because the InstructorID column in the ScheduledClasses table is a foreign key to the InstructorID primary key in the Instructors table. Therefore, you can join the two tables where the InstructorID in the ScheduledClasses table is equal to the InstructorID in the Instructors table. The join looks like this:

```
SELECT ScheduledClasses.ClassID, ScheduledClasses.StartDate,
       Instructors.FirstName, Instructors.LastName,
       Instructors.PerDiemCost
FROM ScheduledClasses, Instructors
WHERE ScheduledClasses.InstructorID = Instructors.InstructorID;
```

```
CLASSID  STARTDATE FIRSTNAME          LASTNAME                  PERDIEMCOST
-------- --------- ------------------ ------------------------- -----------
      50 06-JAN-01 David               Ungar                     600
      51 13-JAN-01 Lisa                Cross                     750
      53 14-FEB-01 Kyle                Jamieson                  500
```

In this example, a single query and a single result set draw information from two separate tables.

This type of join is referred to as an *equijoin* because it returns data from a table *only* when the join condition exists in both tables. Eight instructors are listed in the Instructors table, but only three appear in the result set. The reason that only these instructors are returned is that their InstructorIDs appear in both tables. All of the other instructors exist only in the Instructors table and are, therefore, not returned. When Oracle processes this query, it takes the instructor ID from the ScheduledClasses table for each row and uses that value to locate the corresponding first name, last name, and per diem cost values. It ignores all other instructors in the table. If some classes do not have instructors assigned to them, they also are excluded from the result set.

When you write this query, you do not have to include the table source for each column, only for those that are ambiguous. A column is considered ambiguous when it exists in both tables. If you were to execute the previous query without any table references, you would receive the following error:

```
WHERE InstructorID = InstructorID;
                     *
ERROR at line 3:
ORA-00918: column ambiguously defined
```

This error is raised because Oracle is presented with an InstructorID column in both tables and is unable to determine which InstructorID column is referenced by either side of the join condition statement. All of the other columns in the query are unique to one table or the other and, therefore, do not raise any ambiguity errors. To execute this join correctly using the fewest number of table references, the query looks like this:

```
SELECT ClassID, StartDate, FirstName, LastName, PerDiemCost
FROM ScheduledClasses, Instructors
WHERE ScheduledClasses.InstructorID = Instructors.InstructorID;
```

Because only InstructorID is ambiguous, it is the only column that must be defined to avoid ambiguity.

With an equijoin, it is even possible to join more than two tables. To do so, you simply include all tables in the FROM clause and then include all of the join conditions in the WHERE clause separated by an AND operator.

Using table aliases in joins

One practice that is very useful (although not necessary when working with equijoins) is the use of aliases. In Oracle, you can give tables alias names in the FROM clause. You can then use those aliases in the rest of the query (including the SELECT list). Using these aliases with all column objects makes the code more readable and avoids the possibility of ambiguous column references. If you were to rewrite the previous example with aliases, it would look like this:

```
SELECT s.ClassID, s.StartDate, i.FirstName, i.LastName,
i.PerDiemCost
FROM ScheduledClasses s, Instructors i
WHERE s.InstructorID = i.InstructorID;
```

You are able to use the alias even before you declare it because Oracle parses the entire statement before it attempts to reconcile the object names in the data dictionary. The table aliases are not included in the column headers in the result set.

Suppose that, in the previous example, you want the actual name of the course, a single column for instructor, and the city that the course is being held in. You can use the following query:

```
SELECT c.CourseName, s.StartDate, i.FirstName || ' '
||i.LastName as Instructor, l.City
FROM ScheduledClasses s, Instructors i, Courses c, Locations l
WHERE s.InstructorID = i.InstructorID
      AND s.CourseNumber = c.CourseNumber
      AND s.LocationId = l.LocationID;

COURSENAME                  STARTDATE INSTRUCTOR      CITY
--------------------------- --------- --------------- --------
Basic SQL                   06-JAN-01 David Ungar     New York
Database Performance Basics 13-JAN-01 Lisa Cross      Toronto
Basic SQL                   14-FEB-01 Kyle Jamieson   Toronto
```

Note that in this example, you must place a table reference on the City column. The reason is that both the Instructor and Locations tables contain City columns, and this causes ambiguity if you do not specifically reference it. This type of accidental ambiguity is easily avoided by using table aliases.

After you have referenced a table in the FROM clause, you can use any column from that table, even if that column is not mentioned in the SELECT list. For example, you can limit the previous query to courses running in the United States by including the Country column from the Locations table in the WHERE clause with another AND operator:

```
SELECT c.CourseName, s.StartDate, i.FirstName || ' '
||i.LastName as Instructor, l.City
FROM ScheduledClasses s, Instructors i, Courses c, Locations l
```

```
WHERE s.InstructorID = i.InstructorID
      AND s.CourseNumber = c.CourseNumber
      AND s.LocationId = l.LocationID
      AND l.Country = 'USA';
```

Of course, none of this is possible unless you are able to associate all of the various tables together with the appropriate join conditions. In an equijoin, the number of conditions in the WHERE clause should be one less than the number of tables listed in the FROM clause. If this is not the case, your query will produce a CROSS JOIN.

Cross-joins

There is a real concern in omitting the join condition in the WHERE clause. When you list multiple columns in the FROM clause but do not include a join condition, Oracle must decided for itself how to join the two tables together. Because the software is not intuitive enough to come up with a complex relationship (such as InstructorID being equal to InstructorID, it simply joins every row in one table with every row in the second table. This is known as a *Cartesian Product* or a *cross-join*. Consider the output of the following example:

```
SELECT i.InstructorID, l.LocationID
FROM Instructors i, Locations l;

INSTRUCTORID LOCATIONID
------------ ----------
         300        100
         310        100
         100        100
         110        100
         200        100
         210        100
         410        100
         450        100
         300        200
         310        200
         100        200
         110        200
         200        200
         210        200
         410        200
         450        200
         300        300
         310        300
         100        300
         110        300
         200        300
         210        300
         410        300
         450        300

24 rows selected.
```

These two tables include eight instructors and three locations. Because you have not told Oracle how to relate the two tables together, it has simply joined every InstructorID with every LocationID (8 *3 = 24 rows returned). Imagine now if each one of these tables included over 100,000 rows; you might find yourself with more output than you expected!

A cross-join, however, can be useful if you want to create a sample table and populate it with a large amount of test data. Performing an insert based on a cross-join can, as you have seen, produce a large amount of data quickly.

Nonequijoins

In the previous equijoin examples, you joined the tables on a condition of equality (that is, where a value in one table was equal to a value in another table). For example, you were able to match classes with the names of the instructors teaching them by taking the InstructorID value from the ScheduledClasses table and finding an exact match with that InstructorID in the Instructors table. However, at times you may want to join a table on a condition of something other than simple equality. This is called a nonequijoin. The syntax for a nonequijoin is the same as the equijoin syntax. The only difference between the two is that the join condition uses a comparison operator other than the equal sign (=).

The examples for this section rely on tables that are not created by the sample data script (see Appendix E). You can create the two tables used by running the nonequijoin.sql script on the CD.

Consider the following situation. A number of sales people earn bonuses based on their annual sales. The bonus is calculated by setting low and high sale values. To calculate the bonus percentage for each sales person, you need to compare each salesperson's annual sales to the low and high range and determine which bonus they qualify for. The bonuses are calculated using a table called Bonus that looks like this:

```
SQL> DESC Bonus
 Name                           Null?    Type
 ------------------------------ -------- --------------------
 BONUS_RATE                              NUMBER(4,2)
 LOW_VALUE                               NUMBER(8,2)
 HIGH_VALUE                              NUMBER(8,2)
```

A table called Sales contains the name of each salesperson and their total sales for the year. The table looks like this:

```
SQL> DESC Sales;
 Name                            Null?    Type
 ------------------------------- -------- --------------------
 NAME                                     VARCHAR2(15)
 ANNUAL_SALES                             NUMBER(8,2)
```

In this example, you can clearly not use an equijoin to find the information you required because there is no common column between the two tables. If you join annual_sales to either the low_value or high_value, you only get a bonus_rate for those sales people who sold exactly the high or low value. No value is returned for those sales people whose annual sales fall somewhere between the two values.

The join condition then must be created between the annual_sales column and a range created by considering both the low_value and high_value. You must compare the annual salary for each salesperson to this range. Luckily SQL includes a comparison operator that enables you to view data in a range — the BETWEEN operator. Therefore, the join condition is that the annual salary in the Sales table must be matched to the range created by the low and high value columns. Such a query looks like this:

```
SELECT s.name, s.annual_sales, b.bonus_rate
FROM Sales s, Bonus b
WHERE s.annual_sales BETWEEN b.low_value AND b.high_value;

NAME            ANNUAL_SALES BONUS_RATE
--------------- ------------ ----------
Sam                     3200         .1
Paul                    2400         .1
Leslie                  7200        .25
Phil                    9900        .25
Louis                   6400        .25
Stewart                13450         .3
Peter                  16245         .4
Scott                  18400         .4
Fred                   17050         .4
Jane                   22000        .45
Larry                  24990        .45
Emily                  64000         .5
Eve                    27100         .5
Bill                   28000         .5
```

As with the equijoin, after the conditional relationship has been established between the two tables, you can reference any column in the query.

For example, if you want to limit the result set to only those with a minimum annual sales of $20,000 and calculate the actual dollar value of the bonus, you can use the following:

```
SELECT s.name, s.annual_sales, b.bonus_rate, (b.bonus_rate *
s.annual_sal)
       AS "Bonus Amount"
FROM Sales s, Bonus b
WHERE s.annual_sales BETWEEN b.low_value AND b.high_value;
      AND b.low_value >= 20000;
```

```
NAME            ANNUAL_SALES BONUS_RATE Bonus Amount
--------------- ------------ ---------- ----------------
Jane                   22000        .45             9900
Larry                  24990        .45          11245.5
Emily                  64000         .5            32000
Eve                    27100         .5            13550
Bill                   28000         .5            14000
```

Outer joins

In the discussion of equijoins, you saw that an equijoin returns only those rows that are common to both tables referenced in the join condition. In the example that returned courses and the name of the instructors, the names of those instructors not scheduled to teach classes were not returned by the query. At times, however, you may want all elements from one table, regardless of whether there is a match between the two tables. For example, suppose you want a list of all locations and the courses running at those locations; to get a list of all courses and the locations where they are running, you can use the following query:

```
SELECT l.LocationID, l.City, s.Classid
FROM Locations l, ScheduledClasses s
WHERE l.locationid = s.locationid;

LOCATIONID CITY                           CLASSID
---------- ------------------------------ ---------
       100 New York                            50
       300 Toronto                             51
       300 Toronto                             53
```

This list does not provide the result set you are looking for because it does not include San Francisco. The reason for its omission is simple: No classes currently are scheduled in that center. If you want to write a query that returns all cities, regardless of the courses running, you require an outer join.

An outer join returns all data from one column in the join condition and matches it to the other table where possible. When Oracle is unable to find a match, it leaves all columns referenced in the second table NULL. An outer join is written using the outer join operator: (+). This operator is placed in the WHERE clause next to the column that will have NULL values returned if no match is found. Consider the previous query expressed as an outer join:

```
SELECT l.LocationID, l.City, s.Classid
FROM Locations l, ScheduledClasses s
WHERE l.locationid = s.locationid(+);

LOCATIONID CITY                           CLASSID
---------- ------------------------------ ---------
       100 New York                            50
       200 San Francisco
       300 Toronto                             51
       300 Toronto                             53
```

In this result set, all of the locations are returned, including San Francisco. The outer join operator has returned a NULL for its ClassID. You can use this behavior to find only those centers that do not have any classes running by using this generated NULL as an additional search argument in the WHERE clause. You can also use the NVL function to replace the NULL with a more meaningful output.:

```
SELECT l.LocationID, l.City,
       NVL(TO_CHAR(s.Classid), 'No Class') AS ClassID
FROM Locations l, ScheduledClasses s
WHERE l.locationid = s.locationid(+)
      AND s.Classid IS  NULL;

LOCATIONID CITY                           CLASSID
---------- ------------------------------ --------------------
       200 San Francisco                  No Class
```

The placement of the outer join operator can be a bit confusing. If you were to place the operator next to the l.LocationID, you would receive a result set that looks the same as an equijoin. This is because it returns all classes (with or without LocationID) and matches them to a city, if possible, or returns a NULL, if not possible. Because LocationID is the primary key for the Location table, all classes must have a LocationID; therefore, no NULL rows are in the result set. Always place the operator next to the column in the join condition that references the table that has the NULLs added to it.

Self-joins

All of the examples so far have involved joining multiple tables together; however, in some situations, it is necessary to join a table to itself. This is usually required when a table contains a self-referential key. A self-referential key occurs when one column is a foreign key to another column in the same table. Consider the following table:

```
SQL> DESC Management
 Name                          Null?    Type
 ----------------------------- -------- -----------------
 ID                            NOT NULL NUMBER(2)
 NAME                                   VARCHAR2(15)
 MANAGER                                NUMBER(2)
 POSITION                               VARCHAR2(15)
```

This table can be created using the selfjoin.sql script on the CD included with this book.

In this table, the Manager column contains the ID value for each employee's manager. If you want to write a query that returns the name of each employee, their position, and the name of their manager, you have to join a row in the Management table with another row in the same table. This is called a *self-join*.

In order to write a self-join, you must use table aliases and reference the same table twice using different aliases, and you must use the join condition that establishes a relationship between employees and their managers:

```
SELECT emp.name, emp.position, mgr.name
FROM Management emp, Management mgr
WHERE emp.Manager = mgr.ID;

NAME            POSITION        NAME
--------------- --------------- ---------------
Bob             Vice-President  Jane
Susan           Vice-President  Jane
Rich            Manager         Bob
Ellen           Manager         Bob
Jim             Manager         Susan
Sam             Manager         Susan
Joan            Analyst         Rich
Pat             Supervisor      Ellen
Fred            Supervisor      Ellen
Joe             Developer       Sam
Stu             Accountant      Pat
Arthur          Operator        Fred
```

To process this query, Oracle retrieves each row and finds the Manager number for each employee; it then goes back into the Management table and finds an employee ID value that matches that Manager number. After it finds a match in the ID column, Oracle returns the name that corresponds to the second ID value.

Notice in this result set that Jane does not appear in the Name column. The reason is that the query has performed a self-equijoin and Jane (as company president) does not have a manager (and, therefore, has a NULL in the Manager column). Because a NULL is in her Manager column, the equijoin does not return a row for her. If you want to return all employees, you must use a self outer join. A self-outer join works just like a normal outer join. In this case, you put the outer join operator on the Mgr.ID column because you want to return a NULL for Jane's manager:

```
  SELECT emp.name, emp.position, mgr.name
  FROM Management emp, Management mgr
  WHERE emp.Manager = mgr.ID(+)
  ORDER BY mgr.name DESC

NAME            POSITION        NAME
--------------- --------------- ---------------
Jane            President
Jim             Manager         Susan
Sam             Manager         Susan
Joe             Developer       Sam
Joan            Analyst         Rich
Stu             Accountant      Pat
Bob             Vice-President  Jane
Susan           Vice-President  Jane
Arthur          Operator        Fred
```

```
Pat             Supervisor      Ellen
Fred            Supervisor      Ellen
Rich            Manager         Bob
Ellen           Manager         Bob
```

As with the other outer joins, you can also use the NVL function to provide a value in place of the null generated by the outer join.

Working with Subqueries

Objective

- Describe the types of problems that subqueries can solve
- Define subqueries
- List the types of subqueries
- Write single-row and multiple-row subqueries
- Write multiple-column subqueries
- Describe and explain the behavior of subqueries when NULL values are retrieved
- Write subqueries in a FROM clause

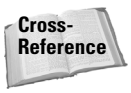
Working with basic subqueries

Other than joins, other ways of writing queries derive their information from one or more tables. One of these is the subquery. A subquery is an embedded query that runs inside the context of another query. Subqueries are often used to answer a question that must be answered before a greater question can be answered. Consider the following query: "Which instructor(s) charge a higher per diem than the instructor who is teaching class 53?" In order to answer this question, you actually need to ask two questions: "How much does the instructor teaching class 53 charge?" and "Who charges more than that person?" The first question is implied in the second and must be answered before the second question can be addressed.

In order to express this embedded question in SQL, you use a subquery. Subqueries are most often contained in the WHERE clause (although they can show up in most clauses including the SELECT list). The SQL for the previous example looks like this.

```
SELECT FirstName, LastName, PerDiemCost
FROM Instructors
WHERE PerDiemCost > (SELECT i.PerDiemCost
                     FROM Instructors i, ScheduledClasses s
                     WHERE i.IinstructorID = s.InstructorID
                     AND s.ClassID = 53);

FIRSTNAME               LASTNAME            PERDIEMCOST
----------------------- ------------------- ------------
David                   Ungar               600
Lisa                    Cross               750
```

All subqueries (regardless of which clause they are contained in) must be enclosed in parentheses. The subquery executes first and passes its values to the outer query. The subquery can contain any SQL elements except the ORDER BY clause. The ORDER BY clause must be the last clause executed in a SQL statement. Because the subquery is considered part of the calling query, placing an ORDER BY clause in the subquery causes it to execute before the outer query executes, which causes a syntax error.



As with most things, there is an exception to this rule. The ORDER BY clause can be used when working with inline views. See "Working With Inline Views" later in this chapter.

It is also possible to nest subqueries inside of subqueries. Oracle does not impose any limit on the number of nested subqueries (although there are some practical performance-based limitations when you nest too many subqueries). With nested subqueries, the innermost subquery executes first and passes its value up to the next level and so on until you reach the outermost query. Only the outermost query returns a result set to the user.

As an example of a nested subquery, it is possible to rewrite the previous query using a second subquery rather than a join:

```
SELECT FirstName, LastName, PerDiemCost
FROM Instructors
WHERE PerDiemCost >(SELECT PerDiemCost
                    FROM Instructors
                    WHERE InstructorID = (SELECT InstructorID
                                          FROM ScheduledClasses
                                          WHERE ClassId = 53));
```

In performance terms, the join is more efficient than the nested subquery. You should avoid using nested subqueries in the real world if it is possible to use a join (particularly if you have appropriate indexes on all of the columns in the join condition).

In this second example, the innermost subquery determines who teaches class 53 and passes this information to the outer subquery. The outer subquery determines that instructor's per diem, and it is that value which is passed to the outer query. Only the outer query returns a result set.

Working with subqueries that return multiple rows

In all of the previous examples, the subquery is limited to returning one row. This limit is imposed because of the comparison operator. All of the comparison operators in SQL can accept only one value.

The use of comparison operators is discussed in detail in Chapter 2, "Retrieving Data Using Basic SQL Statements."

The effect of NULL subqueries

In all queries, if an inner query returns a NULL, the outer query also returns NULL. The outer query requires that the inner query provide a value in order to evaluate itself. If the comparison operator in the WHERE clause is presented with a NULL, it nullifies the outer query. Consider the output for the following query:

```
SELECT FirstName, LastName, PerDiemCost
FROM Instructors
WHERE PerDiemCost > (SELECT PerDiemCost
                     FROM Instructors
                     WHERE InstructorID = 0);

no rows selected
```

In this query, the output is simply no rows selected. The reason for this outcome is because the Instructors table does not include an instructor 0. When the subquery is run, it is unable to determine a value for PerDiemCost, so it returns a NULL to the outer query. Remember that a NULL is not the same as a zero or a blank; rather, it is a value that is unknown. Because it is unknown, Oracle is unable to find which instructors earn more (it simply can't determine what is greater than "I don't know"). Because the outer query is unable to return a value, the "no rows selected" message is returned. The only exception to this rule is when you use the IS NULL or IS NOT NULL operator in the outer WHERE clause. In this case, the NULL generated by the subquery is converted into a Boolean TRUE or FALSE value. This Boolean can be used by the outer query to return a value.

For example, suppose you want to find all of the instructors who charge a higher per diem than any of the Oracle trainers. If you entered the following query, you receive an error:

```
SELECT InstructorId, InstructorType, PerDiemCost
FROM Instructors
WHERE PerDiemCost > (SELECT PerDiemCost
                     FROM Instructors
                     WHERE InstructorType = 'ORACLE');

WHERE PerDiemCost > (SELECT PerDiemCost
                     *
ERROR at line 3:
ORA-01427: single-row subquery returns more than one row
```

The greater than operator is expecting only one row; however, the table includes three Oracle instructors. Therefore, three PerDiemCost values are returned. In order to make this query work, you need an operator that can handle more than one row. Oracle provides two operators for this particular situation: ANY and ALL. Each of these work slightly differently, and you should understand the distinction and where the operators are used.

The ANY and ALL operator can be used in conjunction with any comparison operator. The ANY operator is used when you want to compare against any individual result, the ALL operator is used when you want to compare the result against all of the combined results. The difference between the two can be best illustrated using the previous example.

If you were to rewrite the last query using the ANY operator, you would receive the following result:

```
SELECT InstructorId, InstructorType, PerDiemCost
FROM Instructors
WHERE PerDiemCost > ANY (SELECT PerDiemCost
                         FROM Instructors
                         WHERE InstructorType = 'ORACLE');

INSTRUCTORID INSTRUCTOR PERDIEMCOST
------------ ---------- -----------
         100 ORACLE             600
         200 UNIX               750
```

In this case, one of the instructors returned is an Oracle instructor. This instructor is returned because he charges more than the other two Oracle instructors (who both charge $500 per day). With the ANY operator, the subquery returns all of its rows, and each row in the outer query is evaluated against all of these values. As long as the WHERE condition is satisfied by one of the values passed out of the subquery, the row is returned in the result set. In this particular example, the values 500 (twice) and 600 are passed out of the subquery to the WHERE clause. All rows in the Instructors table are then tested against the following condition:

```
WHERE PerDiemCost > 500 or PerDiemCost > 600
```

Instructor 100 charges $600 per day and thus satisfies the first WHERE condition, and the row is returned. If you want to find out which instructor charges more than all of the Oracle instructors, you must use the ALL operator.

You can replace =ANY with the IN operator. Both of these operators evaluate to: WHERE col = x or col = y [or col = ...].

The ALL operator combines every value returned from the subquery. Consider the outcome of the previous example when you use ALL instead of ANY:

```
SELECT InstructorId, InstructorType, PerDiemCost
FROM Instructors
WHERE PerDiemCost > ALL (SELECT PerDiemCost
                         FROM Instructors
                         WHERE InstructorType = 'ORACLE');

INSTRUCTORID INSTRUCTOR PERDIEMCOST
------------ ---------- -----------
         200 UNIX               750
```

In this example, the subquery returns the values 500 and 600, and Oracle tests all rows against the following condition:

```
WHERE PerDiemCost > 500 AND PerDiemCost > 600
```

In order for a row to be returned, it must test true against both conditions. Only instructor 200 satisfies this condition.

An easy shorthand enables you to remember the distinction between ANY and ALL. If a query condition is >ANY, each row in the result set is greater than the lowest value returned. When a query is >ALL, each row in the result set is greater than the highest value returned.

Working with multi-column subqueries

In Oracle, subqueries can return only one column for consideration. That is, if you pass multiple columns out of a subquery, you receive a syntax error. For example:

```
SELECT InstructorID, InstructorType, PerDiemCost
FROM Instructors
WHERE InstructorType, PerDiemCost =(SELECT InstructorType,
                                      PerDiemCost
                                    FROM Instructors
                                   WHERE InstructorID = 110);

WHERE InstructorType, PerDiemCost =(SELECT InstructorType,
                      *
ERROR at line 3:
ORA-00920: invalid relational operator
```

However, in some situations, you may want to base a WHERE condition on more than one column from a subquery. To do so, you have two options: You can perform either a pairwise or a nonpairwise comparison.

The difference between these two types of comparison is in how they treat the two columns. In a pairwise comparison, Oracle applies the single column return rule by combining the two columns together and comparing the combined value against all rows considered by the outer query. This is achieved syntactically by placing parentheses around the two columns. Consider the previous example using a pairwise comparison. If you want to find all instructors with the same per diem and instructor type as instructor 110, the query looks like this:

```
SELECT InstructorID, InstructorType, PerDiemCost
FROM Instructors
WHERE (InstructorType, PerDiemCost) =(SELECT InstructorType,
                                      PerDiemCost
                                      FROM Instructors
                                      WHERE InstructorID =
                                       110);
```

```
INSTRUCTORID INSTRUCTOR PERDIEMCOST
------------ ---------- -----------
         300 ORACLE             500
         110 ORACLE             500
```

In this query, InstructorType and PerDiemCost are considered a single value. All other rows in the table are compared against this combined value. If you do not want instructor 110 in the result set, you have to eliminate her using the <> operator in a second WHERE condition.

A nonpairwise condition is one that treats each column as a separate WHERE condition, each with its own subquery. If you were to rewrite the previous example as a nonpairwise comparison, you get a very different comparison depending on how you link the paired elements. For example, the following query returns the same result set as the pairwise example:

```
SELECT InstructorID, InstructorType, PerDiemCost
FROM Instructors
WHERE InstructorType =(SELECT InstructorType
                       FROM Instructors
                       WHERE InstructorID = 110)

AND

      PerDiemCost = (SELECT PerDiemCost
                     FROM Instructors
                     WHERE InstructorID = 110);
```

However, if you use the OR operator rather than the AND operator, you get a very different result:

```
SELECT InstructorID, InstructorType, PerDiemCost
FROM Instructors
WHERE InstructorType =(SELECT InstructorType
                       FROM Instructors
                       WHERE InstructorID = 110)
OR

      PerDiemCost = (SELECT PerDiemCost
                      FROM Instructors
                     WHERE InstructorID = 110);

INSTRUCTORID INSTRUCTOR PERDIEMCOST
------------ ---------- -----------
         300 ORACLE             500
         100 ORACLE             600
         110 ORACLE             500
```

In this case, a row is returned if either condition returns TRUE. That is, this query returns all Oracle instructors and any non-Oracle instructors that charge $500 per day. No non-Oracle instructors charge $500; however, instructor 100 shows up because, as an Oracle instructor, he satisfies the first condition (which is all that is needed for the row to be returned).

Working with inline views

An inline view is treated like a view, but it is not created using the CREATE VIEW statement. Instead, an inline view is a subquery in the FROM clause that gets treated like a view object. To use an inline view, you must first give the subquery an alias name. This alias enables you to reference the result of the subquery as if it were a literal object (hence, the term *view*). Here is a simple example of using an inline view to show each class and its total cost:

```
SELECT s.ClassID, s.StartDate, a.totalcost
FROM ScheduledClasses s, (SELECT InstructorID, (PerDiemCost +
                          NVL(PerDiemExpenses, 0)) * 5 AS
totalcost
                          FROM Instructors) a
WHERE s.InstructorID = a.InstructorID;

  CLASSID STARTDATE TOTALCOST
--------- --------- ---------
       50 06-JAN-01      4000
       51 13-JAN-01      5000
       53 14-FEB-01      3500
```

When this query is executed, Oracle executes the subquery first and generates a result set. It gives the result set the name of "a". For the rest of the process, Oracle is able to treat the output as if it were a literal object — as with any other table or view. The alias used in the subquery becomes a valid column name in the outer query, and the values in the result set are treated as literal column values.

Inline views are useful when dealing with derived data. In the discussion of the GROUP BY statement in Chapter 3, "Using Single- and Multi-Row Functions," a problem was identified: You want to return each instructor, the city he or she is based in, and the highest per diem for all instructors in that city. To solve this problem, the only legal query was this:

```
SELECT InstructorID, City, MAX(PerDiemCost) AS "Highest Cost"
FROM Instructors
GROUP BY InstructorID, City;
```

However, this query does not return the required results because it groups MAX(PerDiemCost) by the unique occurrence of both InstructorID and City. You want the function to group only by City, but to do so violates the GROUP BY function, by having nonaggregate columns in the SELECT list that were not included in the GROUP BY clause. The problem here is that you want to treat MAX(PerDiemCost) as a literal,

but you are forced to calculate the value in the query. This problem can be solved by the use of an inline view because when you generate the aggregate value in a subquery, you can treat it as a literal value in the outer query:

```
SELECT i.InstructorID, i.City, a.Highcost
FROM Instructors i, (SELECT City, Max(PerDiemCost) AS highcost
                     FROM Instructors
                     GROUP BY City) a
WHERE i.city = a.city;

INSTRUCTORID CITY                           HIGHCOST
------------ ------------------------------ ---------
         100 New York                             600
         110 New York                             600
         450 New York                             600
         410 New York                             600
         200 Palo Alto                            750
         300 Toronto                              500
         310 Toronto                              500
         210 Toronto                              500
```

In this query, the subquery calculates the MAX(PerDiemCost) for each city and aliases that value to Highcost. The outer query is then able to treat that column as a literal value and returns it in a join just as it does with any other table or view. In order for this to work, there must be a common column between a table referenced in the outer query and a column in the inline view. This is required in order to establish an appropriate join condition.

Another use of inline views is possible. When an inline view is the only object in the FROM clause, it is possible to place an ORDER BY clause in the view. This enables you to preorder data before returning it. This is most useful in conjunction with the ROWID function introduced in Chapter 2. By combining the two, it is possible to generate the TOPn values.

For example, if you want to know the top five instructors in terms of per diem cost, you can use the following query:

```
SELECT InstructorID, PerDiemCost
FROM (SELECT InstructorID, PerDiemCost
      FROM Instructors
      ORDER BY PerDiemCost DESC)
WHERE ROWNUM <=5;

INSTRUCTORID PERDIEMCOST
------------ -----------
         200         750
         100         600
         300         500
         110         500
         310         450
```

This is the only condition where an ORDER BY clause is accepted in a subquery. In all other instances, the presence of an ORDER BY clause returns a syntax error.

Working with correlated subqueries

Another type of subquery available in SQL is the correlated subquery. A correlated subquery is a subquery that requires some information from the outer query before it can process the inner query. In order to write a correlated subquery, you must alias both the table source in the FROM clause for both the inner and outer table objects, and then create a join condition in a WHERE clause in the inner query that binds the inner and outer query together. This syntax looks like the following:

```
SELECT <select list>
FROM Table outer
WHERE colx = (SELECT colx
              FROM table inner
              WHERE inner.column = outer.column)
```

These subqueries are used most often when you want to base a condition on a result that is tied to another aspect of each row. Consider the following example. You want to retrieve a list of all instructors who have a higher per diem than the average for their instructor type. In order to solve this, you must first find the instructor type for each employee and then find the average for that type. The subquery looks like this:

```
SELECT InstructorID, PerDiemCost
FROM Instructors outer
WHERE PerDiemCost > (SELECT AVG(PerDiemCost)
                     FROM Instructors inner
                     WHERE inner.InstructorType =
                           outer.InstructorType);

INSTRUCTORID PERDIEMCOST
------------ -----------
         100         600
         200         750
         450         450
```

When this query is executed, each row is retrieved by the outer query, and the InstructorType value for that row is passed into the inner query. The inner query uses that value to evaluate the AVG(PerDiemCost) based on the InstructorType that was passed in. The subquery then returns this value to the outer query, and the row is either accepted or rejected based on the WHERE clause condition. Oracle then fetches the next row and repeats the process until no more rows remain. In this example, the subquery is processed eight times to return the result set, because eight instructors are listed in the table.

Using the EXISTS and NOT EXISTS operators

Correlated subqueries are often used in conjunction with the EXISTS and NOT EXISTS operators. These operators are used to test for the existence or lack of existence of a correlated value in the same or another table. These operators work by passing a value in the correlated subquery and testing to see whether a value is returned. Suppose, for example, you want a list of all instructors who are or have been scheduled to teach a course. You can use the following query:

```
SELECT FirstName, LastName
FROM Instructors i
WHERE EXISTS (SELECT 1
              FROM ScheduledClasses s
              WHERE i.InstructorID = s.InstructorID);

FIRSTNAME                      LASTNAME
------------------------------ ---------------------------
David                          Ungar
Kyle                           Jamieson
Lisa                           Cross
```

In this example, each InstructorID is passed into the subquery and tested. If there is a match, the subquery returns the literal value 1. If there is no match, the subquery returns a NULL. The EXISTS operator simply tests to see if any value comes back. If a value is returned, the row that corresponds to the InstructorID passed into the subquery is returned by the outer query. If a NULL is returned, the row corresponding to the InstructorID is discarded.

It is a good practice to return a literal value with the subquery rather than actually looking up a value for performance reasons. The EXISTS operator does not care what value is returned; it cares only that a value is returned. Therefore, it is easier for Oracle to pass back a declared literal rather than go through the processing involved in retrieving an actual value from the table. The end result, however, is the same whether you use a literal or return an actual value from the subquery table.

You can perform the opposite operation by using the NOT EXISTS operator. The following example returns all instructors who are not currently on the schedule:

```
SELECT FirstName, LastName
FROM Instructors i
WHERE NOT EXISTS (SELECT 1
                  FROM ScheduledClasses s
                  WHERE i.InstructorID = s.InstructorID);

FIRSTNAME                      LASTNAME
------------------------------ ------------------------------
Michael                        Harrison
Susan                          Keele
Geoff                          Williams
Lana                           Chiu
Adele                          LaPoint
```

In this case, the procedure is the same; however, rows are returned in the outer query only if the subquery returns a NULL.

Performance issues with correlated subqueries

For the exam, you are expected to know how correlated subqueries work. In the real world, however, you may want to find alternatives to correlated subqueries. By using joins and inline views, it is often possible to return the same result set in a more efficient manner. Consider the first correlated subquery example. For each row, the InstructorType is passed into the subquery, and the AVG(PerDiemCost) for that instructor type is passed out. However, this calculation is performed once for every row. Three instructors have the instructor type "ORACLE", and this means the AVG(PerDiemCost) for an Oracle instructor is recalculated three times, even though the value does not change. The same is true for all of the other instructor types. Even in this small table, the effect is noticeable, but when you have a table with hundreds of thousands of rows and thousands of different values, the cost of this type of subquery is very high. Consider this query when you use an inline view rather than a correlated subquery:

```
SELECT InstructorID, PerDiemCost
FROM Instructors i, (SELECT InstructorType, AVG(PerDiemCost)
                     AS avgsal
                     FROM Instructors
                     GROUP by InstructorType) a
WHERE i.InstructorType = a.InstructorType AND i.PerDiemCost >
      a.avgsal ;

INSTRUCTORID PERDIEMCOST
------------ -----------
         100         600
         450         450
         200         750
```

In this example, the result set is the same as the one generated by the correlated subquery, but, unlike the correlated subquery, the AVG(PerDiemCost) is calculated only once for each instructor type. Again, the difference is not noticeable in a table this small but would be a significant in a very large table.

Working with SET Operators

Up to this point in the chapter, we have been writing single queries that returned a single result set drawn from multiple tables. Set operators work slightly differently. Rather than joining tables in a single query, set operators join the output of multiple queries into a single result set. The basic syntax for a set operator is as follows:

```
SELECT <select list>
FROM table
[WHERE]
```

```
{UNION | UNION ALL | INTERSECT | MINUS}
SELECT <select list>
FROM table
[WHERE]
```

Each query must be syntactically correct and complete. That is, the query must have a SELECT and FROM clause and may contain other clauses such as a WHERE clause. It is also possible to use an ORDER BY statement with a set operator (although only one query can contain the statement).

Because the set operators link the results sets from the various queries into a single result set, each query must have a SELECT list that returns the same number of columns with the same datatypes. If the queries do not agree in the number of columns, or if the datatypes for each column do not match, Oracle returns an error. You can return a literal, if necessary, in one of the queries, as long as the literal value matches the datatype of the corresponding column in the other query. If you want to use column aliases, you must place them in the first query. When Oracle processes a set of queries, it uses the column names of the first query for the entire result set.

There are four set operators in Oracle: UNION, UNION ALL, INTERSECT, and MINUS. Each operator dictates how the result sets of the two columns are combined. This chapter examines each operator in detail.

Using the UNION operator

The UNION operator is the most common set operator. It and the UNION ALL operator are also the only ANSI SQL set operators. The UNION operator is used to unite the result sets of two queries into a single query. If a duplicate row is returned between the two queries, the UNION operator returns only one row. For example, the following query returns a list of all instructors and students, and includes a literal value that identifies which table the row is derived from:

```
SELECT FirstName, LastName,  'Instructor' AS Title
FROM Instructors
UNION
SELECT FirstName, LastName, 'Student'
FROM Students;

FIRSTNAME          LASTNAME         TITLE
------------------ ---------------- -------------
Adele              LaPoint          Instructor
Chris              Patterson        Student
Davey              Jones            Student
David              Ungar            Instructor
Geoff              Williams         Instructor
Gordon             Jones            Student
Jane               Massey           Student
John               Hee              Student
```

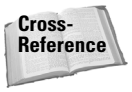
```
John            Smith           Student
Kyle            Jamieson        Instructor
Lana            Chiu            Instructor
Lisa            Cross           Instructor
Michael         Harrison        Instructor
Mike            Hogan           Student
Roxanne         Holland         Student
Sue             Colter          Student
Susan           Andrew          Student
Susan           Keele           Instructor
Trevor          Smith           Student
```

Notice in the result set that the rows have been ordered by the first column value. Oracle has ordered the result to detect any duplicate rows. This example contains no duplicate rows so all rows are returned. If you want to change the order of the result set, you can include an ORDER BY clause in the last SELECT statement. For example:

```
SELECT FirstName, LastName,  'Instructor' AS Title
FROM Instructors
UNION
SELECT FirstName, LastName, 'Student'
FROM Students
ORDER BY Title;

FIRSTNAME       LASTNAME        TITLE
--------------- --------------- ----------
Adele           LaPoint         Instructor
David           Ungar           Instructor
Geoff           Williams        Instructor
Kyle            Jamieson        Instructor
Lana            Chiu            Instructor
Lisa            Cross           Instructor
Michael         Harrison        Instructor
Susan           Keele           Instructor
Chris           Patterson       Student
Davey           Jones           Student
Gordon          Jones           Student
Jane            Massey          Student
John            Hee             Student
John            Smith           Student
Mike            Hogan           Student
Roxanne         Holland         Student
Sue             Colter          Student
Susan           Andrew          Student
Trevor          Smith           Student
```

Notice that the ORDER BY clause in the second query references a column from the first query. As previously mentioned, the resulting columns receive their names from the first query, and those column names must be the ones referenced when ordering columns in the query.

In order to see the full functionality of the operator, you need two tables that contain duplicate rows.



This table can be created using the enrollementhistory.sql script on the CD included with this book.

For the purpose of comparison, assume another table stores archival registration information called Enroll_Hist, which looks like this:

```
SQL> DESC Enroll_Hist;
 Name                                      Null?    Type
 ----------------------------------------- -------- --------------
 CLASS_NUM                                          NUMBER(38)
 STUDENT_ID                                         NUMBER(38)
 GRADE                                              CHAR(4)
 COURSEDATE                                         DATE
```

This table contains values similar to those found in the ClassEnrollment table. Each of the tables includes entries for students 1002 and 1005 taking course 50. As well, the entry for student 1001 in class 46 was accidentally duplicated in the Enroll_Hist table. Consider the result of the following query:

```
SELECT class_num, student_id, grade
FROM   Enroll_Hist
UNION
SELECT ClassID, StudentNumber, Grade
FROM   ClassEnrollment
ORDER BY student_id;

CLASS_NUM STUDENT_ID GRADE
--------- ---------- -----
       46        970 A
       45        990 C
       46       1001 A
       50       1001 B
       48       1002 C
       50       1002 A
       51       1003
       53       1003
       51       1004 A
       50       1005 F
       51       1008 A
```

In the result set for this query, notice that Oracle returns only one row for each of these students. The duplicate values of student 1002 and 1005 in course 50 between the two tables are removed, and the duplication of student 1001 in class 46 is also removed. Students 1001, 1002, and 1003 are returned more than once, but each occurrence has a different Class_num and grade value. The UNION operator combines the results and removes the duplicates but only where *all* elements in the result set are identical. If you wish to include the duplicates, you must use the UNION ALL operator.

Using the UNION ALL operator

The UNION ALL operator works the same as the UNION operator with one exception — the former returns duplicate rows if they exist between tables. To see this behavior, consider the previous example using the UNION ALL rather than the UNION operator:

```
SELECT class_num, student_id, grade
FROM   Enroll_Hist
UNION ALL
SELECT ClassID, StudentNumber, Grade
FROM   ClassEnrollment
ORDER BY student_id;

CLASS_NUM STUDENT_ID GRAD
--------- ---------- ----
       46        970 A
       45        990 C
       46       1001 A
       46       1001 A
       46       1001 A
       50       1001 B
       48       1002 C
       50       1002 A
       50       1002 A
       51       1003
       53       1003
       51       1004 A
       50       1005 F
       50       1005 F
       51       1008 A
```

In this example, you can see two entries for students 1002 and 1005 in class 50. One of these rows is derived from each table. In addition, both instances of the row for student 1001 in class 46 are presented in the output.

Using the INTERSECT operator

The INTERSECT operator is used when you want to find rows that are common between two tables. This operator compares the result sets and returns only rows that are identical in both tables. Consider the following example:

```
SELECT class_num, student_id, grade
FROM   Enroll_Hist
INTERSECT
SELECT ClassID, StudentNumber, Grade
FROM   ClassEnrollment
ORDER BY student_id;
```

```
CLASS_NUM STUDENT_ID GRAD
--------- ---------- ----
       50       1002 A
       50       1005 F
```

The result set for this query returns only the references for students 1002 and 1005 in class 50. These rows are the only two that match exactly between the two tables. Notice that the comparison is only made *between* the tables because the double reference to student 1001 in class 46 is not included because both occurrences occur in the same table.

Using the MINUS operator

The MINUS operator is essentially the opposite of the INTERSECT operator. This operator returns all rows from the first subquery listed *except* those rows that can be matched to the second query or are duplicated in the first query. Consider the following query:

```
SELECT class_num, student_id, grade
FROM   enroll_hist
MINUS
SELECT classid, studentnumber, grade
FROM   classenrollment

CLASS_NUM STUDENT_ID GRAD
--------- ---------- ----
       45        990 C
       46        970 A
       46       1001 A
       48       1002 C
```

This result set does not contain any of the rows from the ClassEnrollment table, and it also does not contain those rows with students 1002 and 1005. Notice also that the duplicate records in the enroll_hist table for student 1001 in class 46 have also been removed. The MINUS operator removes all duplicates, but unlike the UNION operator, it does not return nonduplicate rows from the second table.

Using Hierarchical Queries

At times, you require information to be presented in a hierarchical fashion. For example, you may want to show the organizational chart for a company or the bill of materials for a particular manufactured product. However, in an Oracle table, data is not stored in a hierarchical fashion. Rather, it is stored in a relational manner, with data in tables with relational keys linking rows together. It is, nevertheless,

possible to derive hierarchical information from a table based on a natural hierarchy in the relationship between rows. Consider, for example, the Management table that you created to test the self-joins. The table simply stores employees, their jobs, and the employee ID of their manager. However, due to the relationship between employees and managers, it is possible to re-create the hierarchy for all employees in the table. In Oracle, this is known as "walking the tree." By walking the tree, you are able to derive the entire reporting structure of the company either from the top down or from the bottom up.

This is achieved through the use of two special Oracle SQL clauses: START WITH and CONNECT BY PRIOR. These two clauses are used in conjunction with a SQL query to retrieve hierarchical results. The basic syntax for these clauses is as follows:

```
SELECT <select list>
FROM table
[WHERE              condition(s)]
[START WITH         condition(s)]
[CONNECT BY PRIOR   condition(s)]
```

The START WITH clause is used to establish where in the hierarchy you plan to start the query. You use a condition to establish where in the table you intend to start. You can use any valid conditional operator (for example, START WITH mgr IS NULL). You can even use a subquery in the START WITH clause if you want to determine the starting value based on another table or an unknown condition.

The CONNECT BY PRIOR clause dictates which direction the tree is walked (that is, whether you are going from the top to the bottom or the bottom to the top of the hierarchy). The CONNECT BY PRIOR clause also establishes the relationship that defines the hierarchy. The order in which you present the condition in this clause determines whether you walk from the top down or from the bottom up. In a hierarchy, all levels can be described in terms of a parent and child relationship. An object higher up in the hierarchy can be said to be a *parent* to those objects under it. The object directly under the parent is said to be the *child*. A child can also be a parent to another object under it in the hierarchy. When you present the condition in the CONNECT BY PRIOR statement, the keyword PRIOR indicates that Oracle should look for a parent row. When you place the child before the parent key, you walk up from the bottom to the top. When you place the parent before the child value, you walk from the top to the bottom.

To see how this works, consider this condition:

```
CONNECT BY PRIOR ID = Manager
```

In this clause, you instruct Oracle to take the value in the ID column (starting with the row referenced in the START WITH clause) and match it in the Manager column for the rest of the table. This ID is then seen as the parent for those rows that

match. It then takes the ID values for each of these child rows and looks in the Manager column to see if they are parents to other rows. It continues this process until it gets to the bottom of the tree.

If you reverse this condition, you walk from the bottom to the top:

```
CONNECT BY PRIOR Manager = ID.
```

In this case, Oracle takes the value in the Manager column of the starting row and finds the ID of the manager; it then repeats this for each row until it gets to the top of the tree structure. In both examples, the starting point is determined by the START WITH condition.

Putting these two elements together, you can see in the following example a query that returns the entire hierarchy of the Management table starting at the top and walking down to the bottom:

```
SELECT ID, Name, Manager
FROM Management
CONNECT BY PRIOR ID = Manager
START WITH Manager IS NULL;

        ID NAME               MANAGER
---------- --------------- ---------
         1 Jane
         2 Bob                     1
         4 Rich                    2
         8 Joan                    4
         5 Ellen                   2
         9 Pat                     5
        12 Stu                     9
        10 Fred                    5
        13 Arthur                 10
         3 Susan                   1
         6 Jim                     3
         7 Sam                     3
        11 Joe                     7
```

In this output, Jane is the first row returned because she has a NULL in the Manager column and is, therefore, the START WITH value. Under her are Bob and Susan. The result set first takes Bob and returns those under him. It then returns Susan and all of the people under her. It may seem that the order is a bit odd, but Oracle is presenting the starting point and all values under it in each of the branches. It returns Bob and Jane and all of the branches off of Bob, and then returns Susan and the branches off of Susan.

You can also use the PRIOR command in the SELECT clause to enable you to see the parent value. For example:

```
SELECT name || ' is managed by ' || PRIOR name
    AS ORG_CHART
FROM Management
CONNECT BY PRIOR id = manager
START WITH manager IS NULL;

ORG_CHART
---------------------------------------------
Jane is managed by
Bob is managed by Jane
Rich is managed by Bob
Joan is managed by Rich
Ellen is managed by Bob
Pat is managed by Ellen
Stu is managed by Pat
Fred is managed by Ellen
Arthur is managed by Fred
Susan is managed by Jane
Jim is managed by Susan
Sam is managed by Susan
Joe is managed by Sam
```

You can also present this information in reverse order by reversing the arguments in the CONNECT PRIOR statement. In this example, the query returns the reporting structure from Arthur, up through his superiors, to the top of the tree:

```
SELECT name || ' is a Manager for ' || PRIOR name
    AS ORG_CHART
FROM Management
CONNECT BY PRIOR manager = id
START WITH Name = 'Arthur';

ORG_CHART
-----------------------------------------------
Arthur is a Manager for
Fred is a Manager for Arthur
Ellen is a Manager for Fred
Bob is a Manager for Ellen
Jane is a Manager for Bob
```

Because you have listed the child first in the CONNECT BY PRIOR statement, it walks backwards. Oracle takes the Manager value starting with Arthur and finds the ID for that Manager value. It then finds the Manager value for Arthur's manager and continues until it does not find any more manager values (with Jane).

Using the LEVEL pseudo-column

To make the elements of this output more readable, Oracle also includes a pseudo-column that shows the level in the tree that a particular row occupies. The name of the pseudo-column is LEVEL. The column does not actually exist and its values are not stored in any table. Rather, it is generated when the hierarchical query is generated. For example:

```
SELECT name || ' is managed by ' || PRIOR name
    AS ORG_CHART, LEVEL
FROM Management
CONNECT BY PRIOR id = manager
START WITH manager IS NULL;

ORG_CHART                                          LEVEL
-------------------------------------------- ---------
Jane is managed by                                   1
Bob is managed by Jane                               2
Rich is managed by Bob                               3
Joan is managed by Rich                              4
Ellen is managed by Bob                              3
Pat is managed by Ellen                              4
Stu is managed by Pat                                5
Fred is managed by Ellen                             4
Arthur is managed by Fred                            5
Susan is managed by Jane                             2
Jim is managed by Susan                              3
Sam is managed by Susan                              3
Joe is managed by Sam                                4
```

In this query, all values with the same LEVEL value occupy the same level down the tree from the starting position.

You can also use this pseudo-column in conjunction with the LPAD function to format your output to show the hierarchy graphically.

The LPAD function is described in Chapter 3.

Consider the following:

```
SELECT LPAD(' ', 2* LEVEL -2) || NAME AS Report_structure,
LEVEL, position
FROM Management
CONNECT BY PRIOR id = manager
START WITH manager IS NULL;

REPORT_STRUCTURE                  LEVEL POSITION
------------------------------ --------- --------------
```

```
Jane                                    1 President
  Bob                                   2 Vice-President
    Rich                                3 Manager
      Joan                              4 Analyst
    Ellen                               3 Manager
      Pat                               4 Supervisor
        Stu                             5 Accountant
      Fred                              4 Supervisor
        Arthur                          5 Operator
  Susan                                 2 Vice-President
    Jim                                 3 Manager
    Sam                                 3 Manager
      Joe                               4 Developer
```

In this example, the LPAD function places two blank spaces at the beginning of the Name value for each level above 1 (2 * 1 -2 = 0, 2*2 -2 = 2, etc). This indents all rows so that values of the same level are on the same line and each branch on the tree is visibly identifiable.

Limiting rows in the hierarchy

At times you want only certain values returned. You may want to remove a reference to an individual, or you may want to remove an entire branch of the tree. You can do either, and that which is eliminated is dictated by where you put your limiting condition.

If you want to eliminate a single value, you can simply include a WHERE clause in the query. For example, this query eliminates only the reference to Bob:

```
SELECT LPAD(' ', 2* LEVEL -2) || NAME AS Report_structure,
LEVEL, position
FROM Management
WHERE Name <> 'Bob'
CONNECT BY PRIOR id = manager
START WITH manager IS NULL;

REPORT_STRUCTURE                LEVEL POSITION
------------------------------ --------- ---------------
Jane                                1 President
    Rich                            3 Manager
      Joan                          4 Analyst
    Ellen                           3 Manager
      Pat                           4 Supervisor
        Stu                         5 Accountant
      Fred                          4 Supervisor
        Arthur                      5 Operator
  Susan                             2 Vice-President
    Jim                             3 Manager
    Sam                             3 Manager
      Joe                           4 Developer
```

The output does not include Bob, but the rest of the tree under Bob is maintained (starting with Rich and Ellen). If you want to remove Bob and the entire branch of the tree under him, you must place the condition in the CONNECT BY PRIOR clauses, joining it with an AND operator:

```
SELECT LPAD(' ', 2* LEVEL -2) || NAME AS Report_structure,
LEVEL, position
FROM Management
CONNECT BY PRIOR id = manager AND name <> 'Bob'
START WITH manager IS NULL;

REPORT_STRUCTURE                LEVEL POSITION
------------------------------ --------- ---------------
Jane                                1 President
  Susan                             2 Vice-President
    Jim                             3 Manager
    Sam                             3 Manager
      Joe                           4 Developer
```

In the output to this query, only Susan and the branch of the tree under Susan is presented. Everyone under Bob has been removed.

Using Substitution Variables

Objective

✦ Produce queries that require an input variable

In all of the examples so far, the output of the queries has been static; that is, the values have been coded into the queries, and every time the query is executed (provided the underlying data does not change), the same values are returned. At times, however, you may not want this rigid behavior. For example, the following query always returns Oracle instructors:

```
SELECT InstructorID, FirstName, LastName
FROM Instructors
WHERE InstructorType = 'Oracle';
```

But what if you want to determine the type of instructor at runtime? To do so, you need a *runtime variable*. A variable is an object that can hold another value. Runtime variables are variables whose value is not determined until the code is actually executed. Standard SQL does not contain any runtime variables; however, such variables are available in SQL*Plus. SQL*Plus commands are used by the SQL*Plus editor. When a script is executed, SQL*Plus recognizes the variable and prompts the user to supply a value. When the user supplies a value, SQL*Plus substitutes this value for the runtime variable and executes the statement. The runtime variable in SQL*Plus is identified by the ampersand (&) and a name. To see the use of a runtime variable, consider the following statement and execution:

```
SQL> SELECT InstructorID, FirstName, LastName
  2  FROM Instructors
  3  WHERE InstructorType = '&Type';
Enter value for type: UNIX
old   3: WHERE InstructorType = '&Type'
new   3: WHERE InstructorType = 'UNIX'

INSTRUCTORID FIRSTNAME                      LASTNAME
------------ ------------------------------ --------------------
         310 Susan                          Keele
         200 Lisa                           Cross
         210 Geoff                          Williams
```

In this query, the WHERE clause contains a runtime variable: &type. When it is identified, SQL*Plus raises the prompt: "Enter value for <var>:" where <var> is the runtime variable without the ampersand.

By default, SQL*Plus returns the extra two lines of information showing the old and new values for the variable. You can turn this functionality off using the SET VERIFY OFF command.

Notice that the previous query contained single quotes around the variable name. If the runtime variable is for a date or a character string, it is necessary to place the single quotes around the variable; otherwise, the user has to include them in the variable value. If the user does not include single quotes, the statement returns a syntax error.

A single query can contain multiple runtime variables. Consider the execution the following statements:

```
SQL> SET VERIFY OFF
SQL> SELECT &col, City
  2  FROM locations
  3  WHERE City = '&city';
Enter value for col: LocationID
Enter value for city: Toronto

LOCATIONID CITY
---------- ------------------------------
       300 Toronto
```

This statement contains two runtime variables, and each is collected in order before the query is executed. It is possible to rerun this query with a completely different result set.

Dealing with repeating values

One problem with using the ampersand in queries is that SQL*Plus prompts for a value every time it encounters an ampersand, even when it has already received a value for a variable of the same name. Consider the following:

```
SQL> SELECT &Colx, &Coly
  2  FROM Instructors
  3  WHERE &Colx > &cond
  4  ORDER BY &Colx;
Enter value for colx: PerDiemCost
Enter value for coly: LastName
Enter value for colx: PerDiemCost
Enter value for cond: 400
Enter value for colx: PerDiemCost

PERDIEMCOST LASTNAME
----------- ------------------------------
        450 Keele
        450 LaPoint
        500 Harrison
        500 Jamieson
        600 Ungar
        750 Cross
```

In this example, the value &Colx is repeated three times, and SQL*Plus prompts for each occurrence, even though you may want all three to be the same value. This behavior can be overridden by using double ampersands (&&). When SQL*Plus encounters &&, it prompts for a value the first time and then reuses that value for each subsequent occurrence. The last occurrence of the variable must use a single ampersand. If you were to rewrite the previous query with double ampersands, the outcome would look like this:

```
SQL> SELECT &&Colx, &Coly
  2  FROM Instructors
  3  WHERE &&Colx > &cond
  4  ORDER BY &Colx;
Enter value for colx: PerDiemCost
Enter value for coly: LastName
Enter value for cond: 400

PERDIEMCOST LASTNAME
----------- ------------------------------
        450 Keele
        450 LaPoint
        500 Harrison
        500 Jamieson
        600 Ungar
        750 Cross
```

In this example, you are prompted only for the Colx variable once, and that value is reused through the query.

Using the ACCEPT command

In addition to the ampersand, you can also use the ACCEPT SQL*Plus command to create a runtime variable in your scripts. The ACCEPT command differs from a standard runtime variable in that it is defined before the query executes and can be reused in the query. The syntax for the ACCEPT command is as follows:

```
ACCEPT variable [datatype] [format] [PROMPT] text
```

This statement must be executed before the query is run. When you create the variable in the ACCEPT statement, you do need to include the ampersand next to the variable name, but you must use the ampersand when you reference the variable in your SQL statement. For example, if you were to rewrite the previous example using an ACCEPT statement, it would look like this:

```
SQL> ACCEPT Colx PROMPT 'Please enter column name: '
Please enter column name: PerDiemCost
SQL> ACCEPT cond PROMPT 'Please enter lowest salary value: '
Please enter lowest salary value: 400
SQL>  SELECT InstructorID, LastName, &Colx
  2   FROM Instructors
  3   WHERE &Colx > &cond
  4   ORDER BY &Colx;

INSTRUCTORID LASTNAME                       PERDIEMCOST
------------ ------------------------------ -----------
         310 Keele                                  450
         450 LaPoint                                450
         300 Harrison                               500
         110 Jamieson                               500
         100 Ungar                                  600
         200 Cross                                  750
```

In this example, the values for &Colx and &cond are captured by the ACCEPT statement. You are prompted for the values when you execute the ACCEPT statement, and you are not prompted again for a value.

One final point to be aware of when using SQL*Plus runtime variables is that they are cached with the query. When you reuse a variable in the same session, rather than prompting for a value, SQL*Plus sometimes uses the value in the cache.

You can override this behavior by undefining the variable. You do so using another SQL*Plus command — UNDEFINE. For example, if you want to clear the value for Colx in the last query, you simply run this command:

```
UNDEFINE Colx
```

and the value for Colx is removed from the cache.

The ROWID pseudo-column

One other SQL element is the ROWID pseudo-column. The ROWID pseudo-column represents the specific row location for data on the physical disk. This value exists for every row and changes only if data is moved on the physical disk.

To retrieve the ROWID, you simply reference it as if it were any other column in the tables. For example:

```
SELECT ROWID, LocationID, City
FROM Locations;

ROWID              LOCATIONID CITY
------------------ ---------- ------------------------------
AAADA6AAHAAAABiAAA        100 New York
AAADA6AAHAAAABiAAB        200 San Francisco
AAADA6AAHAAAABiAAC        300 Toronto
```

The ROWID value contains all of the information that Oracle needs to locate a row and can be used to speed data retrieval. Remember, however, that a particular ROWID value is not permanently bound to a row, and if the row changes its physical location (for example, you move a table to another tablespace), the ROWID value for that row also changes.

Key Point Summary

It is possible to write a single query that returns data from multiple tables. This can be done using either a join or a subquery.

- ✦ **Joins:** In order to join multiple columns together, you must have a common join condition. A join condition is a method of relating rows in one table with rows in another table.
  - The join condition is often on the primary key/foreign key relationship between tables.
  - Once a join condition has been established, you can reference any column from the joined tables.
  - If you have ambiguous columns (that is, columns with the same name in different tables) you must reference the table that the column is derived from (tablename.columname).
  - You can alias a table in the FROM clause and use the alias value as shorthand when identifying the table source for a column.

If you include more than one table in the FROM clause and do not reference it with a condition in the WHERE clause, you get the Cartesian product of the tables. The Cartesian product is simply every value in the first table joined to every value in the second table. To avoid a Cartesian product, you need one less condition in the WHERE clause than the number of tables in the FROM clause. There are four types of joins: equijoins, nonequijoins, outer joins, and self-joins.

Equijoins

- Join two or more tables based on a condition of equality.
- Only return rows that are common to both tables in the join condition.
- Can include all other SQL clauses (for example GROUP BY, HAVING, ORDER BY).

Nonequijoins

- Follow all the rules of equijoins.
- Are created with any join condition that is not based on equality including BETWEEN.
- Can reference more than one column in a table as part of the join condition.

Outer joins

- Are used when you want to return all values from one table, even if there is no corresponding value in the other table.
- Return a value from the joined table where possible and return a NULL when no join can be made.
- Are indicated using the outer join operator (+).
- This operator is placed in the WHERE condition on the column that has NULLs added to it when a match cannot be made.

Self-joins

- Occur when you need to join a table to itself.
- Are created by adding the same table twice in the FROM clause with different alias names.
- Occur most often when a table has a self-referential foreign key (for example, the Manager column referring to the employee ID column in the same table).
- Can be inner or outer joins.
- Follow all the rules of the other join forms.

- ✦ **Subqueries:** Another method of writing a query that references more than one table is to use a subquery. A subquery is an independent statement nested inside a parent query.
  - All subqueries must be enclosed in parentheses.
  - A subquery must return data from only one column unless you are using a pairwise comparison.
  - A pairwise comparison binds two columns together and treats them as one column.
  - You can nest subqueries. The innermost query always executes first.
  - Subqueries can return only one row unless you are using an operator that can accept multiple rows (IN, ANY, and ALL).
  - Some subqueries can be correlated. Correlated subqueries are achieved by aliasing the tables in the FROM clause of both queries and making a join condition between the inner and outer table aliases in a WHERE clause in the subquery.
  - Subqueries can also appear in the FROM clause.
  - Subqueries in the FROM clause are executed before the outer query and can be treated as temporary table objects by the outer query.

Aside from joins and subqueries, you can also use set operators to derive data from more than one table by combining the results of several queries. When you use a set operator, the queries must all return the same number of columns with the same datatypes. There are four set operators:

- ✦ The UNION operator returns the result set from all queries but eliminates duplicate values both between tables and within a single table.
- ✦ The UNION ALL operator returns the results from all queries including duplicate values.
- ✦ The INTERSECT operator returns only those values that exist in the result sets for both queries.
- ✦ The MINUS operator returns all data from the first query except for values that exist in the second query.

Runtime variables are available in SQL*Plus.

- ✦ You can define a runtime variable with either the ampersand (&) or the ACCEPT statement.
- ✦ When you use a runtime variable to represent character or date data, you should place single quotes around the variable in the query.
- ✦ When you have a repeating runtime variable, SQL*Plus prompts you for a value each time it occurs unless you use double ampersands (&&).
- ✦ The ACCEPT command is used to define a variable before the statement is executed.

✦ ✦ ✦

STUDY GUIDE

In this chapter, you looked at some of the advanced elements in SQL. You learned how to used joins and subqueries to access multiple tables in a single query, and you have used the set operators to combine result sets from multiple queries. In addition, you have learned how to write hierarchical queries and use runtime variables to control execution of a statement at runtime. The exam focuses mostly on joins, subqueries, and runtime variables; however, all of this information is important and may come up, not just in the exam, but in the real world. Now you can test your understanding with the sample questions, scenarios, and exercises that follow.

Assessment Questions

1. You issue the following query against the Instructors table:

```
SQL> SELECT InstructorID, City, ClassID, StartDate,
2    StartDate + DaysDuration AS EndDate
3    FROM Instructors, ScheduledClasses
4    WHERE instructorid = instructorID
5    AND country = 'USA'
6    ORDER BY
```

Which line in this statement causes an error?

A. 2

B. 4

C. 5

D. 6

2. In which of the following situations do you use an outer join?

A. You need to return the class ID and start date from the ScheduledClasses table and the city from the Locations table.

B. You need to join information on one table to a range of values on another table.

C. You have a self-referential foreign key on your Employees table between the Empid and Manager columns, and you want to return all employees and their managers including the CEO, who does not have a Manager value listed.

D. You need to join more than two tables.

3. You execute the following query:

```
SELECT InstructorID, PerDiemCost
FROM Instructors
WHERE PerDiemCost >
     (SELECT AVG(PerDiemCost)
      FROM Instructors
      WHERE City =
           (SELECT City
            FROM Locations
             WHERE LocationID =
                (SELECT LocationID
                 FROM ScheduledClasses
                 WHERE startdate = '01-JAN-01')
```

Which query is executed first?

A. SELECT city

B. SELECT LocationID

C. SELECT InstructorID, PerDiemCost

D. SELECT AVG(PerDiemCost)

4. You issue the following query:

```
SQL> SELECT s.ClassID, c.Coursename, l.LocationName
2    FROM ScheduledClasses s, Courses c, Locations l
3    WHERE s.coursenumber = c.coursenumber AND
4    s.locationID = c.locationID
5    AND city = &city
6    ORDER BY s.classid
```

When you are prompted, you enter the string "Toronto". However, you receive the following error:

```
ORA-00904: invalid column name
```

What is the most likely reason for the error?

A. You did not place single quotes around the runtime variable.

B. You did not alias the City column in line 5.

C. You used aliases in the SELECT list before they were established in the FROM clause.

D. The City column in line 5 is not referenced in the SELECT list.

5. You want to issue a hierarchical query that lists all of the managers beginning with a clerk named Stu and working up to the president whose name is Jane. If the tables have a column for employee IDs called id and a column for manager IDs called manager, which CONNECT BY PRIOR and START WITH combination do you use to write this query?

A.

```
CONNECT BY PRIOR manager = id
START WITH name = 'JANE'
```

B.

```
CONNECT BY PRIOR id = manager
START WITH 'JANE'
```

C.

```
CONNECT BY PRIOR id = manager
START WITH    'Stu'
```

D.

```
CONNECT BY PRIOR manager = id
START WITH    'Stu'
```

6. Which operator do you use to return all of the rows from one query except where those rows are returned in a second query?

A. UNION

B. UNION ALL

C. MINUS

D. INTERSECT

7. You issue the following query:

```
SELECT FirstName, LastName, PerDiemCost
FROM Instructors
WHERE PerDiemCost > (SELECT PerDiemCost
                     FROM Instructors
                     WHERE InstructorID = 10);
```

If no instructors have an InstructorID of 10, what is returned by the outer query?

A. No rows

B. A syntax error

C. All instructors with a PerDiemCost greater than 0

D. All instructors with a PerDiemCost greater than NULL

8. You issue the following query:

```
SQL> SELECT i.InstructorID, i.PerDiemCost
2           i.InstructorType, a.avgcost
3    FROM Instructors i, (SELECT InstructorType,
4                                AVG(PerDiemCost)
5                          FROM Instructors
6                          GROUP BY InstructorType) a
7    WHERE i.instructortype = a.instructortype
```

Which line in this statement will cause an error?

   A. 2

   B. 3

   C. 4

   D. 6

   E. 7

9. Which of the following clauses can contain a subquery? (Choose all that apply.)

   A. GROUP BY

   B. WHERE

   C. HAVING

   D. FROM

10. Which of the following is not true of the ROWID function?

   A. Each ROWID value is unique.

   B. The ROWID value contains everything Oracle requires to find a particular row.

   C. The ROWID value is permanently bound to a particular row.

   D. The ROWID value is not stored in the table.

Scenarios

1. You have been asked to create a report that draws information from several tables. The report has three separate components, and you require separate queries.

   A. The first query requires you to list all instructors whose combined cost and expenses are greater than the average for the city they are located in. How do you write this query? Is there more than one way to write it? Which way is preferable?

B. In the second report, you are asked to list each class, the name of the course, and the center in which the class is running. You are also asked to include all centers, even those without classes. Is it possible to return this information? If so, how do you find this information?

C. In the third query, you need to compare the current enrollment figures to past enrollments on an archive table. You have to write a query that lists all of the students who exist on both tables. How do you find this information? You are also concerned that some entries were copied to the archive table but not removed from the enrollment tables. How do you eliminate these rows?

2. You have been asked to write a script that most users will run in SQL*Plus. The script must return information about orders in your company's order-entry system. These scripts need to be flexible and efficient.

A. You have a table called ORDERS, and you want to create a script that returns all orders within x number of days of the current date. You want the user to supply the value for x at runtime. How do you write this query to allow the current date to change? How do you enable the user to choose the value of x?

B. Your products are manufactured from other components. Some of the these other components are also manufactured by your company of other subcomponents. Each product ID is stored in a column called prod_id, and each component is listed in a column called comp_id. How can you write a query that shows each product and the bill of materials (that is, the subcomponents) that make up that component?

Lab Exercises

Lab 4-1 Using equijoins

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.

2. Write a query that joins instructor and location names based on city.

3. How many rows are returned? _____

4. How many instructors are included in the Instructors table? _____

5. Why is an instructor not listed?

6. Rewrite the previous query to include the ClassID column from the ScheduledClasses table.

7. How many rows are returned? _____

8. Why is this number of rows returned?

Lab 4-2 Using self-joins and outer joins

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.
2. Using the Management table, write a query with an output that looks like this:

```
ORG_CHART
------------------------------------------------------------
--
Bob is a Vice-President and reports to Jane
  . . .
```

If you have not run the selfjoin.sql script to create the Management table, you must run it before completing this lab.

3. Does this query return all employees in the table?
4. Rewrite the query so that it returns all employees in the table. If any employees have NULL for their manager's name, replace the NULL with the string "no one".

Lab 4-3 Using subqueries

1. Open SQL*Plus and connect to your instance using the Student account with password oracle
2. Write a query using a subquery that returns all instructors by ID and name whose total cost and expense are higher than the average for their city.
3. Rewrite this query to return that average value in the SELECT list of the outer query with the name AVG_COST.

Lab 4-4 Using hierarchical queries

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.
2. Using the Management table, write a query that lists a hierarchical tree starting with Bob and walking downward. The result set should include every persons' name, their level within the hierarchy, and their position. Each name should be indented three spaces for each level.
3. Rewrite the query so that the branch starting from Rich is removed.
4. Rewrite the first query in this section so that you can determine your starting point at runtime.

Answers to Chapter Questions

Chapter Pre-Test

1. The three syntax elements required to write a join are a SELECT list, a FROM clause, and a WHERE clause that contains the join condition.
2. Without the WHERE clause, Oracle generates a Cartesian product and joins all rows with all other rows.
3. Yes, you can join a table to itself. In order to do so, you must have two columns that can be related to each other. You must reference the table name twice in the FROM clause and give each reference a different alias name.
4. You want to use an outer join when you want all rows from one table in a join returned, even if they cannot be joined to a row on the other table in the join.
5. A subquery can return only one column unless you are performing a pairwise comparison. In a pairwise comparison, you join multiple columns together and return the unique occurrence of several columns.
6. If any subquery returns a NULL, the output of entire query is NULL.
7. A subquery can be placed in most clauses, including in the SELECT list, in the FROM clause, the WHERE clause, the HAVING clause, and the ORDER BY clause. The only clause that cannot contain a subquery is the GROUP BY clause. Subqueries are most often found in the WHERE and HAVING clauses.
8. You want to use a correlated subquery when the inner query requires a value from the outer query to evaluate. For example, a subquery that returns the AVG(PerDiemCost) based on the InstructorType of the instructor in the outer query. In the real world, you might consider using an inline view or a join rather than a subquery.
9. You can control the execution of a SQL statement at runtime by using a runtime variable. Without runtime variables, the statement must be hard-coded at design time.
10. Oracle's implementation of SQL does not have any provision for runtime variables. To use runtime variables, you have to use runtime variables available in other platforms such as SQL*Plus or PL/SQL.

Assessment Questions

1. **B —** The error in this query occurs at line 4. It is generated because you have an ambiguous column reference. An InstructorId column is included in both the Instructors and ScheduledClasses tables, and Oracle is unable to resolve which column is associated with which table. In order to correct this problem, you have to specify the table source for each column. For example:

   ```
   WHERE instructor.instructorid =
   scheudledclasses.instructorid
   ```

   You can also use column aliases to avoid this ambiguity between tables.

2. **C —** You use an outer join in situation C because you want to return all employees including the CEO. The fact that the table is self-referential only suggests a self-join; however, the CEO most likely has a NULL in the Manager column, and you have to use an outer join to have him returned with the rest of the employees. Situation A simply recommends an equijoin, and situation B requires an nonequijoin.
3. **B —** When Oracle executes a statement with nested subqueries, it always executes the innermost query first. This query passes its results to the next query and so on until it reaches the outermost query. It is the outermost query that returns a result set.
4. **A —** The correct answer is A. If you do not put single quotes around the runtime variable, the user is required to enter the quotes manually as part of the supplied value for the variable. If the user does not add the quotes, no quotes are placed around the string. When Oracle encounters character data that is not enclosed in single quotes, it tries to interpret the value as an object name. This explains the error message. You do not have to alias unambiguous columns, and it is possible to reference a column elsewhere in the query without referencing it in the SELECT list. It is also possible to use alias values in the SELECT list before they are declared. This is because Oracle parses the whole query before it attempts to reconcile the objects in the query.
5. **D —** If you want to walk up the tree, you must use the CONNECT BY PRIOR values where child = parent (in this case, where ID = Manger). This takes the Manager column from the starting row and finds that value in the ID column. It then takes that new ID look for its manager and continues until it encounters a null manager. You must put Stu in the START WITH clause as you are walking up from him.
6. **C —** You use the MINUS operator to return all rows from one query except where duplicate rows are found in a second query. The UNION operator returns all rows from both queries minus duplicates. The UNION ALL operator returns all rows from both queries including duplicates. The INTERSECT operator returns only those rows that exist in both queries.
7. **A —** If any subquery returns a NULL (which occurs when the subquery references a nonexistent instructor), the entire query returns a NULL. When the output of a query is NULL, Oracle returns the "no rows returned" message.
8. **A —** The error occurs at line 2. The SELECT list at line 2 references a nonexistent column. The source of the error is actually at line 5. When you place a subquery in the FROM clause of a query, it generates a temporary table with column names equal to the column name in the SELECT list of the subquery. In order for this query to work, you have to alias the AVG(PerdiemCost) as avgcost in the subquery. Otherwise, Oracle is unable to find the column and returns a "column not found" error.
9. **B, C, D —** A subquery can be contained in all SELECT clauses except for the GROUP BY clause. It can be used in the WHERE clause and HAVING clause to return a value for the condition. It can also be used in the FROM clause to create a temporary table source.

10. **C —** The ROWID function is not bound to the row. If the row is physically moved (such as through an import or export operation), it is assigned a new ROWID value. The value is unique and is used by Oracle to locate the row. ROWID is a pseudo-column and is not stored in the table itself.

Scenarios

1. **A.** This query can be written using either a correlated subquery or an inline view. The inline view (that is, a subquery in the FROM clause) is preferable because it calculates each value once rather than once per row.

   B. In order to find this information, it may seem that you are going to have to write an outer join, because you must find those centers that do not have classes. The problem is an outer join is only possible between two tables, and this query requires joining three tables. It is possible to retrieve the required data, but it requires the use of the UNION operator. The query looks something like this:

```
SELECT s.Classid, c.coursename, l.LocationName
FROM ScheduledClasses s ,courses c,  Locations l
WHERE s.coursenumber = c.coursenumber AND s.locationid =
l.locationid
UNION
SELECT s.classid, null, l.locationName
FROM locations l, scheduledclasses s
WHERE l.locationid = s.locationid(+)
AND s.classid is null;
```

   The first query returns all rows except for the centers that do not have classes. The second query uses an outer join to find the other centers. Notice that you need to add an extra column in the SELECT list of the second query using the NULL operator. With the UNION operator, both queries must agree in the number of columns.

   C. In order to find all rows including duplicates, you can use the UNION ALL operator. To determine whether rows are duplicated between tables, rather than in the same table, however, you can include some kind of flag in each query, like this:

```
SELECT class_num, student_id, grade, 'EH' AS Flag
FROM   enroll_hist
UNION
SELECT classid, studentnumber, grade, 'CE'
FROM   classenrollment
ORDER BY student_id;
```

   This flag value indicates whether a duplicate row came from different tables or the same table.

2. **A.** To write this query, you can use a combination of system functions and runtime variables. To allow the script to always use the current date, you simply have to use the SYSDATE function in the WHERE clause of the query. Using this function saves the user from having to enter the current date. To provide the value of x, you have to use a runtime variable. Because you are working in SQL*Plus, you can use either the ampersand or the ACCEPT statement in your scripts.

   B. To produce a bill of materials, you have to build a hierarchical query. In this case, you start with the finished product and walk down the tree through all of the components and subcomponents that make up that product. The query might look something like this:

```
SELECT prod_id, prod_name, comp_id
FROM manufacturing
CONNECT by Prior prod_id = compid
Start with prod_id = x
```

   If you have subcomponents in the list with child comp_id values, they are included in the tree under the product.

Lab Exercises

Lab 4-1 Using equijoins

2.

```
SELECT i.InstructorID, l.LocationName
FROM Instructors i, Locations l
WHERE i.City = l.City;
```

3. Seven rows are returned.

4. There are eight instructors.

5. One of the instructors is from Palo Alto, and there is not a location in that city.

6.

```
SELECT i.InstructorId, l.LocationName, s.ClassID
FROM Instructors i, Locations l, ScheduledClasses s
WHERE i.InstructorID = s.InstructorID AND l.LocationID =
s.LocationID;
```

7. Three rows are returned.

8. Only three classes have instructors assigned to them.

Lab 4-2 Using self-joins and outer joins

2.

```
SELECT e.Name || ' is a ' || e.position || ' and reports to '
|| m.name Org_Chart
FROM management e, management m
WHERE e.manager = m.id;
```

3. No, Jane is not included on the list.

4.

```
SELECT e.Name || ' is a ' || e.position || ' and reports to '
|| NVL(m.name, 'No one' AS Org_Chart
FROM management e, management m
WHERE e.manager = m.id(+);
```

Lab 4-3 Using subqueries

2.

```
SELECT InstructorID, FirstName, LastName
FROM Instructors o
WHERE (PerDiemCost + PerDiemExpenses) > (SELECT
                                         AVG(PerDiemCost +
                                         PerDiemExpenses)
                                         FROM Instructors i
                                         WHERE o.city =
                                               i.city);
```

3.

```
SELECT i.InstructorID, i.FirstName, i.LastName, a.AVG_Cost
 FROM Instructors i, (SELECT City, AVG(PerDiemCost +
                       PerDiemExpenses) AS AVG_COST
                       FROM Instructors
                       GROUP by City) a
 WHERE i.city = a.city AND (i.PerDiemCost +
i.PerDiemExpenses) > AVG_COST;
```

Lab 4-4 Using hierarchical queries

2.

```
 SELECT LPAD(' ' , LEVEL * 3 -3) || Name AS Name,
 LEVEL, Position
 FROM Management
 START WITH name = 'Bob'
 CONNECT BY PRIOR id = Manager;
```

To make this script more readable, you should shorten the Name column to a column 30 characters wide with the following command:

```
COLUMN Name FORMAT A30
```

3.

```
SELECT LPAD(' ' , LEVEL * 3 -3) || Name AS Name,
LEVEL, Position
FROM Management
START WITH name = 'Bob'
CONNECT BY PRIOR id = Manager AND name <> 'Rich';
```

4.

```
SELECT LPAD(' ' , LEVEL * 3 -3) || Name AS Name,
LEVEL, Position
FROM Management
START WITH name = '&Name'
CONNECT BY PRIOR id = Manager;
```

CHAPTER 5

Adding, Updating, and Deleting Data

EXAM OBJECTIVES

- Manipulating Data
  - Describe each DML statement
  - Insert rows into a table
  - Update rows in a table
  - Delete rows from a table
  - Control transactions

CHAPTER PRE-TEST

1. How many rows can be inserted in a single INSERT statement?
2. How is data modified once it has been added to a table?
3. Can data in one table be modified based on data from another table? If so, how is this done?
4. What happens if you do not want to supply values for all columns when you insert a row into a table?
5. Is there an easy way to change the date of a record to today's date?
6. Can all rows in a table be deleted with a single statement?
7. What is a transaction? How do you control transactions?
8. What happens to transactions when you combine DML statements with other kinds of SQL statements?
9. What do you see when you issue a SELECT statement on a row that has been locked by another user?
10. What happens when you try to modify this row?

✦ Answers to these questions can be found at the end of the chapter. ✦

This chapter examines one of the key language elements of SQL in Oracle — the Data Manipulation Language (DML). Unlike the SQL elements discussed in Chapters 2 and 3, DML is not used to query existing data; rather, it's used to add, remove, or manipulate data within tables. The three main commands in DML are INSERT, UPDATE, and DELETE. For the exam, you will be expected to understand thoroughly how each command works. This chapter begins by examining each command individually and then looks at how you can use subqueries to modify data from one table based on data in another. Next, we examine how Oracle organizes these statements using transactions. Sometimes a single logical operation requires multiple DML statements. The transaction helps us group these statements and ensure that they execute as a logical unit of work. Finally, we look at how Oracle uses locking to maintain transactional consistency and what happens when users encounter locked data.

As a Relational Database Management System (RDBMS), the Oracle engine controls all access to the database. This means that when you want to place or modify data in a table, you must instruct Oracle to add the data for you. This is achieved through the INSERT, UPDATE, and DELETE statements. Because storing and manipulating data is the key function of an Oracle database, you must have a full understanding of how to issue these statements in order to begin working with Oracle.

DML Statements: Inserting Data into Tables

- Insert rows into a table

In an active production database, new information is constantly being added. You may have new employees join the company, new products, new customers, or new Locations. In order to add data to one or more tables, you must issue the correct DML statement. When you want to add data to a table, you use the INSERT statement. The basic syntax for an INSERT statement is as follows:

```
INSERT INTO table [(column, [column . . . ])]
VALUES  (value [, value . . . ]);
```

You can insert information into only one table at a time using the INSERT statement. Thus, you can include only one table name in the INSERT statement. The VALUES keyword is followed by a list of values that are entered into the table in the same order as the columns for that table. When you are inserting character data or date data into a table, you must enclose this information in single quotes. Numeric data does not require single quotes. When you do not use single quotes with character or date data, Oracle attempts to interpret the data as a schema object, and you receive the following error:

```
ERROR at line 2:
ORA-00984: column not allowed here
```

Remember that string data is stored in the case in which you enter it, and that it must be accessed in that case when performing subsequent queries. Date information must be in the format of your National Language Support (NLS) date format. If it is not, you must use the TO_DATE function (see Chapter 2, "Retrieving Data Using Basic SQL Statements"). Date issues are discussed later in this chapter in "Inserting Values Using Additional Language Elements."

Here is a simple INSERT statement that adds a new training center into the Locations table:

```
INSERT INTO Locations (LocationID, LocationName, Address1,
Address2, City, State, Country, PostalCode, Telephone, Fax,
Contact, Description)
VALUES (
  123,
   'Tor_Downtown',
   '123 Main St.',
   'Suite 2101','Toronto',
   'ON',
   'Canada',
   '4165551234',
   '4165553342',
   'Jane Simpson',
   'New Toronto location');
```

This statement inserts the data into the Location table in the order that it appears in the values list. You must ensure that the order of the data in the VALUES list matches the order of the columns in the table. If you are not sure of the order of columns in the table, you can use the SQL* Plus DESC[RIBE] command to find this information, as follows:

```
SQL> DESC Instructors;
 Name                      Null?              Type
 ------------------  --------------   --------------------
 INSTRUCTORID         NOT NULL         NUMBER(38)
 SALUTATION                            CHAR(4)
 LASTNAME             NOT NULL         VARCHAR2(30)
 FIRSTNAME            NOT NULL         VARCHAR2(30)
 MIDDLEINITIAL                         CHAR(5)
 ADDRESS1                              VARCHAR2(50)
 ADDRESS2                              VARCHAR2(50)
 CITY                                  VARCHAR2(30)
 STATE                                 CHAR(2)
 COUNTRY                               VARCHAR2(30)
 POSTALCODE                            CHAR(10)
 OFFICEPHONE                           CHAR(10)
 HOMEPHONE                             CHAR(10)
 CELLPHONE                             CHAR(10)
```

```
 EMAIL                                     VARCHAR2(50)
 INSTRUCTORTYPE                            CHAR(10)
 PERDIEMCOST                               NUMBER(38)
 PERDIEMEXPENSES                           NUMBER(38)
 COMMENTS                                  VARCHAR2(2000)

SQL>
```

This query also assumes that you are supplying information for all columns. When the table contains more columns than there are values in the VALUES list, the statement returns an error.

Using the column list

If you are not going to insert a value into each column in the table, you must include a column list after the table name. This list specifies the column or columns that you want to insert data into, and the order in which those columns appears in the VALUES list. The following example inserts data into only a select number of columns in the Instructor table:

```
INSERT INTO Instructors (
   InstructorID,
   first_name,
   LastName,
   InstructorType)
VALUES (
   1234,
   'Stephen',
   'Giles',
   'Full Time');
```

Looking at the description for the Instructors table (see the preceding), you notice that many columns in the table are not included in the preceding INSERT statement. Also notice that in the schema, LastName appears before FirstName. When you include a column list after the table name, this list is matched with the VALUES list. Therefore, the data in the VALUES list must be in the same order as it appears in the column list. A NULL value is inserted into any column not listed in the column list. When the column has a NOT NULL constraint, you must include the column in the INSERT statement; otherwise, the statement returns an error.

Information on the NOT NULL constraint can be found in Chapter 7, "Creating and Managing Oracle Database Objects."

Except as noted in the following statement, it is important to remember that you can affect only one row with each INSERT statement. For example, the following statement, which tries to insert two instructors, results in the error shown:

```
INSERT INTO Instructors (InstructorID, FirstName, LastName)
VALUES (1, 'Bob', 'Smith', 2, 'Stephen', 'Giles')
```

```
ERROR at line 1:
ORA-00913: too many values
```

The only exception to the single-row insert rule is when you insert data using a subquery. This action enables you to insert multiple rows. Inserting data with subqueries is discussed later in this chapter in "Using Subqueries in DML Statements."

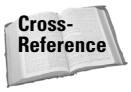
Inserting values using additional language elements

SQL contains a number of additional language features that enable you to insert values into a table. One such feature is the NULL operator, which indicates to Oracle that you intend to leave the value of a column NULL. The following example adds a new course but leaves the ReplacesCourse column null:

```
INSERT INTO Courses
VALUES (3456, 'Intermediate SQL', NULL, 'This course introduces
more advanced elements of the SQL language', 1875);
```

This statement does not return any errors because it supplies the correct number of values, and because the ReplacesCourse column allows NULL values. Be careful not to include single quotes around the NULL operator. When you include quotes, Oracle interprets it as a character string and not the NULLoperator.

If you use the NULL operator on a column that has a default, the null overrides the default. Oracle views the null operator as an explicit null. When you want to insert the default values into the column, omit the column from the insert list.



For more information on defaults, see Chapter 7.

You can also use SQL functions to generate values in INSERT statements. A function is a predefined block of code that can receive parameters and return values. Oracle SQL contains a number of built-in functions to assist you in working with data. One such function is the SYSDATE function. This function returns the current data and time. You can use this function to add the current date to a column.

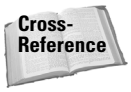
For example, if you add a row to the ClassEnrollment table with the current date in the EnrollmentDate column, you can use the following statement:

```
INSERT INTO classEnrollment (ClassID, StudentNumber, Price,
EnrollmentDate)
VALUES (1234, 454, 1875, SYSDATE);
```

As with the NULL operator, be sure not to place single quotes around the SYSDATE function; otherwise, Oracle attempts to enter it as a character string.

SYSDATE normally includes the hours and minutes when it is used during an INSERT. If the column is intended to include date and time, SYSDATE by itself works well. However, the time components can sometimes cause problems later when trying to select data from the rows using just the date component. To capture just the date component on a SELECT, INSERT, or UPDATE, include the truncate function (TRUNC) as follows: TRUNC(SYSDATE).

Finally, you can use the conversion functions (TO_CHAR, TO_DATE, TO_NUMBER) to modify data before inserting it into the table. This is especially useful when dealing with dates. As we mentioned earlier in the chapter, date data must be enclosed in single quotes. When the SQL engine interprets this data, it attempts to convert the character string into Oracle's internal date format by applying the NLS_DATE_FORMAT value against it.



For a discussion of TO_CHAR, TO_DATE, and TO_NUMBER, see Chapter 2, "Retrieving Data Using Basic SQL Statements."

If you are not sure what the NLS defaults are for your server, you can find out by querying the V$NLS_PARAMETERS view. For example, to find the NLS_DATE_FORMAT for your server, you use the following query:

```
SELECT value FROM v$nls_parameters
WHERE parameter = 'NLS_DATE_FORMAT';
```

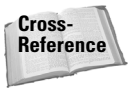
You can see the type of problem that may arise due to the NLS_DATE_FORMAT in the following example. When your NLS_DATE_FORMAT is "DD-MON-YY", the following query returns an error:

```
INSERT INTO ClassEnrollment (ClassID, StudentNumber, Price,
EnrollmentDate)
VALUES (123, 445, 1875, 'January 12, 2001')

ERROR at line 2:
ORA-01858: a non-numeric character was found where a numeric
was expected
```

The problem here is that Oracle is not able to convert the nonstandard date string into its internal date format. However, the following query will execute successfully:

```
INSERT INTO ClassEnrollment (ClassID, StudentNumber, Price,
EnrollmentDate)
VALUES (123, 445, 1875, TO_DATE('January 12, 2001', 'Month DD,
YYYY')
```

In this query, TO_DATE explicitly translates the date format for Oracle using the format information specified in the parameters of the function.



See Chapter 2, "Retrieving Data Using Basic SQL Statements," for more information on the TO_DATE conversion function.

DML Statements: Modifying Existing Data

- Update rows in a table

In an active database, data is constantly changing — shipping dates are added to orders, product inventories are modified, and client information is updated as clients move locations and change positions. To modify data after a row has been inserted into a table, you must use the UPDATE command. The basic syntax for the UPDATE statement follows:

```
UPDATE table
SET column = value [, column = value, . . . ]
[WHERE condition];
```

As you can see, it is possible to modify multiple columns with a single UPDATE statement; however, you can update only one table at a time. The WHERE condition in an UPDATE statement controls which columns are updated. An UPDATE statement updates all rows that match the WHERE condition. If the WHERE clause is omitted, all rows in the table are modified. The following example changes the address for instructor 123 in the Instructors table:

```
UPDATE Instructors
SET Address1 = '222 Green St.'
WHERE InstructorID = 123;
```

It is very important to include the WHERE clause in the UPDATE statement to get the required results. The UPDATE statement updates all rows that are included by the WHERE clause. Therefore, the following query:

```
UPDATE Instructors
SET Address1 = '222 Green St.';
```

updates the Address1 column for all of the instructors in the Instructors table. The WHERE clause is evaluated before the UPDATE takes place so only rows that satisfy the condition in the WHERE clause are updated. The condition of the WHERE clause in the UPDATE statement accepts any of the operators mentioned in Chapter 2.

It is also possible to use arithmetic operators and functions in the SET clause to modify data. For example, suppose instructor 123 added $50 to his per diem price. You can update his record with the following query:

```
UPDATE Instructors
SET PerDiemCost = PerDiemCost + 50
WHERE InstructorID = 123;
```

In this query, Oracle enters a new value in the PerDiemCost column for instructor 123 based on the outcome of the arithmetic operation.

As an example of using a function in the SET clause, the following query will change the value of the FirstName column to place a value into upper case:

```
UPDATE Instructors
SET FirstName = UPPER(Firstname)
WHERE InstructorID = 123
```

In this example, Oracle retrieves the current value for FirstName, applies the UPPER function, and sets the result as the new value in the column.

Again, without the WHERE clause, this UPDATE affects all rows in the table. This behavior, however, can sometimes be beneficial. For example, if you want to increase the total cost of all courses by 10 percent, you can use the following statement:

```
UPDATE Courses
SET RetailPrice = RetailPrice * 1.1;
```

This query sets the outcome of the arithmetic expression based on the retail price for each row in the Courses table. This operation is repeated for each row in the table.

DML Statements: Removing Data from Tables

Objective

- Delete rows from a table

In an active database, you will want to remove records at times — employees may leave, products may be discontinued, data may be moved to archival tables. To remove a row from a table, you use the DELETE command. The basic syntax for the DELETE statement is as follows:

```
DELETE [FROM] table
[WHERE condition];
```

As with the UPDATE statement, the WHERE clause is used to control which rows are affected by the DELETE statement. For example, if you wanted to delete student 4 from the Students table, you use the following statement:

```
DELETE FROM Students
WHERE StudentNumber = 4;
```

If you omit the WHERE clause, the DELETE statement removes all rows from the table. For example, the following statement removes all rows from the Students table:

```
DELETE FROM Students;
```

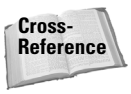
When this statement is executed, the definition for the table remains, but all data in the table is removed. Data still can be inserted into this table, but a query against the table returns the following message:

```
no rows selected
```

If you do want to remove all rows from a table, it is better to use the TRUNCATE command rather than issuing a DELETE statement without a WHERE clause because the DELETE is repeated and logged for each row, whereas the TRUNCATE executes as a single operation. The downside to the TRUNCATE command is that it cannot be rolled back.

Both the DELETE and TRUNCATE commands may fail if the table has a dependent FOREIGN KEY. In this case, you will receive the following error:

```
 ERROR at line 1:
ORA-02292: integrity constraint
(STUDENT.FK_SCHEDCLASSES_LOCATIONID) violated - child
record found.
```



For more on the TRUNCATE command, see Chapter 7, "Creating and Managing Oracle Database Objects."

As was the case with the UPDATE statement, you can use any of the WHERE operators in the DELETE statement. For example, if you want to delete all records for 1997 from the CourseEnrollment table, you can use the following statement:

```
DELETE FROM CourseEnrollment
WHERE EnrollmentDate BETWEEN '01-JAN-1997' AND '31-DEC-1997';
```

Using subqueries in DML statements

It is possible to issue DML statements based on data in other tables. This is achieved by including subqueries in the DML statements. Each type of statement treats subqueries slightly differently. In this section, we look at how to use subqueries in INSERT, UPDATE, and DELETE statements.

Using subqueries in INSERT statements

With INSERT statements, you can add a new row in a table using data derived from one or more tables. For example, suppose some of your contract instructors (with an InstructorID of 456) takes courses at your training facility, and you want to add them into the Students table. You could do so with the following statement:

```
INSERT INTO Students (StudentNumber, LastName, FirstName,
MiddleInitial, Address1, Address2, City, State, Country,
PostalCode, Homephone, WorkPhone, Email)
SELECT 999, LastName, FirstName, MiddleInitial, Address1,
Address2, City, State, Country, PostalCode, Homephone,
WorkPhone, Email
FROM Instructors
WHERE InstructorID = 456;
```

Note that when you use a subquery with the INSERT statement, you do not need the VALUES keyword. For this statement to work, the subquery must return the same number of columns listed in the INSERT list, and the datatype must match between the two columns. If the SELECT list in the subquery does not return enough values, or if the number of columns do not match, or the datatypes do not match, the INSERT returns an error.

This statement also inserts one row for each row returned by the subquery. For example, the following query adds four rows to the Students table:

```
INSERT INTO Students (StudentNumber, LastName, Firstname,
MiddleInitial, Address1, Address2, City, State, Country,
PostalCode, Homephone, WorkPhone, Email)
SELECT 999, LastName, Firstname, MiddleInitial, Address1,
Address2, City, State, Country, PostalCode, Homephone,
WorkPhone, Email
FROM Instructors
WHERE InstructorID IN (123, 234, 345, 456);
```

This is the only way to insert more than one row with a single INSERT statement.

Using subqueries in UPDATE statements

Subqueries can be used in both the SET and WHERE clauses of the UPDATE statement. This enables you to alter the values of a column in one table from values in another table (or even a different row in the same table). For example, if you want to change the price of all courses that have the same price as course 123 to the price of course 456, you can use the following statement:

```
UPDATE Courses
SET RetailPrice =
                    (SELECT RetailPrice FROM Courses
                     WHERE CourseNumber = 456)
```

```
WHERE RetailPrice =
                    (SELECT RetailPrice FROM Courses
                     WHERE CourseNumber = 123);
```

This statement evaluates each row in the table. Wherever it finds a row that has the same retail price as course 123, it changes the price to the retail price of course 456. It evaluates the subqueries first, so it also updates course 123 without affecting subsequent rows.

Using a subquery in the SET clause of an UPDATE statement returns an error when the subquery returns multiple rows. Oracle does not allow you to update the same row more than once in a single UPDATE statement, so the SET clause accepts only a single value for each column being modified.

You can also use subqueries to update multiple columns by placing more than one column in the SELECT list of the subquery. Suppose you wanted to update both the PerDiemCost and the PerDiemExpenses for instructors 3, 5, 7, and 9 to be the same as for instructor 11; you can use the following statement:

```
UPDATE Instructors
SET (PerDiemCost, PerDiemExpenses) =
              (SELECT PerDiemCost, PerDiem Expenses
                FROM Instructors WHERE InstructorID =
                10)
WHERE InstructorID in (3, 5, 7, 9);
```

Make sure to enclose the two columns to the left of the equal sign in parentheses. In order for this statement to function correctly, Oracle must treat the two columns as a *pair-wise* operation. A pair-wise operation is one in which the contents of the two columns are treated as a single value. If the parentheses are omitted, Oracle tries to resolve each column independently, and you receive the following error:

```
ERROR at line 2:
ORA-00927: missing equal sign
```

In this case, Oracle matches the subquery with the PerDiemExpenses and then looks for a value to set for the PerDiemCost column.

Using subqueries in DELETE statements

You can use subqueries in DELETE statements to remove rows from one table based on rows from a different table. Suppose, for example, that you have closed your London location, and you want to remove all classes scheduled to run in that location. You can remove these rows using the following statement, prior to removing London from the Locations table:

```
DELETE FROM ScheduledClasses
WHERE LocationID = (SELECT LocationID
                    FROM Locations
                    WHERE City = 'London');
```

As with the UPDATE statement, you can also use subqueries to reference other rows in the same tables. For example, if you want to remove all courses from the schedule that are scheduled on the same day as class 123, you can use the following statement:

```
DELETE FROM ScheduledClasses
WHERE StartDate = (SELECT StartDate
                   FROM ScheduledClasses
                   WHERE ClassID = 123);
```

Oracle evaluates the subquery first and then deletes each row that matches the date returned (including ClassID 123).

How Oracle Processes DML Statements

Exam Tip Understanding how Oracle processes DML statements is important for understanding transactional controls; however, you will not be tested directly on this section.

When a user connects to an Oracle database (for example, by starting a SQL*Plus session and providing a valid username and password for a database instance), an area of memory, called the Private Global Area (PGA), is made available to the user. The PGA stores cursor state information and the values of bind variables used, as well as other connection information. This area of memory resides on the server side in a process launched for the user when the database connection is established. This process is called the *server* process.

When an Oracle instance is started on the server to connect to an Oracle database, several other processes are also launched. These include the following:

- ✦ The DB Writer (DBWR) to write data to the disks
- ✦ Process Monitor (PMON) to monitor server processes and user connections
- ✦ Log Writer (LGWR) to write data to redo log files as transactions are processed
- ✦ System Monitor (SMON) to perform maintenance tasks on the instance
- ✦ The Checkpoint process (CKPT) to ensure that all files are synchronized, and users don't see uncommitted data

The Oracle instance also includes a shared memory structure called the System Global Area (also called the Shared Global Area) or SGA. The SGA contains a number of structures that enable Oracle to process requests received by users and return the right data. These include the following:

- ✦ The Shared Pool, which contains the Data Dictionary Cache (to hold frequently accessed database object definitions) and the Shared SQL Areas of the Library Cache (to hold the text and execution plan of recently executed SQL statements)

- ✦ The Database Buffer Cache, which holds copies of recently accessed database blocks from the data files
- ✦ The Redo Log Buffer, which holds copies of changes made to the database prior to those changes being written to the Redo Log Files

The server process, the PGA, and the SGA, along with the other processes that make up the Oracle instance, all work together when a SQL statement is sent to the Oracle server to query or update data. When a user issues any SELECT, INSERT, UPDATE, or DELETE statement, Oracle must go through several steps to process these queries. Consider the processing of the following statement:

```
UPDATE Courses
SET RetailPrice = 1900
WHERE CourseID = 101;
```

When this statement is executed, Oracle goes through the following steps. Oracle first parses the statement to make sure that it is syntactically correct. During this *parse* phase, Oracle also determines whether the objects that are referenced (in this case, the Courses table) are available for the user and whether the user has permission to access the object and perform the required task (that is, the UPDATE). It does this by locating information about the object in the Data Dictionary Cache or, if this information is not in cache, by reading the information from the Data Dictionary and placing it in the cache. By placing information about the object in the cache, it ensures that future requests for the object are performed quicker, in case other users are also referencing the Courses table. If the user does not have permissions or the object does not exist, an error is returned to the user.

When the object is located and the user has permissions, the next element of the *parse* phase is to check whether the statement has been previously executed by anyone. If so, then Oracle does not need to build the execution plan (the series of tasks to be performed to satisfy the query). It can simply keep the execution plan that was previously created and use it in the next phase of processing. If it cannot find the execution plan, indicating this is the first time the statement is being run, or the statement is no longer in the Shared SQL Areas and has been aged out, Oracle then builds the execution plan and places it in the Shared SQL Areas. Oracle then proceeds to the *execute* phase of processing.

During the *execute* phase, Oracle runs the execution plan in the Shared SQL Areas and performs whatever tasks are contained therein. This includes locating the relevant blocks of data in the Database Buffer Cache, or if they are not in the cache, the *server* process reads the datafiles where the data resides and loads the data blocks into the Database Buffer Cache within the SGA.

The server process then places a lock on the data being modified (in this case, the row containing course 101). This lock prevents other users from updating the row at the same time you are updating it. Oracle then updates a Rollback Segment and a Data Segment in another area of cache in the SGA — the Redo Log Buffer. It places the data in the row prior to the update in a rollback block and the new value in a data block.

The Rollback Segment is used for two purposes:

- **Read consistency:** Until the change is committed, any user who executes a query for the retail price of course 101 sees the price prior to the upgrade. The new value is not visible until the update is committed.
- **Recoverability:** If the system crashes before the transaction is committed, or if the user issues an explicit ROLLBACK command, the data in the Rollback Segment can be used to return the row to its initial state.

When the modifications to the data are first initiated, the transaction is assigned a unique value called a System Change Number (SCN), indicating that the change is to be synchronized with the datafile, as well as a timestamp, indicating when the change took place. The changes are processed and placed in the Redo Log Buffer to be written to the Redo Log Files by the LGWR process. The process continues until all the data has been updated and the entire query has been satisfied.

The final phase of processing is the *fetch* phase. For a SELECT statement, the *fetch* phase of processing returns the actual data to the user, and it is displayed in SQL*Plus. For an UPDATE operation, or any DML statement, the *fetch* phase simply notifies the user that the requisite number of rows has been updated.

When other statements are part of the same transaction, the same series of steps (that is, parse, execute, and fetch) take place for each statement until the user issues a COMMIT or ROLLBACK statement. When the transaction is committed or rolled back, Oracle ensures that all information in the Redo Log Buffer pertaining to the transaction is written to the Redo Log Files, in the case of a COMMIT, or the data blocks are restored to their previous state, in the case of a ROLLBACK, and removes all locks. Oracle also erases the values held in the rollback segment. This means that once a transaction is committed, it is not longer possible to roll it back.

Controlling Transactions

Objective

- Control transactions

As you have seen, each DML statement is a single operation that affects one or more rows. However, at times you need to link more than one statement into a logical unit. Consider the steps involved in a bank transfer. Because SQL does not include a transfer command, transferring $1,000 between two bank accounts requires two UPDATE statements:

```
SQL>UPDATE Accounts
2  SET balance = balance - 1000
3  WHERE AccountName = 'SourceAccount';

SQL>UPDATE Accounts
2  SET balance = balance + 1000
3  WHERE AccountName = 'TargetAccount';
```

Physically these are two independent statements, but logically they are a single operation. If the first statement executes, but something prevents the second statement from executing, there is a problem — $1,000 is removed from the source account but is never deposited into the target account. This is also a serious financial problem because you are now missing $1,000! With a transaction, you can group these two operations into a single action. Transactions do more than just group statements together; they also ensure the logical consistency of the database and your data.

The ACID test

Transactional control in Oracle has been designed to meet the ACID test first suggested by Jim Gray and Andreas Reuter. ACID stands for "Atomicity," "Consistency," "Isolation," and "Durability," and these characteristics are discussed in the sections that follow.

Atomicity

This means that the entire transaction must commit entirely, or not at all. In the previous example, when the second statement fails, the first statement should be reversed (or *rolled back*) to maintain transactional consistency.

Consistency

Consistency means that the transaction must follow some logical rules. In the account transfer example, you are moving money from one location to another. Consistency, in this case, requires that money not be created or destroyed. It must be moved, and the amount credited must be the amount debited.

Isolation

Isolation means that while one process is processing a transaction, it must have absolute control over all of the elements it is affecting. Oracle provides a concurrent environment. This means that many users are able to access the same data at the same time. In our example, isolation prevents two different database users from updating the balance of the source account at the same time. Oracle achieves isolation through the use of *locking*. While a process has a row locked, it cannot be modified by another process. Isolation also extends to reading data. If someone were to read the balance from the target account after the first statement had executed but *before* the second statement, he or she would receive information that was not necessarily accurate. Oracle uses the Rollback Segment discussed in the previous section to maintain read isolation.

Durability

Durability means that after a transaction completes (or *commits*), it must remain that way. It is a built-in mechanism that enforces atomicity. In other words, when a system failure occurs, you must be able to return the system to the state it was in before the failure. Because changes in Oracle are made in cache, a mechanism must exist to restore lost transactions. This is achieved through the redo logs.

In addition, when a failure occurs before the transaction is committed, a mechanism must exist that automatically rolls back any part of the transaction that completed before the failure. Oracle's recovery mechanisms provide this type of durability.

Transaction control statements

A transaction is implicitly begun when the first executable statement is executed. It is not terminated until it is either committed or rolled back.

An executable statement includes the INSERT, UPDATE, and DELETE commands, as well as DDL and DCL statements (see Chapters 6 and 7). SELECT statements are not considered part of a transaction. However, if you use the SELECT FOR UPDATE statement, Oracle locks the rows being selected.

In Oracle SQL, you can explicitly control the end of a transaction using the COMMIT and ROLLBACK commands. When you issue a COMMIT statement, the changes made by the DML statement become permanent. When you issue a ROLLBACK, the changes are reversed. For example, to complete the bank transfer example, you submit the following statements:

```
SQL>UPDATE Accounts
2  SET balance = balance - 100
3  WHERE AccountName = 'SourceAccount';

SQL>UPDATE Accounts
2  SET balance = balance + 100
3  WHERE AccountName = 'TargetAccount';

SQL>COMMIT;
```

The transaction is started implicitly when the first UPDATE statement is issued and ends when the COMMIT statement is executed. Every DML statement between these two events is considered part of the same transaction.

The advantage of this system is that is possible to verify an operation before committing it. As you saw in the previous section, when you query the table that you have modified, Oracle displays the results of all DML statements within the transaction. These changes are not visible to other users until you either commit or roll back the transaction. This provides isolation. You can issue several DML statements, realize that you have made a mistake, and undo your actions without other users being aware of the change. To undo the changes, you simply issue a ROLLBACK statement.

Consider the following:

```
UPDATE Accounts
SET balance = balance - 1000
WHERE AccountName = 'SourceAccount';
```

```
UPDATE Accounts
SET balance = balance + 100
WHERE AccountName = 'TargetAccount';
```

An error occurs in this case. One thousand dollars was withdrawn from the source account, but only one hundred dollars was deposited in the target account. If you discover this mistake, you can undo it by issuing the following statement:

```
SQL>ROLLBACK;
```

When the ROLLBACK statement is issued, it undoes all of the statements that have been executed since the transaction began. In the preceding example, when the ROLLBACK is issued, it takes the $100 out of the target account and places the $1,000 back into the source account.

A ROLLBACK can be issued at any point as long as the transaction is still open. If you commit a transaction, it is no longer possible to issue a ROLLBACK statement.

At times, however, you may not want to roll back the entire transaction. For instance, you may want to verify each step of a multi-step transaction and roll back individual steps. This level of transactional control can be achieved with the use of *savepoints*. Savepoints are named markers within the transaction that can be used as the target for a rollback. You set savepoints with the following syntax:

```
SAVEPOINT <name>;
```

After the savepoint is set, you can roll back to that point by including the TO operator and the name of the savepoint. Consider the following example:

```
SQL> DELETE FROM Courses
2    WHERE CourseID = 10;
SQL> SAVEPOINT ID10del;
Savepoint Created
SQL> UPDATE Instructors
2    SET city = 'New York'
3    WHERE InstructorID = 4;
SQL> SAVEPOINT update4;
SQL> UPDATE Instructors
2    SET PerDiemExpense = 6000
3    WHERE PerDiemExpense = 500;
SQL> ROLLBACK TO update4
```

In this example, a transaction is started when the DELETE statement is executed. It then processes the two UPDATE statements, setting a savepoint before each UPDATE. With the final UPDATE, you intended to raise only the per diem expenses to $600, and you want to roll back only this last statement. When the ROLLBACK TO update4 statement is executed, only the second UPDATE statement is rolled back. The other two statements (the DELETE and the first UPDATE) remain unchanged, and the transaction remains open. It remains this way until the transaction is either committed or rolled back using the COMMIT or ROLLBACK (without the TO parameter) statement.

After you issue a ROLLBACK TO statement, it is also possible to add subsequent statements to the transaction. For example, you could correct the final statement this way:

```
SQL> DELETE FROM Courses
2    WHERE CourseID = 10;
SQL> SAVEPOINT ID10del;
Savepoint Created
SQL> UPDATE Instructors
2    SET city = 'New York'
3    WHERE InstructorID = 4;
SQL> SAVEPOINT update4;
SQL> UPDATE Instructors
2    SET PerDiemExpense = 6000
3    WHERE PerDiemExpense = 500;
SQL> ROLLBACK TO update4
SQL> UPDATE Instructors
2    SET PerDiemExpense = 600
3    WHERE PerDiemExpense = 500;
SQL> COMMIT;
```

In this example, the DELETE and the first and third UPDATE statements are committed. The ROLLBACK TO statement reverses only the second UPDATE statement. All savepoints and locks are removed after the transaction is committed. When you issue a ROLLBACK statement without the TO operator, it reverses all steps in the transaction and also removes all savepoints.

One final consideration concerns transactions. Under certain conditions, transactions can be implicitly committed or rolled back. Whenever a Data Control Language (GRANT or REVOKE) or Data Definition Language (CREATE, ALTER, DROP) statement is issued, it *automatically* commits any open transactions. This is because both of these types of statements must run as their own transaction. This means that once you issue one of these statements, all savepoints are removed, and it is no longer possible to roll back any DML statement issued prior to the DCL or DDL statement.

Also, if you are using SQL*Plus as your SQL editor and you do not commit your work, SQL*Plus automatically commits any changes when you exit the program. The same is also true for any program that issues a CONNECT command to connect to an Oracle instance including, among others, Oracle Forms and Oracle Enterprise Manager. Exiting any of these programs automatically issues a DISCONNECT command that commits any active transactions before cleaning up the connection.

On the other hand, if the program were to fail for some reason (for example, a client system or the Oracle instance crashes or is in some way abnormally disconnected from the server), Oracle automatically rolls back all open statements up to the last COMMIT or ROLLBACK.

In the case of an abnormal termination of a client program (for example, the system crashes or the program is killed), the actual rollback of any open transaction is not immediate and may take some time. This is because a process on the Oracle server called Process Monitor (PMON) must ensure that the client has indeed gone away and does not want to terminate the session prematurely. This is analogous to renting an apartment: You want to make sure the landlord has verified the previous tenant is gone before the landlord tells you to move in.

Controlling concurrent operations with locking

You will remember that part of the ACID test was isolation. Isolation implies that while a transaction is being processed, no other user can manipulate the data that was being treated in the transaction. Oracle controls this isolation with the use of locks. There are two types of locks in Oracle:

- ✦ Shared locks
- ✦ Exclusive locks

Shared locks are acquired when you issue a SELECT statement. Oracle locks the table to ensure that no one modifies its structure while you are using its data, but does not place any locks on the rows being queried. Shared locks do not prevent other users from reading or modifying the data in the table — only from making changes to the table's structure using the ALTER TABLE command or from dropping the table using the DROP TABLE command. Multiple users can acquire shared locks on the same data.

Exclusive locks are acquired when you issue a DML statement, and they are acquired for each row being modified. The exclusive lock prevents other users from acquiring exclusive locks on the data you are working with as part of your transaction until you issue a COMMIT or ROLLBACK statement. This prevents two users from attempting to update the same data at the same time. When a user attempts to update data that is locked by another user, the user must wait until the lock is removed.

Exclusive locks do not prevent another user from querying the data, however; the query returns the data from the Rollback Segment (that is, values that do not reflect the outcome of any DML statement issued as part of the uncommitted transaction). Oracle returns values from the Rollback Segment because the user who has placed the exclusive lock still has the option to roll back his or her changes. Changes are visible only to other users when the transaction is committed. Up to that point, the changed data is considered only potential data and is visible only to the user who acquired the exclusive lock. This prevents unrepeatable reads.

Oracle holds exclusive locks until the transaction is committed. This means that one user can block other users who are trying to modify the same data. For performance reasons, you should try to keep transactions as short as possible. The longer a transaction, the more locks it will hold and, therefore, the more likely it is to block other users on database.

Key Point Summary

The Oracle relational database engine controls all access to the database. This engine understands only SQL and, therefore, when you want to add, modify, or remove data from tables in the database, you must use the Data Modification Language of SQL. DML is made up primarily of three statements:

- ✦ **INSERT:** The INSERT statement is used to add rows to a table.
  - An INSERT statement can add only one row at a time (with the exception of INSERTS based on the results of subqueries).
  - The values being inserted into a column are either derived from a subquery or listed in brackets as part of the VALUES clause.
  - Character and date data must be enclosed in single quotes.
  - If you intend to insert data into some but not all columns in the table, you must include a column list surrounded by parentheses after the table name in the INSERT statement that lists the names and order of the columns that you are supplying data for.
  - You can use SQL functions such as SYSDATE and TO_DATE to insert data in the VALUES clause.
- ✦ **UPDATE:** The UPDATE statement is used to modify existing rows in a table.
  - You use the WHERE clause in the UPDATE statement to determine which rows will be updated by the statement. The statement updates all rows identified in the WHERE clause.
  - If you omit the WHERE clause, all rows in the table are updated.
  - You can use arithmetic expressions and functions in the SET clause to change the value of a column based on the current value.
  - You can use subqueries in both the SET and WHERE clauses to modify data in one table based on another table or on other rows in the same table.
- ✦ **DELETE:** The DELETE statement is used to remove rows from a table.
  - Like the UPDATE statement, you use a WHERE clause to determine which rows are deleted.
  - If you omit the WHERE clause, all rows are deleted from a table.

Each DML statement can form part of a transaction. A transaction is a collection of DML statements that act as a logic unit. A transaction must complete in its entirety or not at all. Oracle implicitly begins a transaction whenever the first executable statement is executed. Once a transaction is started, it remains open until it is either committed or rolled back. You can commit a transaction with the COMMIT statement and roll it back using a ROLLBACK statement.

When you issue a ROLLBACK, all DML statements issued after the start of the transaction are undone. If you do not want to roll back an entire transaction, you can set savepoints. Savepoints are named markers in the transaction. When a savepoint is added to a transaction, it is then possible to use the TO operator with a ROLLBACK statement and roll back to that savepoint in the transaction.

To prevent users from overwriting each other's transactions, Oracle locks data. It sets either a shared lock or an exclusive lock. A shared lock ensures that no one modifies the structure of the table and blocks other users from acquiring shared locks on the same resources. An exclusive lock is acquired when you issue a DML statement. This lock prevents other users from issuing DML statements against the same rows on which you are working until you either commit or roll back your transaction. When another user attempts to query data that has an exclusive lock on it, the user sees the state of data prior to any UPDATE, INSERT, or DELETE statements until the transaction is committed. The user who is controlling the transaction is able to see any changes to the data.

✦ ✦ ✦

STUDY GUIDE

In this chapter, we looked at the various elements of the Data Modification Language of SQL. A full understanding of this part of Oracle's SQL is essential to interact fully with the database. Without writing proper DML statements, it is impossible to add, modify, and remove data from the database. Now you can test your understanding with the sample questions, scenarios, and exercises that follow.

Assessment Questions

1. You attempt to add a row to the Instructors table with the following statement:

```
SQL> INSERT INTO Instructors (InstructorID, Firstname,
2      Lastname, MiddleInitial, Address1, City, State)
3      VALUES (3, 'Bob', 'Smith', 'A', 1500 Main St.,
4      'NULL', 'IN')
```

Which line in this statement will cause an error?

A. 1

B. 2

C. 3

D. 4

2. Which of the following executable statements will end a transaction? (Choose three.)

A. ROLLBACK TO

B. DROP

C. UPDATE

D. GRANT

E. CREATE

3. You issue the following statement:

```
SQL> UPDATE Courses
2    SET RetailPrice = 1500
3    WHERE RetailPrice = 1400;
```

At the same time, another user issues a SELECT statement looking for the price of a course that has a retail price of $1,400. What value will be returned to this user?

A. 1500.

B. 1400.

C. Oracle does not return a value until the UPDATE statement is committed.

D. ORA-000000X: Current row locked by another process.

4. You execute a script with the following statements:

```
1     INSERT into Instructors (FirstName, LastName)
2     VALUES ('Bob', 'Green');
3     SAVEPOINT spA;
4     INSERT into Instructors (FirstName, LastName)
5     VALUES ('Julius', 'Black');
6     SAVEPOINT spB;
7     INSERT into Instructors (FirstName, LastName)
8     VALUES ('Jane', 'Grey');
9     SAVEPOINT spC;
10    ROLLBACK TO spA;
11    INSERT into Instructors (FirstName, LastName)
12    VALUES ('Jean', 'Brown');
13    Commit;
```

Which instructors are inserted into the table when this script is executed? (Choose all that apply.)

A. Bob Green

B. Julius Black

C. Jane Grey

D. Jean Brown

5. You attempt to modify a row to the ScheduledClass table with the following statement:

```
SQL> UPDATE ScheduledClass
2    SET StartDate = 'January 12, 2001'
3    WHERE  ClassID = 4321
```

and you receive the following error:

```
ERROR at line 2:
ORA-01858: a non-numeric character was found where a numeric
was expected
```

How can you alter this statement to avoid this error? (Choose two.)

A. Use the SYSDATE function.

B. Change the date string to match your server's NLS date format.

C. Remove the single quotes around the date in the SET clause.

D. Use the TO_DATE function with the parameters "Month DD, YYYY".

6. You want to add data to a table using SQL*Plus, but you are not sure of the names of the columns within the table or their datatypes. What is the easiest way to find the names and datatypes of a table?

A. Use the TAB[LE] SQL*Plus command.

B. Query the tab_col system table.

C. USE the SCH[EMA] SQL*Plus command.

D. Use the DESC[RIBE] SQL*Plus command.

7. You have been working in SQL*Plus for some time. You started by modifying several records and then changed your password with the ALTER USER command. You then made several more updates to the data when your session of SQL*Plus crashed. When you restart the application and reconnect, what is the state of your data?

A. All DML statements and the ALTER statement are rolled back.

B. Only statement issued after the ALTER statements are rolled back.

C. Oracle rolls back all operations up to the first DML statement.

D. The transaction is open until you issue a COMMIT.

8. You issue a DELETE statement in SQL*Plus but omit the WHERE clause. You then exit the application. You realize afterward that you did not want to delete all rows. How can you undo this delete?

A. Do nothing. Because you closed the application without committing the changes, the DELETE is automatically rolled back.

B. Restart SQL*Plus and issue a ROLLBACK statement. The transaction is still open because you did not explicitly commit it.

C. You cannot undo the deletion.

D. You have to log in as the SYSTEM account and issue a ROLLBACK statement from this account.

9. One of your instructors has enrolled in a course, and you want to add her to the Students table. Her instructor ID is 14. You issue the following statement:

```
SQL> INSERT INTO Students (StudentNumber, Firstname,
2    Lastname, Address1, City, State, PostalCode)
3    SELECT 456, FirstName, LastName, Address1,
4    City, State FROM Instructors
5    WHERE InstructorID = 14;
```

When you execute the statement, it returns an error. Which line contains the error?

A. 2

B. 3

C. 4

D. 5

10. You attempt to update two columns based on the outcome of a subquery using the following query:

```
SQL> UPDATE Students
2    SET HomePhone, Email =
3    (SELECT HomePhone, Email
4    FROM Instructors WHERE InstructorID = 10)
5    WHERE StudentNumber = 345;
```

When you execute the query, you receive an error. How can you edit this query to remove the error?

A. Place brackets around the two columns in the SET clause.

B. Place double quotes around the two columns in the SET clause.

C. Remove the brackets from around the subquery.

D. Place single quotes around the InstructorID value.

Scenarios

1. You are working on an order-entry system for your company. Fifty order-entry people in a call center will use this system. These order entry people will access a common set of customer and product records, and they may change these records after they are entered. Because these users will be working while on the phone with your customers, speed is an issue.

   A. Can one user update customer records while other users are viewing them? What happens when two users try to update the same record?

   B. How can you modify the INSERT statement used by the operators so they don't have to enter the current date in the OrderDate column?

   C. What steps can you take to speed up the update process?

   D. Sometimes, your application will be called upon to update the prices of all your products to reflect changes in costs. How many UPDATE statements are required to update the prices? How can you change the value of the unitprice column in the Products table to reflect a 15 percent increase?

2. You have been asked to move some data from an older Orders table into an archival Order_History table. The data to be moved is any order placed before January 1, 1998.

   The layout of the Orders table follows:

```
CREATE TABLE Orders
(OrderID number(10),
 Order_Date char(25),
 CustomerID number(10),
 SalespersonID number(10),
 Ship_date char(25));
```

The layout of the Order_History table follows:

```
CREATE TABLE Order_History
(Order_num number(15),
 Customer_no number(15)
 Orderdate date,
 shipdate date,
 Sales_no number(15),
 Archive_date date)
```

The Archive_date column in the Order_History table holds the date that a record was moved to this table.

A. How can you write the statement to migrate the data? How do you deal with the extra column in Order_History?

B. The date data in the Orders table is stored as text in the form of "Month dd, yyyy". Your NLS date format is "DD-MON-YY". How do you deal with the date as you move the data?

C. You want to verify that all rows are moved in the operation before you commit the INSERT. Can you do this? If so, how do you verify that all rows are transferred?

D. What statement do you use to delete the records from Orders after they have been moved to the Order_History table?

Lab Exercises

Lab 5-1 Inserting data

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.
2. Add the new instructors listed in Table 5-1 to the Instructors table.

Table 5-1
New Instructors List

InstructorID	*FirstName*	*LastName*	*Address1*	*City*	*State*
101	Jessica	Jones	4 Apple Rd.	Flint	MI
102	Fred	Baker	123 Center St.	Patton	NJ
103	Suzie	Smith	34 North rd.	Toronto	ON

3. What is inserted into all of the other columns after you complete your inserts?

Lab 5-2 Updating data

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.
2. Use the information in Table 5-2 to modify the instructors you added in the last exercise.

Table 5-2
Modification List

InstructorID	*InstructorType*	*PerDiemCost*	*PerDiemExpenses*
101	FT	500	400
102	FT	650	400
103	FT	500	400

3. What is the fewest number of UPDATE statements required to make these modifications?
4. Increase the PerDiemExpense for all instructors with an expense of less than $500 by 25 percent.

Lab 5-3 Testing transactional controls

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.
2. Issue the following statement:

```
UPDATE Courses
SET RetailPrice = 1000
WHERE CourseNumber = 110;
```

3. Place a savepoint called "sp_1" after the UPDATE.
4. Open a second session of SQL*Plus and log in using the Student account.
5. Issue the following query:

```
SELECT RetailPrice from Courses
WHERE CourseNumber = 110;
```

6. What is the retail price returned by the query? Record your results: _____.
7. Go back to the first session and run the same query.
8. What is the retail price returned by this query?

9. In the first session of SQL*Plus, issue a second UPDATE statement:

```
UPDATE Courses
SET RetailPrice = 0
WHERE CourseNumber = 201;
```

10. In the second session, run the following query:

```
SELECT CourseNumber, RetailPrice
FROM Courses
WHERE CourseNumber IN (110, 201);
```

11. What are the two retail prices returned? Record the results in Table 5-3.

Table 5-3
Testing Transactional Control

CourseID	*RetailPrice*

12. In the first SQL*Plus session, roll back to sp_1.
13. In the second session of SQL*Plus, rerun the last query.
14. Has anything changed in the retail prices?
15. In the first session of SQL*Plus, issue a COMMIT statement.
16. In the second session, run the query a third time.
17. What changes have been made to the retail price? Record the answers in Table 5-4.

Table 5-4
Testing Transactional Control

CourseID	*RetailPrice*

Lab 5-4 Deleting data

1. Open SQL*Plus and connect to your instance using the Student account with password oracle.
2. Delete the instructors that you added in Lab 5-1.

Answers to Chapter Questions

Chapter Pre-Test

1. Only one row can be inserted in a singe INSERT statement. When you include data for more than one row in the VALUES column, you receive an error. The only exception to this is when you use a subquery. Oracle executes an INSERT statement for each row returned by the subquery.
2. After data has been added to a table, it can be modified with an UPDATE statement.
3. Data from one table can be modified based on data in another table. This is accomplished by adding a subquery in the SET clause of the UPDATE statement. The value returned by the subquery is the new value of the column referenced in the SET clause.
4. When you do not want to supply values for all columns, you have two options. You can include an INSERT list and list only those columns for which you are supplying values, or you can use the NULL operator in the VALUES list to indicate that a particular column should be left NULL.
5. When you want to change a date record to the current date, you can use the SYSDATE function in the SET clause of an UPDATE statement.
6. It is possible to delete all rows in a table with a single DELETE statement. To do so, simply omit a WHERE clause. When you want to dump all of the data in a table, you can also use the TRUNCATE TABLE statement (see Chapter 6). Only the DELETE statement can be rolled back.
7. A transaction is one or more DML statements grouped into a logical unit of work. Transactions are treated as a single unit of work that must be completed in its entirety or not at all. You can control transactions using the COMMIT, SAVEPOINT, and ROLLBACK statements.
8. If you combine DML statements with SELECT statements, there is no effect. However, if you issue a DDL (such as CREATE TABLE or TRUNCATE TABLE) or DCL statement (such as GRANT or REVOKE) together with DML statements, these statements commit any open transactions. This means that once you issue a DCL or DDL statement, you cannot roll back any prior DML statements. They are committed prior to the execution of the DCL or DDL statement.
9. When you issue a SELECT statement against a row that has been locked by another user, you see the data as it appeared before the user made any modifications to the row.
10. When you attempt to modify a row that has been locked by another user, you are blocked and have to wait until that user either commits or rolls back the transaction.

Assessment Questions

1. **C** — The error in this statement occurs at line 3. The problem is that the value for the address does not have single quotes around it. All character and date data must be enclosed in single quotes. The single quotes around the NULL operator in line 4 does not create an error. It does, however, insert the value NULL as a text string rather than just leaving the column NULL. For more information, see the "DML Statements: Modifying Existing Data" section in this chapter.
2. **B, D, E** — A transaction can be ended in many ways. The obvious one is to execute either a COMMIT or ROLLBACK command. However, when you issue a Data Definition Language (DROP and CREATE) or Data Control Language statement (GRANT), you also commit any open transactions. The ROLLBACK TO rolls back only to a savepoint. It does not terminate the transaction. For more information, see the "Transaction control statements" section in this chapter.
3. **B** — Until a user commits a transaction, the modification made cannot be seen by other users. Therefore, the old value of 1400 will be visible to other users. Exclusive locks do not block readers so there is no "ORA-000000X" error. For more information, see the "Controlling concurrent operations with locking" section in this chapter.
4. **A, D** — When the ROLLBACK TO statement is issued, it reverses any DML statements between the rollback statement and the savepoint. This removes Julius Black and Jane Grey. It does not, however, commit the transaction, so the addition of Jean Brown is still considered part of the same transaction and is made permanent when the COMMIT statement commits the transaction. For more information, see the "Transaction control statements" section in this chapter.
5. **B, D** — The source of the error is that Oracle is unable to convert the date string into its internal date format. Editing the text string will solve this problem, but it requires knowing your NLS date format. With the TO_DATE function, you do not have to know the date format; the function performs the translation for you. The SYSDATE function does not help in this case because it sets only the current date. Date data requires single quotes in both INSERT and UPDATE statements. Removing the single quotes only adds a second error to the statement. For more information, see the "DML Statements: Modifying Existing Data" section in this chapter.
6. **D** — You can use the SQL*Plus DESC[RIBE] command to find the names and datatypes of columns within a table. The other commands do not exist. There are also tab_columns views in the data dictionary.
7. **B** — Only statements issued after the ALTER statement are rolled back. As part of the recovery process, Oracle automatically rolls back all uncommitted transactions when recovering from a failure. However, the ALTER statement is a DDL statement, and it commits any open transactions. Transactions do not stay open when connection to the client is lost. For more information, see the "Transaction control statements" section in this chapter.

8. **C —** There is no way to undo the DELETE after you close SQL*Plus. When you close this utility normally, it automatically commits any uncommitted transactions. The only way to recover the lost data is to restore from a backup. Logging in as the System account does not change this behavior. For more information, see the "Transaction control statements" section in this chapter.
9. **C —** The problem with this statement is that the subquery returns fewer values than are listed in the INSERT list. In order to insert data from a subquery, the subquery must return the same number of columns in the same order as they appear in the INSERT list. If the subquery returns a fewer or greater number of columns in the SELECT list, the statement generates an error. For more information, see the "Using subqueries in INSERT statements" section in this chapter.
10. **A —** To remove the error, you need to place parentheses around the two columns in the SET clause. Without the parentheses, Oracle tries to match both values returned by the subquery to the second listed column and cannot find any value to set for the first. Placing single or double quotes around the two columns does not bind them together. The subquery requires the parentheses, and removing them only creates another error. For more information, see the "Using subqueries in UPDATE statements" section in this chapter.

Scenarios

1. Because writers do not block readers in Oracle, no conflict occurs between users when it comes to the customer records. When your user attempts to query a client record while another user is updating it, the first user receives the pre-update values. When two users try to update the same values, the second user is blocked and has to wait for the first to complete his or her transaction. The best way to modify the INSERT statement to insert the current date is to use the SYSDATE function as part of the VALUES clause in the INSERT statement. To speed up the update process, make sure that the transactions are kept as short as possible. If two DML operations are not logically linked, commit the first before starting the second. This releases any locks and makes the data available to other users. It also prevents blocking. You need only one value to update all of the products. If you include the column name and an arithmetic operator in the SET clause, it updates each row separately:

   ```
   UPDATE Products
   Set unitprice = unitprice * 1.25
   ```

   If you omit the WHERE clause, it updates the entire table.
2. You can move the data by using an INSERT statement with a subquery. You can deal with the extra column by either including an INSERT list and omitting the Archive_date columns or including the NULL operator as a literal in the SELECT list. The easiest way to deal with the translation of the date data is to use the TO_DATE function in the subquery. This passes the data to the destination table in the correct format. The subquery would look something like this:

```
INSERT INTO Order_History (Order_num, Customer_no, Orderdate,
shipdate, Sales_no)
SELECT OrderID, CustomerID, TO_DATE(Order_Date, 'Month DD,
YYYY'), TO_DATE(Ship_Date, 'Month DD, YYYY'), SalespersonID
FROM Orders
WHERE TO_DATE(Order_Date, 'Month DD, YYYY') < '01-JAN-1997';
```

The TO_DATE function is needed in the WHERE clause of the subquery to enable you to use a comparison operator with the date (comparison operators using dates do not work with character data). You can verify the rows by running a query on the Orders table using the COUNT() function (see Chapter 2) to see how many rows match the criteria of the WHERE clause and then run the same query on the Order_History table after the transfer. If the count is not the same, you can issue a ROLLBACK to undo the transaction. To remove the rows from the Orders table, use a DELETE statement with the same WHERE clause condition you used in the subquery of your INSERT statement.

Lab Exercises

Lab 5-1 Inserting data

2.

```
INSERT INTO Instructors
(InsturctorID,FirstName,LastName,Address1,City,State
VALUES (101, 'Jessica',      'Jones', '4 Apple Rd.',
     'Flint', 'MI')
```

All other rows with have the same syntax with different values in the VALUES list.

3. A NULL will be inserted into all columns not referenced in the INSERT list.

Lab 5-2 Updating data

2.

```
UPDATE Instructors
SET InstructorType = 'FT', PerDiemCost = 500,
    PerDiemExpenses = 400
WHERE InstructorID = 101
```

All other rows will use the same syntax with different values in the SET and WHERE clauses.

3. Because you can modify multiple columns with one UPDATE statement, the minimum number of queries need is three — one for each row.

4.

```
UPDATE Instructors
SET PerDiemExpenses = PerDiemExpenses * 1.25
WHERE PerDiemExpenses < 500;
```

Lab 5-3 Testing transactional controls

6. The second user sees a value of 2000. This is the original value before the UPDATE statement was run.

8. The first user sees a value of 1000. This is the result of the UPDATE statement.

11.

Testing Transactional Control	
CourseID	***RetailPrice***
201	4000
110	2000

14. There is no change in the results because the changes from the first UPDATE are still not commited.

17.

Testing Transactional Control	
CourseID	***RetailPrice***
201	4000
110	1000

Lab 5-4 Deleting data

2.

```
DELETE FROM Instructors
WHERE InstructorID = 101;
```

Repeat this query for each Instructor in Table 5-1.

PART II

Managing Database Objects

This part of the book deals with SQL*Plus, creating tables and other database objects, and managing users and permissions in the database.

Chapter 6 deals with SQL*Plus, the primary tool used to create and manage objects in the Oracle database. It also enables you to write and execute PL/SQL code and create, edit, and run scripts. The chapter starts by discussing how SQL*Plus works and then follows with the definition and use of variables. We introduce commands that are specific to the SQL*Plus environment and how to use them to format output. Finally, we show how to customize SQL*Plus to meet your specific requirements.

Chapter 7 is the big one in this book. This chapter walks you through the process of creating tables, views, indexes, sequences, synonyms, and constraints. We also provide information on how to manage these objects, and how to drop them when they are no longer necessary.

Chapter 8 is important if you desire a secure environment for your database, or if you want anyone beside yourself to see the data in the tables you have created. In fact, you won't be able to create tables or other objects unless you have the right permissions. We discuss how to create and manage users, the value of roles in the assignment and management of permissions, how to assign privileges to users and roles, and how to delegate the administration of security to others. The majority of the security management in any database is done by the database administrator (DBA), but all users should be familiar with the basics of security introduced in this chapter.

In This Part

Chapter 6
The SQL*Plus Environment

Chapter 7
Creating and Managing Oracle Database Objects

Chapter 8
Configuring Security in Oracle Databases

CHAPTER 6

The SQL*Plus Environment

EXAM OBJECTIVES

- Producing readable output with SQL*Plus
  - Produce queries that require an input variable
  - Customize the SQL*Plus environment
  - Produce more readable output
  - Create and execute script files
  - Save customizations

CHAPTER PRE-TEST

1. How do you use a substitution variable in a SQL statement?
2. How do you save a SQL statement into a file?
3. How do you execute SQL commands stored in a file?
4. Name four options you can change using the SET command.
5. In what file can you save SET commands so that they are run automatically when you log in to SQL*Plus?
6. What SQL*Plus command enables you to send output from a SQL statement to a file?
7. What table can you create to limit the commands that may be executed from SQL*Plus?
8. How many SQL commands are stored in the SQL buffer?
9. Which SQL*Plus command will give you a description of a database table?
10. What is the difference between the DEFINE and ACCEPT commands?

✦ Answers to these questions can be found at the end of the chapter. ✦

This chapter covers the features of the SQL*Plus tool. Learning to use the features of the SQL*Plus tool will make you a more productive SQL programmer. In this chapter, you learn to use SQL*Plus to access and modify your last command in the SQL buffer, to save commands in files, and to use variables to write reusable scripts. Finally, you learn to format command output and save that output to files.

All the new commands introduced in this chapter are SQL*Plus commands. There are a few differences between SQL commands, such as SELECT and UPDATE, and SQL*Plus commands, such as EXIT and DESCRIBE:

- ✦ SQL*Plus commands can be abbreviated.
- ✦ SQL*Plus commands are not saved in the SQL buffer.
- ✦ SQL*Plus commands do not require a command terminator such as a forward slash (/) or semicolon (;).

The SQL Buffer

Every SQL command you type in SQL*Plus is saved in a SQL buffer. The buffer stores the last SQL command you entered in SQL*Plus.

SQL*Plus enables you to access the SQL buffer. This saves you time correcting and modifying your SQL statements.

In order to see how the SQL buffer works, execute a SELECT statement such as the following:

```
SQL> SELECT firstname, lastname
  2  FROM students
  3  WHERE studentnumber=1000;

FIRSTNAME                      LASTNAME
------------------------------ ------------------------------
   John                           Smith
```

Now use the LIST command to show the contents of the SQL buffer:

```
SQL> LIST
  1  SELECT firstname, lastname
  2  FROM students
  3* WHERE studentnumber=1000
```

Execute a different SQL statement:

```
SQL> SELECT coursename, retailprice
  2  FROM courses
  3  WHERE coursenumber=300;


COURSENAME                                RETAILPRICE
----------------------------------------- -----------
Basic PL/SQL                                     2500
```

Execute the LIST command again and take note that the contents of the SQL buffer have changed. Also notice that only the most recent SQL statement is stored in the SQL buffer. SQL*Plus stores only the last SQL statement.

You can reexecute a command stored in the buffer by using either the RUN command or by typing a forward slash (/) at the command line. RUN can be abbreviated with R. Try both.

```
SQL> RUN
  1  SELECT coursename, retailprice
  2  FROM courses
  3* WHERE coursenumber=300

COURSENAME                                RETAILPRICE
----------------------------------------- -----------
Basic PL/SQL                                     2500

SQL> /

COURSENAME                                RETAILPRICE
----------------------------------------- -----------
Basic PL/SQL                                     2500
```

You can modify the contents of the SQL buffer using the EDIT command. (EDIT can be abbreviated with ED.) Typing the EDIT command, or ED, launches a text editor so you can modify the contents of the buffer.

```
SQL> EDIT
```

After you open the editor, you can use all the features of the text editor to modify your command. Once you have completed all the desired changes, you save your changes and exit the text editor. You are brought back to the SQL> prompt. You can now run the modified command by using the RUN or / command to execute the contents of the buffer.

Defining Variables

Objective

- Produce queries that require an input variable

You can use two types of variables within a SQL statement:

- **Substitution variables:** Can be created within the SQL statement or by using the DEFINE or ACCEPT SQL*Plus commands.
- **Bind variables:** Are created using the SQL*Plus command VARIABLE.

Substitution variables

Substitution variables enable you to specify values to be used within a SQL statement when the SQL statement is run. Substitution variables are prefixed with an ampersand (&).

Try executing the following SQL statement:

```
SQL> SELECT firstname, lastname
  2  FROM students
  3  WHERE studentnumber=&p_student;
```

When you are prompted to enter a value for p_student, specify a student number of 1005.

```
Enter value for p_student: 1005
old   3: WHERE studentnumber=&p_student
new   3: WHERE studentnumber=1005

FIRSTNAME                      LASTNAME
------------------------------ ------------------------------
John                           Hee
```

When you ran the SQL statement, you were prompted to enter a value for the variable p_student. After you entered a value, the SQL statement executed using the value you specified. Now try running the same SQL statement again by typing **RUN** or /:

```
SQL> /
Enter value for p_student: 1002
old   3: WHERE studentnumber=&p_student
new   3: WHERE studentnumber=1002

FIRSTNAME                      LASTNAME
------------------------------ ------------------------------
Jane                           Massey
```

When you ran the statement again, you were prompted again to enter a value for the variable. After you enter a new value, the SQL statement runs with the new value you specify.

The old and new messages you see after you specify a value for the variable are displayed so you can see where the variable is used in the SQL statement and the changes made to the SQL statement after you specify a value for the variable. If you do not want to see these messages, you can type **SET VERIFY OFF** at the SQL prompt.

Let's try another example:

```
SQL> SELECT studentnumber, firstname, lastname
  2  FROM students
  3  WHERE lastname = &p_name;

Enter value for p_name: Hee
old   3: WHERE lastname = &p_name
new   3: WHERE lastname = Hee
SELECT studentnumber, firstname, lastname
*
ERROR at line 1:
OCA-30035: column not found
[POL-5205] column not found
```

In the preceding example, you get an error message. That is because the substitution variable is being used for a character string, and we did not put single quotes around the string. Whenever you use a variable to hold a date or character value, you must put quotes around the value. Let's execute it again, but this time put single quotes around the name:

```
SQL> /
Enter value for p_name: 'Hee'
old   3: WHERE lastname = &p_name
new   3: WHERE lastname = 'Hee'

STUDENTNUMBER FIRSTNAME                      LASTNAME
------------- ------------------------------ ---------------
         1005 John                           Hee
```

It's a good habit to put quotes around the variable name itself in the SQL statement so you don't have to remember to specify the quotes at runtime:

```
SQL> SELECT studentnumber, firstname, lastname
  2  FROM students
  3  WHERE lastname = '&p_name';
Enter value for p_name: Hee
old   3: WHERE lastname = '&p_name'
new   3: WHERE lastname = 'Hee'
```

```
STUDENTNUMBER FIRSTNAME                      LASTNAME
------------- ------------------------------ ---------------
         1005 John                           Hee
```

Substitution variables are versatile. You can use them to specify values used in the WHERE clause, table names, and column names. For example you might try something like this:

```
SQL> SELECT &column1, &column2
  2  FROM &table;

Enter value for column1: firstname
Enter value for column2: lastname
Enter value for table: students

FIRSTNAME                      LASTNAME
------------------------------ ----------------------
John                           Smith
Davey                          Jones
Jane                           Massey
```

If you use the same variable name twice in the same SQL statement, you are prompted for a value each time the variable is referenced. This can cause problems. In the following example, the same variable name &column1 is used in the SELECT clause and the ORDER BY clause:

```
SQL> SELECT &column1, &column2
  2  FROM &table
  3  ORDER BY &column1;

Enter value for column1: firstname
Enter value for column2: lastname
Enter value for table: students
Enter value for column1: firstname

FIRSTNAME                      LASTNAME
------------------------------ ------------
Chris                          Patterson
Davey                          Jones
Gordon                         Jones
```

When we reference the same variable twice, we are prompted for values twice. You can specify a different value for the variable each time you are prompted. If you want to use the same variable twice in one SQL statement, with the same value each time, but do not wish to be prompted twice, put && at the beginning of the variable. When you specify && in front of a variable, that variable retains its value until you log out of SQL*Plus.

```
SQL> SELECT &&column1, &column2
  2  FROM &table
  3  ORDER BY &column1;
```

```
Enter value for column1: firstname
Enter value for column2: lastname
Enter value for table: students

FIRSTNAME                      LASTNAME
------------------------------ ----------------
Chris                          Patterson
Davey                          Jones
Gordon                         Jones
```

DEFINE

The DEFINE command can be used to create substitution variables and assign them values. These variables can then be used within SQL statements. Any variable created with the DEFINE command stays in memory until you exit the SQL*Plus session or until you delete the variable using the UNDEFINE command. The DEFINE command is limited in that it can be used only to create variables of datatype CHAR and you must assign a value to the variable when it is created. If you wish to change the value of the variable later, you execute a second DEFINE command for the same variable but assign it a different value.

```
SQL> DEFINE v_student_nbr=1000
SQL> SELECT firstname, lastname
  2  FROM students
  3  WHERE studentnumber = &v_student_nbr;


FIRSTNAME                      LASTNAME
------------------------------ --------------------
John                           Smith

SQL> SELECT COUNT(*)
  2  FROM students
  3  WHERE studentnumber > &v_student_nbr;


  COUNT(*)
----------
        10
```

You can examine the contents of a variable created using the DEFINE command by typing **DEFINE** and specifying the variable name:

```
SQL> DEFINE v_student_nbr
DEFINE V_STUDENT_NBR   = "1000" (CHAR)
```

You can also use the DEFINE command to get a list of all currently defined variables by typing the DEFINE command and specifying no arguments. Certain variables are created when you start your SQL*Plus session, so you will see variables listed that you did not explicitly create.

```
SQL> DEFINE
DEFINE _O_VERSION      = "Oracle Open Client Adapter for ODBC
2.0.2.15.0
Oracle Lite ORDBMS 4.0.0.2.0" (CHAR)
DEFINE _O_RELEASE      = "0" (CHAR)
DEFINE _RC             = "1" (CHAR)
DEFINE V_STUDENT_NBR   = "1000" (CHAR)
```

To remove a variable created with the DEFINE command, use the UNDEFINE command:

```
SQL> UNDEFINE v_student_nbr
SQL> DEFINE v_student_nbr
symbol v_student_nbr is UNDEFINED
```

The DEFINE command can also be used to list substitution variables created by specifying &&. The UNDEFINE command can be used to delete substitution variables created by specifying && or ACCEPT.

ACCEPT

The ACCEPT command can be used to prompt users to enter values for a substitution variable before the SQL statement is executed. The substitution variable is created when the ACCEPT command is executed and remains in memory until you exit SQL*Plus or you use the UNDEFINE command to delete the variable.

```
ACC[EPT] variable [NUM[BER]|CHAR|DATE] [FOR[MAT] format]
[DEF[AULT] default] [PROMPT text|NOPR[OMPT]] [HIDE]
```

The PROMPT option specifies the message to display when you ask the user to enter a value for the variable:

```
SQL> ACCEPT v_name PROMPT "Enter student last name: "
Enter student last name: Holland
SQL> SELECT studentnumber, firstname, lastname
  2  FROM students
  3  WHERE lastname = '&v_name';


STUDENTNUMBER FIRSTNAME                      LASTNAME
------------- ------------------------------ -----------
         1007 Roxanne                        Holland
```

The ACCEPT command creates a variable of datatype CHAR by default, but datatypes of NUMBER or DATE can also be specified. If you create a variable of datatype NUMBER or DATE, you can also specify the FORMAT in which they must be entered:

```
SQL> ACCEPT p_date DATE FORMAT dd/mm/yy PROMPT "Enter date "
Enter date 01/01/01
```

```
SQL> SELECT classid, coursenumber, locationid, status
  2  FROM scheduledclasses
  3  WHERE startdate > '&p_date';


   CLASSID COURSENUMBER LOCATIONID STATUS
---------- ------------ ---------- ----------
        50          100        100 Confirmed
        51          200        300 Confirmed
        53          100        300 Hold
```

When entering any SQL*Plus command, if you write the command over two lines, you must put a hyphen (-) at the end of the first line to indicate the command continues on the next line:

```
SQL> ACCEPT p_low_date DATE FORMAT dd/mm/yyyy -
> PROMPT "Enter a date (dd/mm/yyyy) "
```

The DEFAULT option enables you to specify a default value for the variable.

The HIDE option displays asterisks when you type a value for the variable. This is useful when you are prompting for sensitive information such as a password.

Bind variables

The VARIABLE command enables you to create bind variables that may be populated and referenced in a PL/SQL block. After the variable is populated within a PL/SQL block, it can then be printed using the PRINT command. The VARIABLE command enables you to specify a datatype of NUMBER, CHAR, VARCHAR, or REFCURSOR. When you use the VARIABLE command to create a variable of datatype CHAR or VARCHAR, you must specify a size. No size is specified for variables of datatype NUMBER.

```
VAR[IABLE] [ variable [NUMBER|CHAR|CHAR(n)|VARCHAR2(n)|
REFCURSOR]]

SQL> VARIABLE my_student NUMBER
SQL> BEGIN
       :my_student := 1000;
     END;
2 @code:/
SQL> PRINT my_student
MY STUDENT
-----------
10
```

Typing the VARIABLE command with no arguments provides a list of all variables created with the variable command:

```
SQL> VARIABLE
variable   my_student
datatype   NUMBER
```

SQL*Plus Commands

SQL*Plus features a set of commands that enable you to save commands to files, execute commands stored in files, and send output to files. These commands enable you to create scripts and reports. In this section, we examine the most important SQL*Plus commands.

DESCRIBE

The DESCRIBE command enables you to see a description of a database table. The DESCRIBE command returns a list of all the columns on the specified database table, their datatypes, and whether the columns are NOT NULL.

```
DESC[RIBE] [table]

SQL> DESCRIBE courses
 Name                           Null?    Type
 ------------------------------ -------- ----
 COURSENUMBER                   NOT NULL NUMBER(10)
 COURSENAME                     NOT NULL VARCHAR2(200)
 REPLACESCOURSE                          NUMBER(10)
 RETAILPRICE                    NOT NULL NUMBER(10)
 DESCRIPTION                             VARCHAR2(2000)
```

SAVE

The SAVE command enables you to save the contents of the SQL*Plus buffer into a file. This file can be executed later using the START or @ commands. The file is saved to the working directory of SQL*Plus by default. You can save it to a different directory by specifying a path with the filename:

```
SAV[E] file_name[. ext] [CRE[ATE]|REP[LACE]|APP[END]]

SQL> select coursename, retailprice
  2  FROM courses;


COURSENAME                             RETAILPRICE
-------------------------------------- -----------
Basic SQL                                     2000
Advanced SQL                                  2000

SQL> SAVE myselect
Created file myselect
```

When you do not specify a file extension, SQL*Plus saves the file with a .SQL extension. If you specify a different file extension, you must include the file extension when editing the file with EDIT command or running the file with the START or

@ commands. The SAVE command uses the CREATE option by default. With the CREATE option, when the file does not exist, a new file is created with the specified name. If the file does exist, you get an error message:

```
SQL> SAVE myselect
File "myselect.SQL" already exists.
Use another name or "SAVE filename REPLACE".
```

If you specify an existing filename with the SAVE command, you must use either the REPLACE or APPEND option. The REPLACE option overwrites the existing file with the contents of the buffer. The APPEND command adds the contents of the buffer to the end of the file.

EDIT

The EDIT command enables you to modify the contents of a file from the SQL*Plus tool. As we saw in the section on the SQL*Plus buffer, typing EDIT brings up the buffer in a text editor so you can modify and execute the last command stored in the buffer. When you specify a filename with the EDIT command, it opens the specified file in the text editor. When the filename specified does not exist, you are given the opportunity to create a new file. When you do not specify a file extension, SQL*Plus uses the default file extension of .SQL:

```
ED[IT] [ file_name[. ext]]
SQL> EDIT my_cmdfile
```

If you are working with SQL*Plus for Windows, you control which text editor is launched with the EDIT command by choosing Edit➪Editor➪Define Editor from the menu and specifying any application such as Notepad or WordPad that creates ASCII text files. If you are working in UNIX, you control which text editor is launched by specifying a value for the SQL*Plus variable _EDITOR:

```
SQL> DEFINE _EDITOR=vi
```

GET

The GET command enables you to fetch the contents of a file into the SQL buffer. Once the contents of the file have been fetched into the SQL buffer, they can be executed using the RUN or / commands, or they can be edited by using the EDIT command.

```
GET file_name[. ext]

SQL> GET myselect
  1  select coursename, retailprice
  2* FROM courses
SQL> /
```

```
COURSENAME                                RETAILPRICE
----------------------------------------- -----------
Basic SQL                                        2000
Advanced SQL                                     2000
```

Because the buffer can hold only one SQL statement at a time, when you use the GET command to read the contents from a file that contains multiple SQL commands or a mixture of SQL and SQL*Plus commands, you receive an error if you try to execute the buffer using RUN or /.

START

The START command enables you to execute the contents of a command file. The default file extension is .SQL. The @ and START commands are interchangeable.

```
STA[RT] file_name[. ext]
SQL> START myselect

COURSENAME                                RETAILPRICE
----------------------------------------- -----------
Basic SQL                                        2000
Advanced SQL                                     2000
```

@

The @ command enables you to execute the contents of a file created using the SAVE or EDIT command. The default file extension is .SQL. The @ and START commands are interchangeable:

```
@ file_name[. ext] [ arg...]
SQL> @myselect

COURSENAME                                RETAILPRICE
----------------------------------------- -----------
Basic SQL                                        2000
Advanced SQL                                     2000
```

In some older versions of SQL*Plus, the @ command did not work when you executed a file that contained a mixture of SQL and SQL*Plus commands. If this is the case with your file, you may have to use the START command.

SPOOL

The SPOOL command enables you to send the screen output to a file:

```
SPO[OL] [ file_name[. ext]|OFF|OUT]
```

To start sending screen output to a file, use the spool command and specify a filename where the output is to be written. The default file extension is .LST or .LIS, depending on your operating system.

```
SQL> SPOOL myoutput
```

After starting to spool your output, execute the SQL statement that generates the desired output, or type a command. The command and the output are written to the specified spool file.

```
SQL> select coursename, retailprice
  2  FROM courses
  3  /

COURSENAME                                RETAILPRICE
----------------------------------------- -----------
Basic SQL                                        2000
Advanced SQL                                     2000
```

After you generate the desired output, halt the spooling by using either SPOOL OFF or SPOOL OUT. SPOOL OFF halts the spooling and closes the spool file. SPOOL OUT halts the spooling and sends the spool file to the printer.

```
SQL> SPOOL OFF
SQL> SPOOL OUT
```

The SPOOL OUT command is not supported on all operating systems. Windows does not support SPOOL OUT.

EXIT

The EXIT command enables you to exit your SQL*Plus session. Any unsaved updates, insertions, or deletions are committed to the database when you exit. QUIT is a synonym for the EXIT command.

```
{EXIT|QUIT}
SQL> EXIT
```

Customizing SQL*Plus with SET Commands

- Customize the SQL*Plus environment

The SQL*Plus environment can be customized using the SET command to change the SQL*Plus environment options. A number of settings can be changed using the SET command. For a full listing of the commands, type **SHOW ALL**:

```
SQL> show all
appinfo is OFF and set to "SQL*Plus"
arraysize 15
...
verify ON
shiftinout INVISIBLE
wrap : lines will be wrapped
```

To change any of these settings, use the SET command. You specify which option you want to change and the new value for that option. You can also modify the settings with the Menu option "Options – Environment," which gives you a complete list of the SQL*Plus environment options. Highlight the name of the setting you want to change and specify the value for the setting under "Value." After a value has been changed, it holds its value until you log out of SQL*Plus or change the value of that setting.

We won't cover all the environment settings that can be modified; we focus on the most commonly modified settings.

ARRAYSIZE

When you execute a SELECT statement, SQL*Plus fetches ARRAYSIZE records at a time from the database and stores them in a local buffer. When you execute a SELECT statement that fetches records that contain large amounts of data, you may get the following error message:

```
buffer overflow. Use SET command to reduce ARRAYSIZE or
increase MAXDATA.
```

This error message indicates that SQL*Plus cannot fit the number of records specified in ARRAYSIZE into the buffer, and you need to decrease ARRAYSIZE, so that it fetches less records into the buffer at a time. The default value for ARRAYSIZE is 20.

```
ARRAY[SIZE] {20| n}
SQL> SET ARRAYSIZE 10
```

COLSEP

The COLSEP option affects the character displayed between columns when you execute a SELECT statement. The default value is spaces.

```
COLSEP {_| text}
```

```
SQL> SELECT firstname, lastname
  2  FROM students;


FIRSTNAME                      LASTNAME
------------------------------ -----------
John                           Smith
Davey                          Jones

SQL> SET COLSEP ,

SQL> SELECT firstname, lastname
  2  FROM students;
```

If you set COLSEP to comma (,) and spool the output of your SELECT statement to a file using the SPOOL command, you can create a comma delimited file. Comma delimited files are often used as input files for programs and applications such as Microsoft Excel.

```
FIRSTNAME                     ,LASTNAME
------------------------------,----------------
John                          ,Smith
Davey                         ,Jones
```

FEEDBACK

The FEEDBACK option affects when you see a message telling you how many records were selected from a query, or affected by a Data Manipulation Language (DML) Statement. The default value is 6, so you see a message telling you how many rows were selected when you retrieve six rows or more from your query. With FEEDBACK set to its default value of 6, when you execute a DML statement such as INSERT, UPDATE, DELETE, COMMIT, or ROLLBACK you see a message displayed regardless of the number of rows affected.You can suppress this message altogether by specifying FEEDBACK OFF.

```
FEED[BACK] {6| n|OFF|ON}

SQL> SELECT firstname, lastname
  2  FROM students;


FIRSTNAME                      LASTNAME
------------------------------ ------------------------------
John                           Smith
...
Chris                          Patterson
11 rows selected.

SQL> SELECT locationid, city
  2  FROM locations;
```

```
LOCATIONID CITY
---------- ------------------------------
       100 New York
       200 San Francisco
       300 Toronto

SQL> SET FEEDBACK 1
SQL> /

LOCATIONID CITY
---------- --------------------
       100 New York
       200 San Francisco
       300 Toronto

   3   rows selected.

SQL> SET FEEDBACK OFF
    SQL> SELECT firstname, lastname
    2  FROM students;

    FIRSTNAME                      LASTNAME
    ------------------------------ -------------
    John                           Smith

    Chris                          Patterson
```

HEADING

The HEADING option enables you to control whether the column headers are displayed.

```
HEA[DING] {OFF|ON}

SQL> SET HEADING OFF
SQL> SELECT firstname, lastname
  2  FROM students;

John                           Smith
Davey                          Jones
```

LINESIZE

The LINESIZE option controls how many characters are displayed on each line in the SQL*Plus window. It is often increased when you are writing output to a file and you want to fit an entire record on one line.

```
LIN[ESIZE] {80| n}
SQL> SET LINESIZE 40
SQL> SELECT coursename, retailprice
  2  FROM courses;
```

```
COURSENAME
----------------------------------------
RETAILPRICE
-----------
Basic SQL
       2000

Advanced SQL
       2000
...
SQL> SET LINESIZE 300
SQL> /

COURSENAME                          ...RETAILPRICE
-------------------------------------------------
Basic SQL                           ...2000
Advanced SQL                        ...2000
```

LONG

The LONG option controls how many characters are displayed when you SELECT a column of datatype LONG from a table. The default value is 80 characters.

```
LONG {80| n}

SQL> SET LONG 500
```

PAGESIZE

The PAGESIZE option controls how many lines appear on a page. The default value is 24. Every time a new page starts, the column headers are displayed, and any TTITLE or BTITLE values will appear. (TTITLE and BTITLE are explained in the section "Headers and Footers," later in this chapter.) When counting the number of lines on a page, the column headers themselves, as well as any TTITLE and BTITLE lines, are included in the count.

```
PAGES[IZE] {24| n}
SQL> SET PAGESIZE 100
```

PAUSE

The PAUSE option enables you to scroll through the data returned by a SELECT statement. The number of lines of text that appear before the screen output stops and you must press a key to continue is defined by the option PAGESIZE. SET PAUSE ON to scroll through data. Set PAUSE OFF to remove scrolling. SET PAUSE text displays the specified text on the screen whenever the screen pauses in scrolling.

```
PAU[SE] {OFF|ON| text}
SQL> SET PAUSE ON
SQL> SET PAUSE 'Hit a key to continue'
SQL> /
Hit a key to continue

     FIRSTNAME                      LASTNAME
     ------------------------------ ---------------
     John                           Smith
     Davey                          Jones

     Hit a key to continue
```

TERMOUT

The TERMOUT option is used within SQL scripts. If you do not wish to see the output from a SQL script displayed on the screen, set TERMOUT OFF. This is usually used when the output from a SQL script is being spooled to a file and the user does not want the output sent to the screen because it will slow down execution.

```
TERM[OUT] {OFF|ON}
SQL> @MYSCRIPT
        ID LAST_NAME                FIRST_NAME
-- --------                 -----
         1 VELASQUEZ                CARMEN

SQL> SET TERMOUT OFF
SQL> @MYSCRIPT
SQL>
```

Formatting Output with SQL*Plus

Objective

- ✦ Produce more readable output

The SQL*Plus tool includes a set of commands that enable you to format query output and create formatted reports. These commands are often saved in script files with SELECT statements to create simple reports.

Headers and footers

SQL*Plus contains a set of commands that enables you to add headers and footers to SQL output. Headers and footers may be displayed on each page or only once per SQL statement.

TTITLE

The TTITLE command enables you to display a title at the top of each report page.

```
TTI[TLE] [ printspec [ text| variable] ...]|[OFF|ON]
```

BTITLE

The BTITLE command enables you to display a title at the bottom of each report page.

```
BTI[TLE] [ printspec [ text| variable] ...]|[OFF|ON]
```

REPHEADER

The REPHEADER command enables you to display a title at the start of the entire report.

```
REPH[EADER] [ printspec [ text| variable] ...]|[OFF|ON]
```

REPFOOTER

The REPFOOTER command enables you to display a title at the end of the report.

```
REPF[OOTER] [ printspec [ text| variable] ...]|[OFF|ON]
```

After you have decided where you want the title to appear, you need to specify what will appear in the title and where. This is specified in the printspec clause of each command.

The following options can be specified in the printspec:

- ✦ S[KIP] [n] — Skip *n* lines.
- ✦ LE[FT] — Left-justify the text.
- ✦ CE[NTER] — Center-justify the text.
- ✦ R[IGHT] — Right-justify the text.
- ✦ FORMAT — Specify a date or numeric format for the text.

The following example displays the text "Course List" centered at the top of each page and then skips one line before displaying data:

```
SQL> TTITLE CENTER "Course List" SKIP 1
SQL> SELECT coursename
  2  FROM courses;

                                    Course List
COURSENAME
------------------------------------------------------------
Basic SQL
Advanced SQL
```

The following example displays a left-justified message "End of Report" at the end of the SQL output:

```
SQL> REPFOOTER "***End of Report***"
SQL> /

COURSENAME
--------------------------------
Basic SQL
...
Using your PL/SQL skills
***End of Report***
```

A series of variables may be used in these commands as well:

- ✦ SQL.LNO — Current line number
- ✦ SQL.PNO — Current page number
- ✦ SQL.USER — Current username

These variables may be displayed in the headers and footers — for example, when you want to display the title "Registration Report" and a page number on each page. You might do the following:

```
SQL> TTITLE LEFT SQL.PNO CENTER "Course List"
SQL> /

         1                          Course List
COURSENAME
--------------------------------------------------
Basic SQL
Advanced SQL
```

When you create a header or footer for your SQL output, those settings are saved until you exit SQL*Plus. If you do not want the headers and footers appearing on the output of subsequent SQL statements, you should turn off the headers and footers.

```
SQL> TTITLE OFF
SQL> REPFOOTER OFF
```

COLUMN

The COLUMN command enables you to format the output of a column with a given column name.

```
COL[UMN] [{ column| expr} [ option ...]]
```

where option is one or more of the following choices:

- ✦ CLE[AR] — Removes the column format setting for a column.
- ✦ FOR[MAT] format — Formats the data within the column.
- ✦ HEA[DING] text — Changes the displayed column heading.
- ✦ JUS[TIFY] {L[EFT]|C[ENTER]|C[ENTRE]|R[IGHT]} — Justifies the column heading.
- ✦ NOPRI[NT]|PRI[NT] — Suppresses or displays the column output.
- ✦ NUL[L] text — Changes the value displayed when the column contains a NULL value.
- ✦ WRA[PPED]|WOR[D_WRAPPED]|TRU[NCATED] — Determines whether data is wrapped or truncated when the format specified is not long enough to display the entire string.

For example, when you want to format the retailprice column on the courses table and change the column heading to "Retail Price", you specify the following:

```
SQL> COLUMN retailprice FORMAT $9,999,990.00 -
HEADING "Retail Price"
SQL> SELECT retailprice
  2  FROM Courses;


Retail Price
--------------
     $2,000.00
```

The format specifications for date and number datatypes are similar to those used in the TO_CHAR function.

For columns of datatype NUMBER, use the following format options:

- ✦ 9 — The number of nines specifies the number of digits to display, leading zeros will be suppressed.
- ✦ 0 — The number of zeros specifies the number of digits to display, leading zeros will be displayed.
- ✦ $ — The dollar sign is used to prefix a number with a dollar sign.
- ✦ , — The comma displays in the position it is placed.
- ✦ . — The period displays in the position it is placed.

```
SQL> COLUMN retailprice FORMAT $0,000,000.00
SQL> SELECT retailprice
  2  FROM courses;
```

```
   RETAILPRICE
--------------
 $0,002,000.00
 $0,002,000.00
```

For character datatype columns, you can specify the number of characters a column can use for display. If the column may contain data that exceeds the specified length, you should also specify whether the text should be wrapped, word-wrapped, or truncated.

```
SQL> COLUMN coursename FORMAT A20 WORD_WRAP
SQL> SELECT coursename
  2  FROM courses;


COURSENAME
--------------------
Basic SQL
Using your
PL/SQL skills
```

By inserting a pipe symbol (|) in the HEADING, you can specify a header that appears over two lines:

```
SQL> COLUMN retailprice FORMAT $9,999,990.00 -
HEADING "Retail|Price"
SQL> SELECT retailprice
  2  FROM courses;


        Retail
         Price
--------------
     $2,000.00
     $2,000.00
```

To check the current format settings for a particular column, you can use the COLUMN command with no options to list the settings for a column:

```
SQL> COLUMN retailprice
column   retailprice ON
heading  'Retail|Price' headsep '|'
format   $9,999,990.00
```

To obtain a list of all the columns with format settings, you use the COLUMN command and do not specify a column name:

```
SQL> COLUMN
```

To remove the format settings for a column, use the CLEAR option:

```
SQL> COLUMN retailprice CLEAR
```

BREAK

The BREAK command enables you to group related data in your SQL output.

```
BRE[AK] [ON column_name [action]]
```

[action] specifies what action should be taken when the specified column changes. You can specify two different actions:

- ✦ [SKI[P] n|[SKI[P]] PAGE] — Enables you to leave blank lines between records or start a new report page between records.
- ✦ [NODUP[LICATES]|DUP[LICATES]] — Enables you to suppress or display duplicate values in a column.

Before you use the BREAK command, you should add an ORDER BY clause to your SQL statement to ensure the data is sorted by the columns by which you want to group.

```
SQL> SELECT city, lastname, firstname
  2  FROM students
  3  ORDER by city;
```

After the data is sorted, you can use BREAK to group the output. You could put each city on a different page, for example:

```
SQL> BREAK ON city SKIP PAGE

SQL> /

CITY            LASTNAME                       FIRSTNAME
--------------- ------------------------------ ------------
Dallas          Andrew                         Susan

CITY            LASTNAME                       FIRSTNAME
--------------- ------------------------------ ------------
New York        Hee                            John
                Massey                         Jane
```

You can also use the BREAK command to insert blank lines between records when the column value changes:

```
SQL> BREAK ON city SKIP 2
SQL> /

CITY            LASTNAME                       FIRSTNAME
--------------- ------------------------------ -----------
Dallas          Andrew                         Susan


New York        Hee                            John
                Massey                         Jane
```

By default, the BREAK command also suppresses duplicate values in the same column. If you want to list the duplicate values, specify the DUPLICATES option:

```
SQL> BREAK ON city SKIP 2 DUPLICATES
SQL> /

CITY            LASTNAME                       FIRSTNAME
--------------- ------------------------------ ----------
Dallas          Andrew                         Susan


New York        Hee                            John
New York        Massey                         Jane
```

To remove BREAKS, use the command CLEAR BREAKS:

```
SQL> CLEAR BREAKS
```

COMPUTE

When you use the BREAK command to group your output, you can use the COMPUTE command to perform calculations on each group:

```
COMP[UTE] [function] OF {column} ON {column}]
```

You can choose from the following functions:

- ✦ AVG — Returns the average of non-null values of datatype NUMBER.
- ✦ COU[NT] — Returns the count of non-null values for all datatypes.
- ✦ MAX[IMUM] — Returns the maximum value for datatypes NUMBER or CHARACTER.
- ✦ MIN[IMUM] — Returns the minimum value for datatypes NUMBER or CHARACTER.
- ✦ NUM[BER] — Returns the count of rows for all datatypes.
- ✦ STD — Returns the standard deviation of non-null values for datatype NUMBER.
- ✦ SUM — Returns the sum of non-null values for datatype NUMBER.
- ✦ VAR[IANCE] — Returns the variance of non-null values for datatype NUMBER.

For example, you can list all students enrolled in courses and see a total number of students in each course:

```
SQL> BREAK ON city
SQL> COMPUTE COUNT OF lastname ON city
SQL> SELECT city, lastname, firstname
     FROM students
     ORDER BY city;
```

```
CITY            LASTNAME                       FIRSTNAME
--------------- ------------------------------ -----------
Dallas          Andrew                         Susan
*************** ------------------------------
count                                       1
New York        Hee                            John
New York        Massey                         Jane
New York        Jones                          Davey
*************** ------------------------------
count                                       3
```

The COMPUTE command has no effect unless a BREAK is defined on the column the computations are grouped by.

To remove any computations, use the CLEAR COMPUTES command:

```
SQL> CLEAR COMPUTES
```

Saving Environment Settings

Objective

- ✦ Save customizations

After you decide which settings you want to change within SQL*Plus, you can save the SET commands into a file so that every time you log in to SQL*Plus, your preferences are set. This is done using the login.sql file. (The exact filename may vary depending on your operating system; see your Oracle installation and user's manual for the exact filename.) You can list any SQL*Plus commands, SQL commands, or PL/SQL commands in this file. The file should be located in the SQL*Plus working directory.

For example, your login.sql file might contain the following:

```
SET PAUSE ON
SET PAGESIZE 50
```

You can edit this file in Notepad or vi and then save it to the working directory of SQL*Plus. Now each time you start SQL*Plus, these setting are changed from their default values to your preferred settings.

Scripts

Objective

- ✦ Create and execute script files

You may want to save frequently used SQL and SQL*Plus commands. You can use scripts to save and reexecute commands. Scripts can contain SQL statements,

SQL*Plus commands, or a mixture of the two. To create a script, you use the EDIT command or the SAVE command to create a file. Files created have a .sql extension unless you specify a different file extension. After you have created the file, you type the SQL statements or SQL*Plus commands you want to execute.

When you put a SQL statement inside a script, you must include a forward slash (/) at the end of the SQL statement.

```
SQL> EDIT myscript
SELECT *
FROM students
/
```

When you have a mixture of SQL and SQL*Plus commands, place the forward slash or a semicolon at the end of each SQL statement. This command terminator also enables you to include multiple SQL statements in one script. Each individual SQL statement in the script should end with a forward slash or a semicolon.

```
SQL> EDIT myscript
SET FEEDBACK OFF
SPOOL myoutput
SELECT *
FROM students
/
SET FEEDBACK ON
```

or

```
SQL> EDIT myscript
SET FEEDBACK OFF
SPOOL myoutput
SELECT *
FROM students;
SET FEEDBACK ON
```

Some users prefer to use a semicolon (;) at the end of a SQL statement instead of a forward slash (/). Scripts accept a semicolon as the command terminator as long as the SQL statement is not the last command in the script. When a SQL statement is the last command in the script, you must use the forward slash and not the semicolon as the command terminator for the last SQL statement; otherwise, you receive an error. When including a SQL statement in PL/SQL code, SQL statements should always end with a semicolon. The forward slash causes an error.

When you include SQL*Plus commands within scripts that affect how output is displayed, it is good practice to return the output to its default at the end of the script. If you change the LINESIZE to 200 at the start of the script, you should set it back to 80 at the end of the script. If you create a TTITLE at the start of the script, you

should turn TTITLE off at the end of the script. If you format a column with the COLUMN command, you should clear those column format settings. For example:

```
TTITLE CENTER "Student Information"
SET LINESIZE 200
COLUMN retailprice FORMAT $9,999,999.99 -
HEADING "Retail|Price"
SELECT *
FROM courses
/
TTITLE OFF
SET LINESIZE 80
COLUMN retailprice CLEAR
```

After you have created a script, you can execute it using the START or @ commands:

```
SQL> @myscript
SQL> START myscript
```

When you need to modify a script after it has been created, use the EDIT command:

```
SQL> EDIT myscript
```

PRODUCT_USER_PROFILE

Objective

- ✦ Customize the SQL*Plus environment

Certain Oracle products, including SQL*Plus, use the PRODUCT_USER_PROFILE table. This table enables you to restrict which commands can be executed within the tool. You may have users who like the functionality of SQL*Plus, but you might need to restrict the commands they can execute to protect data integrity. You create a row in the table that specifies which commands you want to disable and the user(s) for which the command is disabled.

To create the PRODUCT_USER_PROFILE table, you run the PUPBLD.SQL script as the user SYSTEM. The exact filename and file location are system dependent; see *SQL*Plus User's Guide and Reference* for the exact filename and location.

The PRODUCT_USER_PROFILE table contains the following columns:

```
SQL> DESCRIBE PRODUCT_USER_PROFILE
 Name                           Null?    Type
 ------------------------------ -------- ----
 PRODUCT                        NOT NULL CHAR(30)
 USERID                                  CHAR(30)
 ATTRIBUTE                               CHAR(240)
 SCOPE                                   CHAR(240)
 NUMERIC_VALUE                           NUMBER(15,2)
```

```
CHAR_VALUE                                CHAR(240)
DATE_VALUE                                DATE
LONG_VALUE                                LONG
```

Table 6-1 Gives a detailed description of each of the columns in the PRODUCT_USER_PROFILE table.

Table 6-1
PRODUCT_USER_PROFILE Columns

Column	*Description*
PRODUCT	Specifies the name of the product—in this case, "SQL*Plus".
USERID	Specifies the name of the user(s) for whom this command should be disabled. Wild cards (%) may be used in this column.
ATTRIBUTE	Specifies in uppercase the command to be disabled. Wild cards are not allowed in this column.
SCOPE	Ignored by SQL*Plus; set this column to NULL.
NUMERIC_VALUE	Ignored by SQL*Plus; set this column to NULL.
CHAR_VALUE	Should contain the character string "DISABLED".
DATE_VALUE	Ignored by SQL*Plus; set this column to NULL.
LONG_VALUE	Ignored by SQL*Plus; set this column to NULL.

Suppose you want to prevent all students from executing INSERT, UPDATE, and DELETE commands and specify that all student accounts have the prefix "STD"; you add three rows to the table:

```
INSERT INTO PRODUCT_USER_PROFILE (product, userid, attribute,
char_value)
VALUES ('SQL*Plus','STD%','INSERT','DISABLED');

INSERT INTO PRODUCT_USER_PROFILE (product, userid, attribute,
char_value)
VALUES ('SQL*Plus','STD%','UPDATE','DISABLED');

INSERT INTO PRODUCT_USER_PROFILE (product, userid, attribute,
char_value)
VALUES ('SQL*Plus','STD%','DELETE','DISABLED');
```

If you want to disable the SQL*Plus HOST command for all users, you add the following row. (The HOST command enables you to launch a host operating system command from the SQL*Plus command prompt.)

```
INSERT INTO PRODUCT_USER_PROFILE (product, userid, attribute,
char_value)
VALUES ('SQL*Plus','%','HOST','DISABLED');
```

Key Point Summary

SQL*Plus is a tool with a number of features that can enhance development. The capability to save commands and use variables enables you to save and reuse common SQL statements. The capability to customize your environment and format your output enables you to write reports without purchasing a separate reporting tool.

- ✦ Your last SQL statement is saved in a SQL buffer that can be accessed and modified.
- ✦ You can use substitution variables in SQL statements and specify values for those variables when you execute the SQL statement.
- ✦ SQL*Plus includes a set of commands that enables you to save commands and output to files.
- ✦ Using the SET command, you can set various options in the SQL*Plus environment that affect how output is displayed. Those commands can be saved in a file called login.sql, which are executed whenever you start SQL*Plus.
- ✦ SQL*Plus commands and SQL commands can be placed in scripts that can be saved and executed.
- ✦ SQL*Plus includes a set of commands that enable you to format your output and create formatted reports.
- ✦ The table PRODUCT_USER_PROFILE can be created to restrict the commands that users can execute within SQL*Plus.

✦ ✦ ✦

STUDY GUIDE

Now that you have learned about SQL*Plus, you should test your understanding by reviewing the assessment questions and performing the exercises that follow.

Assessment Questions

1. Which two SQL*Plus commands enable you to run the SQL command stored in the SQL buffer?

   A. START

   B. RUN

   C. GO

   D. LIST

   E. /

2. Which SQL*Plus option do you set to enable you to scroll page-by-page through the output from your SQL statement?

   A. SCROLL

   B. LINESIZE

   C. PAUSE

   D. PAGELENGTH

   E. PROMPT

3. Which SQL*Plus command enables you to specify a title at the top of each page of output?

   A. REPHEADER

   B. PAGEHEADER

   C. BTITLE

   D. TTITLE

   E. REPFOOTER

4. Which two commands run the contents of the script myfile.sql?

A. RUN myfile

B. START myfile

C. @myfile

D. EXECUTE myfile

E. GO myfile

5. Which SQL*Plus command enables you to send your output to a file so you can print it?

A. SPOOL myoutput

B. PRINT myoutput

C. SET PRINTER ON

D. SET TERMOUT ON

E. OUTPUT=myoutput

6. Which of these commands returns the description of the student table?

A. SPOOL students

B. SHOW students

C. SELECT columns FROM students

D. LIST students

E. DESCRIBE students

7. Which command prompts you to enter a value for the variable used in the following SELECT statement:

```
SELECT * FROM students WHERE studentnumber = &p_student
```

A. DEFINE p_student

B. VARIABLE p_student NUMBER

C. ACCEPT p_student NUMBER PROMPT “Enter Student Number: “

D. ENTER p_student

E. DEFINE p_student PROMPT “Enter Student Number:

8. Which SQL*Plus command displays the contents of the SQL buffer on the screen?

A. SHOW buffer

B. GET

C. LIST

D. SHOW

E. GET buffer

9. Which SQL*Plus command enables you to start a new page when the value contained in the classid column changes?

   A. SKIP PAGE on classid

   B. START PAGE on classid

   C. BREAK on classid

   D. BREAK on classid SKIP PAGE

   E. SKIP PAGE on BREAK

10. How many SQL commands are stored in the SQL buffer?

    A. 1

    B. 2

    C. 0

    D. 5

    E. 10

Scenarios

1. Your manager has just asked you to set up standards for the development team. He wants all the developers to have the same SQL*Plus option settings when they log in. You need to determine what options you need to set and where you should put the commands to specify these options. Most of the default settings can be kept, but the following changes must be made:

   A. When a SQL statement returns several pages of output, the developer should scroll through the output page-by-page.

   B. The message showing the number of lines selected should always be displayed, regardless of the number of rows actually selected.

   C. There should be 120 characters displayed on each line.

   D. A coworker comes to you with the following script because he or she is getting error messages when the script is executed. The script should prompt you to enter a student name and then return all the information in the students table about the specified student. The output should also be sent to a file called myoutput. What corrections must be made to this script?

```
ACCEPT studname PROMPT "Enter student name: "
SPOOL myoutput
SELECT *
FROM students
WHERE lastname = &studname
```

```
SPOOL OFF
3. @NL:You have been asked to create a simple report with the
output formatted as follows. Write the SQL*Plus commands to
format the output.
SELECT coursename, retailprice
FROM courses
/
COURSE LISTING
    Course
     Name           Cost
--------------- ------------
Basic SQL          $2,000.00
Advanced SQL       $2,000.00
```

Lab Exercises

Lab 6-1 Creating scripts

1. Write a SELECT statement to list all the students in class 50.
2. Using the SAVE command, save the SQL statement in a script called courserep.sql.

Lab 6-2 Using substitution variables

1. Modify the script courserep.sql so that you can specify the class number at runtime using a substitution variable.
2. Using the ACCEPT command, prompt the user to enter a class number with the following prompt:

   ```
   Please specify class for report:
   ```

Lab 6-3 Formatting output

1. Modify the script courserep.sql to include the following:
   - **A.** Use the TTITLE command to add a title at the top of the report that includes the page number on the left and the centered title "Class Registration".
   - **B.** Use the REPFOOTER command to add a center-justified message, "End of Report," at the end of the report.
2. Modify the SQL statement so all classes are listed and, using the BREAK command, show each class on a separate page.

Lab 6-4 Sending output to a file

1. Use the SPOOL command and modify courserep.sql so the output from the SQL statement is written to a file called classreg.lst.
2. Change the FEEDBACK setting to suppress the message that specifies how many rows are selected.
3. Change the LINESIZE setting so one row of data fits on one line.

Answers to Chapter Questions

Chapter Pre-Test

1. Place an ampersand (&) in front of the variable name within the SQL statement.
2. To save a SQL statement to a file, you either write the SQL statement in SQL*Plus and use the SAVE command, or you use the EDIT command to create a file and write the SQL statement in the created file.
3. To execute a SQL command or commands stored in a file, you type either @filename or START filename, where filename is the name of the file where the commands are stored.
4. You can change a number of options using the SET command. They include ARRAYSIZE, COLSEP, FEEDBACK, HEADING, LINESIZE, LONG, PAGESIZE, PAUSE, TERMOUT.
5. Commands contained in the file login.sql are executed automatically when you log in to SQL*Plus.
6. The SPOOL command enables you to send output from a SQL command to a file.
7. The PRODUCT_USER_PROFILE table allows you to limit the commands that may be executed from SQL*Plus.
8. Only one SQL command is stored in the SQL buffer.
9. The DESCRIBE command provides a description of a database table.
10. The DEFINE command creates and populates a substitution variable. The ACCEPT command creates and prompts the user to specify a value for a substitution variable.

Assessment Questions

1. **B, E** — The RUN and the / command both execute the SQL command stored in the SQL buffer. The START command executes a command file. The LIST command displays the contents of the SQL buffer. Go is not a valid SQL*Plus command.
2. **C** — The PAUSE option controls whether output pauses between pages. The LINESIZE command controls how many characters are displayed on each line. SCROLL, PAGELENGTH and PROMPT are not valid options.
3. **D** — TTITLE displays a title at the top of each page of output. BTITLE displays a footer at the bottom of each page. REPHEADER displays a title once at the start of the output. REPFOOTER displays a footer once at the end of the output. PAGEHEADER is not a valid SQL*Plus command.
4. **B, C** — Either START myfile or @myfile executes the contents of the file myfile.sql. The RUN command executes the contents of the SQL buffer. The EXECUTE command executes PL/SQL programs stored in the database. GO is not a valid SQL*Plus command.
5. **A** — SPOOL myoutput sends the output to a file myoutput.lst. SET TERMOUT ON controls whether output from a script is displayed on the screen. PRINT, SET PRINTER ON and OUTPUT are not valid SQL*Plus commands.
6. **E** — The DESCRIBE command is used to get a description of a database table. The SPOOL command sends output to a file. The SELECT statement returns the contents of the students table. The LIST command displays the contents of the SQL buffer. SHOW is not a valid SQL*Plus command.
7. **C** — The ACCEPT command prompts for a value for a substitution variable. DEFINE creates a substitution variable but will not prompt for a value. The VARIABLE command creates bind variables which must be prefixed with a colon (:) instead of an ampersand (&). ENTER is not a valid SQL*Plus command.
8. **C** — The LIST command displays the contents of the SQL buffer. The GET command fetches the contents of a command file into the SQL buffer. SHOW is not a valid SQL*Plus command.
9. **D** — BREAK on classid SKIP PAGE starts a new page when classid changes. You must specify the SKIP PAGE option at the end of a BREAK command.
10. **A** — Only one SQL command is stored in the SQL buffer.

Scenarios

1. In order to modify the SQL*Plus options from their default settings, you must edit or create a file login.sql. In order to have the output scroll, you must specify SET PAUSE ON. In order to have the message specifying the number of rows selected at all times, you must specify SET FEEDBACK ON. In order to fit 120 characters on one line, you must specify SET LINESIZE 120.

2. The DEFINE command cannot be used to prompt a user for input. Instead, the ACCEPT command must be used. There is no semicolon (;) or forward slash (/) after the SQL statement, so SPOOL OFF is considered part of the SELECT statement. You need to insert a command terminator after the SQL statement. Whenever you prompt the user to enter a character string or a date value, place single quotes around the variable name so the user does not need to put quotes around the value entered at runtime. The corrected script should look like the following:

```
ACCEPT studname PROMPT "Enter student name"
SPOOL myoutput
SELECT *
FROM students
WHERE lastname = '&studname'
/
SPOOL OFF
or
ACCEPT studname PROMPT "Enter student name"
SPOOL myoutput
SELECT *
FROM students
WHERE lastname = '&studname';
SPOOL OFF
3. @NL:You need a TTITLE command to specify the title for the
report. You need a COLUMN command to format the course name
and a COLUMN command to format the course cost. The commands
should appear as follows:

TTITLE CENTER "COURSE LISTING" SKIP 1
COLUMN coursename FORMAT A15 HEADING "Course|Name" -JUSTIFY
CENTER
COLUMN retailprice FORMAT $999,990.00 -
HEADING "Cost" JUSTIFY CENTER
```

Lab Exercises

Lab 6-1 Creating scripts

```
SQL> SELECT s.firstname, s.lastname, ce.classid
  2  FROM students s, classenrollment ce
  3  WHERE s.studentnumber=ce.studentnumber
  4  AND ce.classid = 50;

FIRSTNAME                 LASTNAME              CLASSID
------------------------- -------------------- ----------
Davey                     Jones                        50
Jane                      Massey                       50
John                      Hee                          50

SQL> SAVE courserep
Created file courserep
```

Lab 6-2 Using substitution variables

```
SQL> ED courserep

ACCEPT p_classid NUMBER -
PROMPT "Please specify class for report: "

SELECT s.firstname, s.lastname, ce.classid
FROM students s, classenrollment ce
WHERE s.studentnumber=ce.studentnumber
AND ce.classid = &p_classid
/
UNDEFINE p_classid

SQL> @courserep
Please specify class for report: 51
old   4: AND ce.classid = &p_classid
new   4: AND ce.classid =         51

FIRSTNAME                      LASTNAME             CLASSID
------------------------------ -------------------- --------
Trevor                         Smith                     51
Mike                           Hogan                     51
Gordon                         Jones                     51
```

Lab 6-3 Formatting output

```
SQL> ED courserep

TTITLE LEFT SQL.PNO CENTER "Class Registration" SKIP 2
REPFOOTER SKIP 1 CENTER "End of Report"
BREAK ON classid SKIP PAGE

SELECT s.firstname, s.lastname, ce.classid
FROM students s, classenrollment ce
WHERE s.studentnumber=ce.studentnumber
ORDER BY ce.classid
/
TTITLE OFF
REPFOOTER OFF
CLEAR BREAKS

SQL> @courserep

         1                     Class Registration

FIRSTNAME                      LASTNAME             CLASSID
------------------------------ -------------------- -------
Davey                          Jones                     50
Jane                           Massey
John                           Hee
```

```
         2                     Class Registration

FIRSTNAME                      LASTNAME              CLASSID
------------------------------ --------------------- -------
Trevor                         Smith                      51
Mike                           Hogan
Gordon                         Jones

         3                     Class Registration

FIRSTNAME                      LASTNAME              CLASSID
------------------------------ --------------------- -------
Trevor                         Smith                      53
Jane                           Massey

                                  End of Report

8 rows selected.
```

Lab 6-4 Sending output to a file

```
SQL> ED courserep

TTITLE LEFT SQL.PNO CENTER "Class Registration" SKIP 2
REPFOOTER SKIP 1 CENTER "End of Report"
BREAK ON classid SKIP PAGE

SET FEEDBACK OFF
SET LINESIZE 100
SPOOL classreg.lst

SELECT s.firstname, s.lastname, ce.classid
FROM students s, classenrollment ce
WHERE s.studentnumber=ce.studentnumber
ORDER BY ce.classid
/
SPOOL OFF

SET LINESIZE 80
SET FEEDBACK ON

TTITLE OFF
REPFOOTER OFF
CLEAR BREAKS
SQL> @courserep
```

```
         1                          Class Registration

FIRSTNAME                      LASTNAME             CLASSID
------------------------------ -------------------- --------
Davey                          Jones                      50
...
                                        End of Report


SQL> ED classreg.lst
         1                          Class Registration

FIRSTNAME                      LASTNAME             CLASSID
------------------------------ -------------------- --------
Davey                          Jones                      50
Jane                           Massey
John                           Hee

         2                          Class Registration

FIRSTNAME                      LASTNAME             CLASSID
------------------------------ -------------------- --------
Trevor                         Smith                      51
Mike                           Hogan
Gordon                         Jones

         3                          Class Registration

FIRSTNAME                      LASTNAME             CLASSID
------------------------------ -------------------- --------
Trevor                         Smith                      53
Jane                           Massey

                                          End of Report
```

CHAPTER 7

Creating and Managing Oracle Database Objects

EXAM OBJECTIVES

- Creating and managing tables
  - Create tables
  - Alter table definitions
  - Drop, rename, and truncate tables
- Including constraints
  - Describe constraints
  - Create and maintain constraints
- Creating views
  - Describe a view
  - Create a view
  - Retrieve data through a view
  - Insert, update, and delete data through a view
  - Drop a view
- Oracle data dictionary
  - Describe the data dictionary views a user may access
  - Query data from the data dictionary
- Other Database Objects
  - Describe database objects and their uses
  - Create, maintain, and use sequences
  - Create and maintain indexes
  - Create private and public synonyms

CHAPTER PRE-TEST

1. What must be specified when a table is created?
2. When a DEFAULT is defined on a database column, which operations and conditions cause the DEFAULT to be used?
3. How many indexes should you create on a table?
4. Which clause that was previously invalid is now allowed in a view definition? Why is this useful?
5. Suppose you have three tables called Customers, Orders, and Suppliers. Assuming each table has a column to uniquely identify a customer, order, or supplier, how many sequences should you create?
6. Who is allowed to create PUBLIC synonyms, by default?
7. Why would you cache sequences? Why not?
8. What is the difference between a PRIMARY KEY and UNIQUE constraint?
9. If you do not specify a name for a constraint, what name is assigned to it by Oracle?
10. What is the difference between a column of CHAR datatype and a column of the NCHAR datatype?
11. What must an Oracle object name start with? What are the other rules for naming Oracle objects?

✦ Answers to these questions can be found at the end of the chapter. ✦

Up to this point you have been shown how to access data from tables within the database. However, before you can store data in a database, you have to create the necessary database objects to store and manage your data. These objects include the tables that will be used to store information about the entities that you are keeping track of, such as Students, Customers, Inventory, and so on. In order for the data to make sense, to enforce business rules, and to define the relationships between the various tables, you may also need to define constraints on the columns in those tables. Indexes may enable you to speed the retrieval of data in your database as it grows. To ensure duplicate values are not placed in primary key columns that you define, you may decide to use sequences to generate values that will be unique. For reporting, security, or other purposes, you may need to define views to make it easier to perform the necessary data retrieval. Finally, when the table names are long or when you want to more easily access database objects created by other users, you may decide to make use of synonyms.

In this chapter, you learn how to create and manage the objects that make up your database. We show you how to create, alter, and drop tables, as well as perform other actions to manage the data that you store in them. We then present a discussion on using constraints and how to manage them. Finally, the creation of other database objects such as views, indexes, sequences, and synonyms is described.

The Ground Rules for Creating Objects

Before creating any objects in an Oracle database, it is important to understand the language elements and rules that apply to the process. This includes the statements that are used to create and manage database objects, the rules for naming database objects, and some general information about permissions.

Data Definition Language (DDL)

When creating and managing any object in an Oracle database, you use the Data Definition Language (DDL) elements of the SQL language. DDL has three principal statements:

- **CREATE:** The CREATE statement adds a new object to the database. You use it to create new tables, views, stored procedures, and other objects. In order for the CREATE command to succeed, no other object with the same name can exist in the schema .
- **ALTER:** The ALTER statement is used to change the characteristics of tables, indexes, and other objects in the database. The ALTER statement does not apply to all objects in the database.
- **DROP:** The DROP statement is used to remove objects from the database. When used on tables, it also gets rid of any data that existed in the table.

Oracle naming conventions

When creating any object in Oracle, it is important to know the valid characters and other rules for naming objects. If you attempt to create an object that does not follow the required naming conventions, Oracle generates an error.

Names of objects in Oracle must follow these rules:

- ✦ All object names must begin with a letter (A–Z, a–z).
- ✦ An object name may be from 1 to 30 characters in length.
- ✦ An object name can contain letters (A–Z, a–z), numbers (0–9), and the underscore character (_), dollar sign symbol ($), or pound sign symbol (#). Although $ and # are supported, their use is strongly discouraged.
- ✦ An object name cannot be the same as any Oracle reserved word. Oracle reserved words include most commands (for example, SELECT, INSERT, DROP, GRANT), as well as the name of functions, and so on. The complete list of Oracle reserved words can be found in the *Oracle 8i SQL Reference* manual.
- ✦ An object name must not be the same as the name of another object owned by the same user. When you attempt to create a duplicate object, Oracle generates an error. This does not mean, however, that you cannot have the same column name in more than one table — you can. This is because the fully qualified name of any object in Oracle has many subcomponents (see the discussion that follows).

The names of objects in Oracle are case insensitive, so if you use uppercase only, lowercase only, or mixed case when you create an object, Oracle always returns the object name in uppercase.

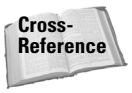
Even though Oracle object names are not case sensitive, character data in Oracle is always case sensitive. This means that the statement "SELECT * FROM Courses" returns the same results as "SELECT * FROM courses". However, the statement "SELECT * FROM Courses WHERE CourseName LIKE '%SQL%'" does not return the same data as "SELECT * FROM Courses WHERE CourseName LIKE '%sql%'".

When naming objects, it is always a good idea to use descriptive names so that it is easier to understand what the object is supposed to represent. This means that it is not a good idea to name tables T1, T2, T3, and so on, but rather you should name them, for example, Courses, Instructors, and Students. Using plural nouns for the names of tables is a recognized practice in the database world.

Fully qualified object names

Some user must own every object in Oracle. Oracle creates two users when the database is created — SYS and SYSTEM. The user SYS owns the data dictionary for the database, and the user SYSTEM has full control of any object in the database. For your database objects, you need to create another user who will become the

owner of any object that he or she creates. It is possible to have many owners of database objects in a single database, although this practice is discouraged because it may become too difficult to manage.



For more information on creating users and assigning permissions to users, see Chapter 8, "Configuring Security in Oracle Databases."

When a user creates an object, he or she becomes its owner. Oracle assigns a fully qualified name to every object in the database. The fully qualified name must be unique in the database. A fully qualified name is written as follows:

```
<ownername>.<objectname>.<objectname>
```

For example, the CustomerID column of the Customers table owned by Sam is identified by the following fully qualified name:

```
sam.customers.customerid
```

As mentioned previously, the names of objects must be unique within the schema of the user that owns them. For example, if user Bob creates a table called Students and user Sally creates a table called Students, there is no naming conflict because the fully qualified names of the tables are as follows:

```
bob.students
sally.students
```

Similarly, having the same column name in two tables owned by the same user does not create a conflict. In fact, it is recommended for columns that relate the two tables together because it makes it easy to see on which values the two tables are related. For example, the LocationID column can exist in both the ScheduledClasses and Locations tables owned by the user Student. The fully qualified names are:

```
Student.Locations.LocationID
Student.ScheduledClasses.LocationID
```

When referencing an object in Oracle, if any part of the fully qualified name is not specified, that portion defaults to a value preset by Oracle. The rules are:

- ✦ If the name of the owner is omitted, it is assumed to be the current user. For example, if Bob issues the command "SELECT * FROM Students", Oracle assumes this to mean "SELECT * FROM Bob.Students".
- ✦ If the object (for example, Bob.Students) does not exist, Oracle checks to see whether the user has been given permissions to another object with the same name in another schema (that is, whether Bob has been granted permission to David's Students table). If this is not the case, Oracle returns an error; if it is, Oracle ensures that the user has not been granted permissions on objects by the same name in several schemas (that is, that Bob has permissions to SELECT from David.Students and Cheryl.Students). If this is true, an error is reported; otherwise, the statement is processed normally.

- ✦ The ownership of columns specified in a SELECT or UPDATE statement is checked to ensure uniqueness of column names for all tables referenced in the statement. If this is not the case, Oracle generates an error and requires that the problem be corrected before the statement can be processed. You then must qualify the name.
- ✦ When creating objects, unless otherwise specified, the object is created in the user's schema and the user creating the object becomes the owner automatically.
- ✦ To create an object in someone else's schema, you must have permissions to do so, and you must qualify the name of the object to be created with the name of the user in whose schema you want to create the object.

The bottom line is that Oracle always assumes that you own every object you are creating or manipulating. If this is not true, to avoid ambiguity, always qualify the name of the object with the owner, or create a synonym.

Creating and Managing Tables

- ✦ Create tables
  - Alter table definitions
  - Drop, rename, and truncate tables

If the purpose of every database is to store and manage data, then the purpose of every table in the database is to act as the primary vehicle for the storage of data. A database without tables is not a database. It is, therefore, critical that you know how to create and manage tables in your databases so that you are able to logically store the data for your organization in a way that makes sense.

One of the most critical elements of creating tables is knowing what tables to create and what pieces of information they should hold. This task is typically the job of the database designer. This individual determines, based upon the business requirements outlined by the organization, how many tables to create, what data they should hold, and the relationships that exist between them. The process of designing databases can be a lengthy one and, when done properly, should be a very lengthy one taking about 80 percent of the time from when the decision to implement a database is made to when the database becomes operational. Because of the complexity of database design, it is not discussed here. It is important to remember that you should not simply create tables because you think you need one—you should create tables because they are needed as part of an overall design.

Learning to design databases

Although the "Introduction to Oracle: SQL & PL/SQL" exam does not test your knowledge of database design concepts and theory, if you want to be a more well rounded individual in the database world, an understanding of design is necessary. A number of useful titles are available that explain the process of designing a database. They include Michael J. Hernandez, *Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design* (Addison Wesley, 1997), which is a good book for designing most databases. If you are looking at working with data warehouses, Ralph Kimball et al., *The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses* (John Wiley & Sons, 1998) provides useful information on the design process for data warehouses.

For books that are more specific to Oracle, a helpful data warehouse guide for the Oracle world is Gary Dodge et al., *Essential Oracle8i Data Warehousing: Designing, Building, and Managing Oracle Data Warehouses* (John Wiley & Sons, 2000). Another useful reference dealing with Oracle database design is David Ensor and Tim Stevenson, *Oracle Design* (O'Reilly & Associates, 1997). The information presented in this last title, though dealing with Oracle 7 and Oracle 8 primarily, still applies to Oracle 8*i*.

The CREATE TABLE statement

✦ Create tables

The statement used to create tables in Oracle is the CREATE TABLE DLL command. The syntax of the CREATE TABLE statement is:

```
CREATE [GLOBAL TEMPORARY] TABLE [schema.]tablename
   (columnname datatype [NULL | NOT NULL] [DEFAULT expression]
        [, ...]);
```

The CREATE TABLE syntax outlined here is not the complete syntax. A great many more options can be specified such as where the table will be stored, the characteristics of storage, whether or not it is partitioned, as well as constraints. You will be tested on the preceding syntax in the "Introduction to Oracle: SQL & PL/SQL" exam. For a complete syntax of the CREATE TABLE command refer to the *Oracle 8i SQL Reference* manual.

For example, to create the table courses in your own schema, you issue the command:

```
CREATE TABLE Courses
   (CourseID number(5) NOT NULL,
    CourseName varchar2(200) NOT NULL,
    ReplacesCourse number(5) NULL,
    RetailPrice number(9,2) NULL,
    Description varchar2(2000) NULL);
```

In creating a table, you must specify a table name and at least one column with a valid datatype and size. NULL or NOT NULL are not required but are recommended to ensure that you are aware whether or not a value must be entered for the column.

Although it is not necessary to place each column definition on a separate line for the statement to work (each column definition need be separated only by a comma), doing so improves the readability of the statement and makes it easier to see specific information about each column. You should follow this rule when creating your own tables.

Datatypes available in Oracle

Each column in an Oracle table must have at least two properties: It must have a name unique within the table and a valid datatype. Datatypes in Oracle enable you to specify what types of data (for example, numbers, characters, dates, or binary large objects — BLOBs) can be stored in the column. The datatypes that are available include the standard Oracle scalar datatypes listed in Table 7-1 or a user-defined datatype that is based upon one of the scalar datatypes or another user-defined datatype.

The "Introduction to Oracle: SQL & PL/SQL" exam does not test your knowledge of the creation or usage of user-defined datatypes.

Table 7-1
Scalar Datatypes

Datatype	*Description*
VARCHAR2(size)	Variable-length character string having maximum length-size bytes. The maximum size is 4,000, and the minimum is 1. You must specify the size for VARCHAR2.
NVARCHAR2(size)	Variable-length character string having maximum length-size characters or bytes, depending on the choice of the National Language character set. The maximum size is determined by the number of bytes required to store each character, with an upper limit of 4,000 bytes. You must specify the size for NVARCHAR2.
NUMBER(p,s)	Number having precision *p* and scale *s*. The precision p can range from 1 to 38. The scale s can range from –84 to 127. Precision determines the total numbers that can be stored. Scale determines the number of decimal places allowed.
LONG	Character data of variable length up to 2GB, or 231 –1 bytes. LONG datatype columns were used in previous versions of Oracle to support large character data; however, they should no longer be used and are strongly discouraged. CLOB and NCLOB datatype columns are the recommended way to store large amounts of character data.

Datatype	*Description*
DATE	Valid date range from January 1, 4712 BC to December 31, 9999 AD.
RAW(size)	Raw binary data of length-size bytes. Maximum size is 2,000 bytes. You must specify size for a RAW value. This datatype is included for backward compatibility and should not be used.
LONG RAW	Raw binary data of variable length up to 2GB. This datatype is included for backward compatibility and should not be used. To store large amounts of binary data, BLOB datatype columns are recommended.
ROWID	Hexadecimal string representing the unique address of a row in its table. This data type is primarily for values returned by the ROWID pseudo-column.
UROWID [(size)]	Hexadecimal string representing the logical address of a row of an index-organized table. The optional size is the size of a column of type UROWID. The maximum size and default is 4,000 bytes.
CHAR(size)	Fixed-length character data of length-size bytes. Maximum size is 2,000 bytes. Default and minimum size is 1 byte.
NCHAR(size)	Fixed-length character data of length-size characters or bytes, depending on the choice of National Language character set. Maximum size is determined by the number of bytes required to store each character, with an upper limit of 2,000 bytes. Default and minimum size is 1 character or 1 byte, depending on the character set.
CLOB	A character large object containing single-byte characters. Both fixed-width and variable-width character sets are supported, both using the database character set. Maximum size is 4GB.
NCLOB	A character large object containing multibyte characters. Both fixed-width and variable-width character sets are supported, both using the National Language character set of the database. Maximum size is 4GB.
BLOB	A binary large object. Maximum size is 4GB.
BFILE	Contains a locator to a large binary file stored outside the database. Enables byte stream I/O access to external large objects (LOBs) residing on the database server. Maximum size is 4GB.

Notice in Table 7-1 that for character data, you have both NCHAR and CHAR columns, as well as NVARCHAR2 and VARCHAR2. This is because Oracle introduced support for National Language character sets in Oracle 8. In essence, this allows you to store data in CHAR and VARCHAR2 columns with one set of characters (Western European, for example) and data in NCHAR and NVARCHAR2 columns with a different set of characters (Japanese, for example). This allows the same

database to house different character data instead of requiring a second database to hold the same information. Typically, in real work environments, the character set of a database and the National Language character set of a database are either the same or very similar.

The "Introduction to Oracle: SQL & PL/SQL" exam does not test your knowledge of National Language characters or how to create a database to support different National Language character sets. It is sufficient to be aware of the two different sets of datatypes and which character set corresponds to which.

The GLOBAL TEMPORARY clause

In Oracle 8*i*, a new feature was introduced that enabled you to create a table whose definition would remain the data dictionary but whose data would persist only for a session or a transaction. The GLOBAL TEMPORARY clause in the CREATE TABLE statement indicates that you wish to create a temporary table.

GLOBAL TEMPORARY tables are useful when you need a temporary place to store information before committing it to the database. For example, if your database is used to enter orders, you can place information about the order in a temporary set of tables while the order process is taking place. When the customer confirms the order, you can issue a command that inserts data from the temporary tables to the permanent Orders and OrderDetails tables. This way, you do not insert data into the "real" tables until customers are sure that they want to place the order. Without the use of temporary tables, you would have to insert and then delete the information, which can slow down other users.

A GLOBAL TEMPORARY table can be assigned permissions just like any other table; however, each user that inserts or updates data in such a table sees only their own information. This is because Oracle assumes, when you create a GLOBAL TEMPORARY table, that its contents are held on a per session basis (that is, every connection to the database will have different data). The data in one session's temporary table is visible only to that session.

To create a GLOBAL TEMPORARY table to hold class registration information as it is being processed, you issue the following command:

```
CREATE GLOBAL TEMPORARY TABLE TempClassEnrollment
   (ClassID number(5) NOT NULL,
    StudentNumber number(5) NOT NULL,
    Status char(10) NOT NULL,
    EnrollmentDate date NOT NULL,
    Price number(9,2) NOT NULL,
    Grade char(4) NULL,
    Comments varchar2(2000) NULL);
```

The DEFAULT clause

When creating a table, you can specify the value that should be assigned to a column if no value is provided when a row is inserted into the table. This is particularly useful on those columns that have NOT NULL specified for them because you are now able to ensure that a value is assigned if the user fails to specify one on insert. Unless a DEFAULT is specified for a column, Oracle always assigns NULL to the column if no value is specified. If this behavior is not what you desire, specify a DEFAULT for the column.

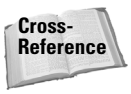
To assign a value to the EnrollmentDate column of the ClassEnrollment table, you use the following CREATE TABLE statement:

```
CREATE TABLE ClassEnrollment
   (ClassID number(5) NOT NULL,
   StudentNumber number(5) NOT NULL,
   Status char(10) NOT NULL,
   EnrollmentDate date NOT NULL DEFAULT SYSDATE,
   Price number(9,2) NOT NULL,
   Grade char(4) NULL,
   Comments varchar2(2000) NULL);
```

This assigns the value of SYSDATE (a system function returning the current date and time) to the EnrollmentDate column if another value was not specified at the time the row was inserted.

It is important to remember that DEFAULTs apply only when a row is *INSERTed* into a table. They do not apply when an UPDATE statement is used to change data for the column. It is, therefore, possible that, unless NOT NULL is specified or a trigger exists to prevent the action, a user can modify the value of the column and set it to NULL by using the UPDATE statement after the INSERT has taken place.

The DEFAULT clause enables you to specify a literal value, a simple expression, or a SQL function such as SYSDATE. When using expressions, it is important to note that you may not reference other columns or tables in the expression, nor may you reference pseudo-columns such as ROWID. When more functionality is required to assign the column a value, you must create a trigger to perform the action.



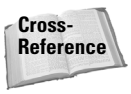
For more information on how to create triggers and how they work, refer to Chapter 13, "Introduction to Stored Programs."

Finally, although it may be obvious, the datatype of the expression must match the datatype of the column. For example, if the column is a date datatype and the DEFAULT expression returns a character value, you must use the TO_DATE system function to convert the result of an expression to a date; otherwise, the assignment of the default will fail.

Creating a table using a subquery

A variation of the CREATE TABLE syntax enables you to create a table and populate it with data that is the result of a subquery specified in the CREATE TABLE command. The syntax to create a table using a subquery is as follows:

```
CREATE TABLE tablename
  [(columnname, columnname, ...)]
  AS subquery;
```



For more information on subqueries, please refer to Chapter 4, "Advanced SELECT Statements."

When creating a table from a subquery, you can also specify the names of columns in the CREATE TABLE portion of the statement, and any columns returned by the subquery are given the names specified there. You cannot specify the datatype of the column because it is determined by the result of the subquery.

Another method, and one that is more accepted, is to alias the column in the SELECT statement of the subquery. Aliasing the name of the column in the subquery actually makes the statement more readable.

The rules for creating tables by using a subquery are straightforward:

- ✦ When you specify the names of columns in the CREATE TABLE portion of the command, the table is created with the specified column names.
- ✦ The column definition in the CREATE TABLE portion can contain only the column name and a DEFAULT.
- ✦ When you specify the names of columns in the CREATE TABLE portion of the command, the number of columns returned by the subquery must be equal to the number of columns specified. If they differ, the command fails.
- ✦ When no column names are specified in the CREATE TABLE portion of the command, the number of columns and all column names are taken from the subquery.
- ✦ The names of columns in the subquery must adhere to Oracle naming conventions or be aliased to conform to those conventions.
- ✦ Any rows retrieved by the subquery are inserted into the table after it is created.

Creating tables using subqueries is a very useful feature in Oracle because it enables you to duplicate some or all of the data in one table under a different table name. In organizations that have large amounts of data, the CREATE TABLE ... AS SELECT syntax can be used to create historical records of information and reduce

the size of production tables so that data retrieval works faster. For example, if you want to create a table with all class enrollments that are more than two years old, you can issue the following command:

```
CREATE TABLE ClassEnrollmentArchive AS
   SELECT * FROM ClassEnrollment
     WHERE EnrollmentDate < (SYSDATE-730);
```

You can also use it to create a table of the amount of revenue generated on a per class basis for the last two years and then use that for reporting. To create the table, you issue a command similar to the following:

```
CREATE TABLE EnrollmentRevenueSummary AS
     SELECT ClassID, SUM(Price) as Revenue
           FROM ClassEnrollment
     WHERE EnrollmentDate
          BETWEEN (SYSDATE-730) AND SYSDATE
     GROUP BY ClassID
     ORDER BY ClassID;
```

Notice that the SUM(Price) aggregate expression must be aliased as Revenue in this case. This is because a table created as a result of a subquery must have valid Oracle names for columns. SUM(Price) cannot be a column name because the open parenthesis and close parenthesis characters are invalid in a column name. When creating tables from a subquery, it is important to ensure that the resulting column names follow proper naming convention rules.

Another use of creating tables from a subquery is to duplicate the structure of an existing table without adding the data at the same time. This is done by adding a WHERE clause to the subquery that always evaluates to FALSE, as in the following example:

```
CREATE TABLE NewStudents AS
     SELECT * FROM Students
     WHERE 0=1;
```

The NewStudents table will have no rows because the expression 0=1 always evaluates to FALSE and does not return any rows.

Getting information on tables from the data dictionary

Objective

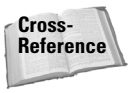
- ✦ Describe the data dictionary views a user may access
- ✦ Query data from the data dictionary

Oracle provides several ways to get information on tables from the data dictionary. One of these methods is by querying USER_ views. Several USER_ views provide information on tables. Two that are the most useful are USER_TABLES, which lists

the names and storage characteristics of tables that a user owns, and USER_TAB_COLUMNS, which lists the columns of all tables in the user's schema and their properties. Oracle actually has three sets of data dictionary views:

- **USER_ views:** These views enable users to get information on objects that are in their schemas (that is, objects that they have created and own).
- **ALL_ views:** These views enable users to get information on objects that they own or that they have been given access to. The ALL_ views contains a superset of the information presented in the USER_ views and enables users to find out what other objects they are allowed to reference or manipulate in the database.
- **DBA_ views:** The DBA_ views are designed to be used by the database administrator (DBA) and provide a full set of information for objects in the database (that is, any object created by any user). Ordinary users do not have access to these views because special privileges are needed to SELECT from them.

To obtain information on the objects that you own, you query the USER_ views. If you wanted to find out information about objects that you have been granted permissions to, you can make use of ALL_ views. Unless you are a DBA, you do not ordinarily have access to DBA_ views.



Appendix F, "Data Dictionary Views," provides a listing and the structure of most of the USER_ views that you will use for the exam.

To get a listing of all tables that you own, you can issue the following command:

```
SQL> SELECT table_name FROM USER_TABLES;

TABLE_NAME
------------------------------
BATCHJOBS
CLASSENROLLMENT
COURSEAUDIT
COURSES
COURSES_TEMP
INSTRUCTORS
LOCATIONS
SCHEDULEDCLASSES
STUDENTS

9 rows selected.

SQL>
```

To get a listing of all tables that you have been granted access to, including the tables that you own, you can issue the command:

```
SQL> SELECT owner, table_name FROM ALL_TABLES
  2  ORDER BY owner, table_name;

OWNER                          TABLE_NAME
------------------------------ --------------------------------
MDSYS                          CS_SRS
MDSYS                          MD$DICTVER
MDSYS                          OGIS_SPATIAL_REFERENCE_SYSTEMS
MTSSYS                         MTS_PROXY_INFO
STUDENT                        BATCHJOBS
STUDENT                        CLASSENROLLMENT
STUDENT                        COURSEAUDIT
STUDENT                        COURSES
STUDENT                        COURSES_TEMP
STUDENT                        INSTRUCTORS
STUDENT                        LOCATIONS
STUDENT                        SCHEDULEDCLASSES
STUDENT                        STUDENTS
SYS                            AUDIT_ACTIONS
SYS                            DUAL
SYS                            PSTUBTBL
SYS                            STMT_AUDIT_OPTION_MAP
SYS                            SYSTEM_PRIVILEGE_MAP
SYS                            TABLE_PRIVILEGE_MAP
SYSTEM                         DEF$_TEMP$LOB
SYSTEM                         HELP

21 rows selected.

SQL>
```

You can also query the USER_TAB_COLUMNS view to get a listing of all columns for a table that you own, for example:

```
SQL> SELECT table_name, column_name, data_type, data_length
  2  FROM USER_TAB_COLUMNS
  3  WHERE table_name = 'COURSES'
  4  ORDER BY column_id;

TABLE_NAME           COLUMN_NAME          DATA_TYPE  DATA_LENGTH
-------------------- -------------------- ---------- -----------
COURSES              COURSENUMBER         NUMBER              22
COURSES              COURSENAME           VARCHAR2           200
COURSES              REPLACESCOURSE       NUMBER              22
COURSES              RETAILPRICE          NUMBER              22
COURSES              DESCRIPTION          VARCHAR2          2000

5 rows selected.

SQL>
```

Oracle SQL*Plus also offers another mechanism to find the structure of a single table, as well as other objects — the DESCRIBE command. To retrieve the structure of the Courses table without having to formulate a SQL query as shown previously, you can issue the following command in SQL*Plus:

```
SQL> DESC Courses
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 COURSENUMBER                              NOT NULL NUMBER(38)
 COURSENAME                                NOT NULL VARCHAR2(200)
 REPLACESCOURSE                                     NUMBER(38)
 RETAILPRICE                               NOT NULL NUMBER(9,2)
 DESCRIPTION                                        VARCHAR2(2000)

SQL>
```

The DESCRIBE command, whose shorthand is DESC, is useful in determining the structure of a table, view, or other database objects.

Another view, USER_CATALOG, simply returns the name of an object (that is, a table or a view) that contains data and its type (table, view, or synonym). USER_CATALOG is an ANSI-standard view and provides a brief listing of all objects that you own. Its corresponding ALL_CATALOG view provides information on all tables, views, and synonyms that you have been granted access to.

The USER_CATALOG view also has a synonym defined. Therefore, the following commands are interchangeable and result in the same output:

```
SELECT * FROM USER_CATALOG;
SELECT * FROM CAT;
```

Issuing either of the preceding commands provides the following result:

```
SQL> SELECT * FROM CAT;

TABLE_NAME           TABLE_TYPE
-------------------- -----------
BATCHJOBS            TABLE
CLASSENROLLMENT      TABLE
COURSEAUDIT          TABLE
COURSES              TABLE
COURSES_TEMP         TABLE
INSTRUCTORS          TABLE
LOCATIONS            TABLE
SCHEDULEDCLASSES     TABLE
STUDENTS             TABLE

9 rows selected.

SQL>
```

Querying the ALL_CATALOG view provides a rather lengthy list of objects that you have been granted access to, as well as those that you own. The output (truncated here because of space limitations) is similar to the following:

```
SQL> SELECT * FROM ALL_CATALOG;

OWNER           TABLE_NAME                TABLE_TYPE
--------------- ------------------------- -----------
SYS             ALL_ALL_TABLES            VIEW
PUBLIC          ALL_ALL_TABLES            SYNONYM
SYS             ALL_ARGUMENTS             VIEW
PUBLIC          ALL_ARGUMENTS             SYNONYM
SYS             ALL_ASSOCIATIONS          VIEW
PUBLIC          ALL_ASSOCIATIONS          SYNONYM
SYS             ALL_CATALOG               VIEW
PUBLIC          ALL_CATALOG               SYNONYM
SYS             ALL_CLUSTERS              VIEW
PUBLIC          ALL_CLUSTERS              SYNONYM
SYS             ALL_CLUSTER_HASH_EXPRESSI VIEW
PUBLIC          ALL_CLUSTER_HASH_EXPRESSI SYNONYM
SYS             ALL_COLL_TYPES            VIEW
PUBLIC          ALL_COLL_TYPES            SYNONYM
SYS             ALL_COL_COMMENTS          VIEW
PUBLIC          ALL_COL_COMMENTS          SYNONYM
SYS             ALL_COL_PRIVS             VIEW
PUBLIC          ALL_COL_PRIVS             SYNONYM
SYS             ALL_COL_PRIVS_MADE        VIEW
PUBLIC          ALL_COL_PRIVS_MADE        SYNONYM
SYS             ALL_COL_PRIVS_RECD        VIEW
PUBLIC          ALL_COL_PRIVS_RECD        SYNONYM
SYS             ALL_CONSTRAINTS           VIEW
PUBLIC          ALL_CONSTRAINTS           SYNONYM
SYS             ALL_CONS_COLUMNS          VIEW
PUBLIC          ALL_CONS_COLUMNS          SYNONYM
SYS             ALL_CONTEXT               VIEW
PUBLIC          ALL_CONTEXT               SYNONYM
SYS             ALL_DB_LINKS              VIEW

... many more rows follow ...
```

The ALTER TABLE command

When you need to change your table definition, by adding or removing a column, modifying the length or datatype of a column, or adding a DEFAULT to a column, you use the ALTER TABLE command.

Adding columns to a table

To add a column to an existing table, the syntax of the ALTER TABLE command is as follows:

```
ALTER TABLE tablename
  ADD (columnname datatype [DEFAULT expr]
              [, columnname datatype ...] ...);
```

For example, if you want to add a new column called BirthDate to the Students table, you issue the following command:

```
ALTER TABLE Students
  ADD (BirthDate date NULL);
```

After a column is added to a table, it becomes the last column in its definition. When issuing a DESCRIBE command on the table, it is listed after all other existing columns, as shown here:

```
SQL> DESC Students
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 STUDENTNUMBER                             NOT NULL NUMBER(38)
 SALUTATION                                         CHAR(4)
 LASTNAME                                  NOT NULL VARCHAR2(30)
 FIRSTNAME                                 NOT NULL VARCHAR2(30)
 MIDDLEINITIAL                                      VARCHAR2(5)
 ADDRESS1                                           VARCHAR2(50)
 ADDRESS2                                           VARCHAR2(50)
 CITY                                               VARCHAR2(30)
 STATE                                              CHAR(2)
 COUNTRY                                            VARCHAR2(30)
 POSTALCODE                                         CHAR(10)
 HOMEPHONE                                          CHAR(15)
 WORKPHONE                                          CHAR(15)
 EMAIL                                              VARCHAR2(50)
 COMMENTS                                           VARCHAR2(2000)
 BIRTHDATE                                          DATE

SQL>
```

Adding a column to an existing table does not physically change the data in the database. Oracle applies the default value, if specified, to the column for all existing data and assigns the column any values that a user assigns to it for a particular row. When querying the database for the value of the BirthDate column, the output includes those rows that have the value and those assigned the DEFAULT (or NULL when no DEFAULT is defined), as shown here:

```
SQL> SELECT studentnumber, lastname, firstname, birthdate FROM Students;

STUDENTNUMBER LASTNAME             FIRSTNAME            BIRTHDATE
------------- -------------------- -------------------- ---------
         1000 Smith                John
         1001 Jones                Davey
         1002 Massey               Jane
         1003 Smith                Trevor
         1004 Hogan                Mike
         1005 Hee                  John
         1006 Andrew               Susan
         1007 Holland              Roxanne
         1008 Jones                Gordon
         1009 Colter               Sue
         1010 Patterson            Chris                18-APR-76

11 rows selected.

SQL>
```

Modifying existing columns in a table

The ALTER TABLE command also enables you to modify existing columns in a table. The syntax of the command is:

```
ALTER TABLE tablename
  MODIFY (column datatype [DEFAULT expression] );
```

Using the ALTER TABLE command you can:

- ✦ Increase the size or precision of a numeric column. For example, you can change a column from NUMBER(9,2) to NUMBER(11,2).
- ✦ Reduce the size of a column, such as making a VARCHAR(2) column smaller. You are able to do this only when the table contains no rows. For this reason, this action is not usually performed; dropping and recreating the table is more common.
- ✦ Change a column from one datatype to another when all rows for the column contain only NULL values. Again, because this is not a very typical situation, it is not likely that you ever actually will change a table's datatype without recreating it.
- ✦ Change a column from VARCHAR2 to CHAR, or vice versa if you do not change the size. When you do want to increase, but not decrease, the size, the table may contain data, and the column may be populated. When you wish to change the size as well as the datatype, the column can contain only NULL values.
- ✦ Add a DEFAULT to a column. When you add the DEFAULT, all new rows added to the table where a value is not specified during the INSERT operation are assigned the DEFAULT value. The values of all existing rows are not changed.

The following example changes the size of the RetailPrice column in the Courses table:

```
ALTER TABLE Courses
  MODIFY (RetailPrice number(11,2)0;
```

Removing a column from a table

Oracle 8*i* introduces a new feature — the ability to drop columns that are no longer needed from a table. You can do so using a single-step process with the ALTER TABLE ... DROP COLUMN command or by first marking the column UNUSED.

Marking columns UNUSED

Marking a column UNUSED no longer makes it available to any application using the table. For all intents and purposes, the columns marked UNUSED no longer exist and cannot be referenced in SELECT, INSERT, or UPDATE statements. However, the advantage of marking a column UNUSED is that Oracle does not physically rebuild the table to remove the column but only flags it as UNUSED. This is useful in those cases where the table whose column you no longer require has large amounts of data. Physically removing the column may take a long time because the entire table must be rebuilt, but marking the column as UNUSED is a quick operation.

Two variations of the syntax can be used to mark a column as UNUSED. One way that can be used is:

```
ALTER TABLE tablename
   SET UNUSED (columnname);
```

Another way to mark a column as UNUSED is to use the following syntax:

```
ALTER TABLE tablename
   SET UNUSED COLUMN columnname;
```

The end result of using either syntax is the same: The column no longer is available, and the table is not physically reorganized.

Dropping a column from a table

Another way to remove a column from a table is to drop it. The syntax of the command is as follows:

```
ALTER TABLE tablename
   DROP COLUMN columnname;
```

When this command is issued, Oracle physically rebuilds the table to not include the column structure in any row. On tables with large amounts of data, this process can take a long time.

If you previously marked one or more columns as UNUSED, you can also drop them all at the same time by using the following version of the ALTER TABLE command:

```
ALTER TABLE tablename
   DROP UNUSED COLUMNS;
```

This command drops any columns that have been previously marked as UNUSED in the table and physically reorganizes the table itself.

It's important to note that when you issue a DROP COLUMN command for any column in a table, if other columns in the same table have already been marked as UNUSED, they also are dropped. While this may seem like a problem, it really is not. After a column is marked as UNUSED, it is not longer available and cannot be recovered. For all intents and purposes, the column is gone. By dropping those columns marked UNUSED at the same time that you explicitly drop a single column, Oracle is merely performing housecleaning and ensuring that it no longer has to remove from the data it retrieves any references to those UNUSED columns.

Being able to mark columns as UNUSED first and then drop them all at the same time is helpful when you need to drop more than one and you do not want to have to rebuild the table physically each time.

Rules for dropping columns or marking them UNUSED

When marking columns as UNUSED or issuing the DROP COLUMN variant of the ALTER TABLE command, you need to ensure that the following rules are followed:

- ✦ The table whose columns are being marked as UNUSED or dropped must have at least one column after the operation is complete. In other words, you cannot mark a column UNUSED or drop it if it is the last column in the table.
- ✦ When using the DROP COLUMN syntax, you cannot drop more than one column at the same time. The ALTER TABLE ... DROP COLUMN command must issued once for each column to be dropped. A better way to do this is to mark the columns as UNUSED and then issue the following command:

  ```
  ALTER TABLE tablename
     DROP UNUSED COLUMNS;
  ```

- ✦ After a column is marked as UNUSED or dropped, it cannot be recovered. Either variation on removing the column will cause it to be logically removed from the table definition. The DESCRIBE command does not return UNUSED or dropped columns.
- ✦ If the column contains data, the data is no longer accessible. It is always recommended that a table whose columns are being marked as UNUSED or dropped be backed up prior to doing so, or another copy of the table with a different name be created using the CREATE TABLE ... AS SELECT syntax.

The DROP TABLE command

Over time, the use of certain tables is no longer required, and you may need to free up disk space for other tables that are being heavily accessed. The Oracle command that enables you to remove a table from the database is the DROP TABLE command. The syntax of the command is as follows:

```
DROP TABLE tablename [CASCADE CONSTRAINTS];
```

When you issue the preceding command, the following happens:

- ✦ Any pending transactions on the table are allowed to complete before the table is dropped. This is similar to making sure that all customers are out of the store before you lock the door.
- ✦ The definition of the table is removed from the data dictionary.
- ✦ Any data that existed in the table is deleted.
- ✦ All indexes created on columns of the table are dropped as well.
- ✦ Any views and synonyms that are based on the table are not dropped, but their definitions become marked as INVALID, and they need to be recreated before they can be used again.

Only certain users are allowed to drop tables. When you create a table, you are its owner. As the owner of any object, you are allowed to make any changes to it, including getting rid of it. The owner of a table can always drop it, as can any user that has been granted the DROP ANY TABLE privilege. The latter is typically true only for DBAs because they can do anything they want in the database.

It is not possible to drop a table, even if you are the owner, when FOREIGN KEYs of other tables depend on a PRIMARY KEY or UNIQUE constraint that exists on your table. The reason for this is that, under these circumstances, your table is the parent to data in another table. For example, the LocationID column of the ScheduledClasses table, which has a FOREIGN KEY defined, is dependent upon the LocationID column of the Locations table, which is the PRIMARY KEY. If you decided to drop the Locations table, the relationship would be broken, and the database would no longer be consistent. In other words, the database would contain child records (ScheduledClasses) without any parents (Locations). Oracle prevents this from happening by default.

More information on constraints can be found later in this chapter in the section "Data Integrity Using Constraints."

If you want to drop a table and break any relationship between it and other tables that depend upon it, you can add the CASCADE CONSTRAINTS clause to the DROP TABLE statement. In this case, Oracle also drops any FOREIGN KEY constraints in child tables that depend upon PRIMARY KEY or UNIQUE constraints in the table being dropped.

After you issue the DROP TABLE command, it cannot be rolled back. This is because any DDL statement in Oracle is its own transaction. As soon as the command completes, a COMMIT is automatically issued by Oracle.

You can drop only one table at a time using the DROP TABLE statement.

The TRUNCATE TABLE command

When you do not want to drop a table from the database, but need to quickly remove all the data in a table, you can use the TRUNCATE TABLE command. The syntax is as follows:

```
TRUNCATE TABLE tablename;
```

This command differs from the DELETE command in the following ways:

- ✦ It removes all rows from the table and releases disk space back to Oracle. The DELETE command does not release disk space back to Oracle.
- ✦ It cannot be rolled back once executed. TRUNCATE TABLE is considered a DDL command and is, therefore, its own transaction.
- ✦ It always removes ALL rows in a table and cannot be used to selectively delete data from a table. When you need to selectively remove only certain rows, use the DELETE command with a WHERE condition instead.

The TRUNCATE TABLE command is the quickest way to remove all of the rows from a table. Like the DROP TABLE command, it also fails when any tables have FOREIGN KEY constraints that depend upon a PRIMARY KEY or UNIQUE constraint in the table being truncated.

Documenting tables and columns

Oracle provides the capability to document your tables and columns using the COMMENT command. While in practice, few organizations make extensive use of this feature, it can be useful in providing insight into the purpose of tables and columns, and what they were intended to be used for to new users or DBAs.

The syntax of the COMMENT command for adding information about a table is:

```
COMMENT ON TABLE tablename IS commentstring;
```

The syntax of the COMMENT command for adding information about a column is:

```
COMMENT ON COLUMN columnname IS commentstring;
```

The comment string must be enclosed in single quotes and must not itself contain any single quotes. If the comment string must contain single quotes, you can use the SQL*Plus escape character (a backslash — \) to tell Oracle that the single quote should be treated as text and not a string terminator.

For example, to add a comment on the Courses table, you can issue the following command:

```
SQL> COMMENT ON TABLE Courses IS
  2  'This is a table providing information on the courses
  3  offered by the training center.  The CourseID column
  4  is the PRIMARY KEY for the table.';

Comment created.

SQL>
```

You can also clear any comments that exist for a table or column in the database by assigning the comment a null string. This removes any existing comments. For example, to remove a comment on the LocationID column of the ScheduledClasses table, you issue the following command:

```
SQL> COMMENT ON COLUMN ScheduledClasses.InstructorID IS '';

Comment created.

SQL>
```

To view comments, you can query the corresponding USER_ or ALL_ views. For tables, comments can be seen by querying the USER_TAB_COMMENTS view for all tables owned by the user, and ALL_TAB_COMMENTS view for all tables that the user has access to. For columns, comments can be seen by querying the USER_COL_COMMENTS view for columns of tables the user owns, or the ALL_COL_COMMENTS view for columns from the tables the user owns and those columns from tables in other users' schemas that the user has access to.

For example, to view comments on any tables that the user Student owns, you issue the following command while connected to the instance as Student, with results similar to those displayed:

```
SQL> SELECT * FROM USER_TAB_COMMENTS;

TABLE_NAME   TABLE_TYPE  COMMENTS
------------ ----------- -------------------------------------------------------
```

```
BATCHJOBS    TABLE
CLASSENROLLM TABLE
ENT

COURSEAUDIT  TABLE
COURSES      TABLE       This is a table providing information on the courses
                         offered by the training center.  The CourseID column
                         is the PRIMARY KEY for the table.

COURSES_TEMP TABLE
INSTRUCTORS  TABLE
LOCATIONS    TABLE
NEWSTUDENTS  TABLE
SCHEDULEDCLA TABLE
SSES

STUDENTS     TABLE

10 rows selected.

SQL>
```

Data Integrity Using Constraints

- ✦ Including Constraints
  - Describe constraints
  - Create and maintain constraints

Constraints are database objects that are used to ensure that data in the database makes sense. They are used to enforce business rules such as "every student must be uniquely identified" or "each student must have a first and last name, but not all students are required to have an email address." Constraints can also be used to prevent deletion of data in one table that may be depended upon by data in another. For example, if you want to ensure that an instructor is not deleted from the Instructors table if he or she is currently teaching or has taught a course, you can create a foreign key on the Classes table that points to the Instructors table's primary key. Once these constraints are defined, an instructor cannot be deleted when one row in the Classes table has that instructor's primary key value in a row in the Classes table.

The types of constraints supported by Oracle 8*i* are listed in Table 7-2.

Table 7-2
Constraints Supported by Oracle 8*i*

Constraint	*Description*
NOT NULL	This constraint states that a column must have a value at all times. Oracle supports NULL by default on all columns in a table, which means that a value for the column does not have to be entered. If a value is required, a NOT NULL constraint can be defined on the column. For example, to ensure that all students have a first and last name entered in the Students table, you specify the NOT NULL constraint for the FirstName and LastName columns.
UNIQUE	A UNIQUE constraint ensures that the value for a column or combination of columns in a table is unique or NULL for the entire table. This can be used to prevent the duplication of data. UNIQUE constraints create or use an existing index to enforce this uniqueness. A table may have multiple UNIQUE constraints.
PRIMARY KEY	A PRIMARY KEY constraint is the combination of a NOT NULL and UNIQUE constraint. This means that any column or columns defined for the PRIMARY KEY constraint ensure that data in the table is UNIQUE and NOT NULL. Like a UNIQUE constraint, a PRIMARY KEY constraint also either creates or uses an existing index to enforce the constraint. A table may have only one PRIMARY KEY constraint.
FOREIGN KEY	A FOREIGN KEY constraint states that data in the column or columns of a table references a PRIMARY KEY or UNIQUE constraint of another table to ensure that the value entered is valid. For example, when specifying that a specific instructor will teach a class, a FOREIGN KEY on the InstructorID column of the Classes table can reference the InstructorID of the Instructors table to ensure that a nonexistent instructor is not assigned to a class.
CHECK	CHECK constraints are used to enforce simple business rules, such as an enrollment date for a class cannot occur after the end date for the class. CHECK constraints can reference data only in the same row of the table and cannot perform any kind of lookups in other tables to verify the condition. For the enforcement of more complex business rules, triggers should be used.

Constraints are a way to ensure that data in your database is entered consistently and that the requirements of the business are met. Should the business rules change, constraints can also be modified without any changes to the client application accessing the database.

Naming constraints

When you specify a constraint on a column or a table, Oracle automatically assigns the constraint a name. The name must be of the format SYS_C*nnnnnn* where *nnnnnn* is a system assigned numerical value to ensure that the name is unique within the schema. This is actually a bad thing because Oracle notifies a user when a constraint is violated while attempting to insert or modify data in a table with a message similar to:

```
*
ERROR at line 1:
ORA-02291: integrity constraint (STUDENT.SYS_C001381)
violated - parent key not found
```

While the name of the owner of the constraint (STUDENT) and the name of the constraint is returned with the error message, it does not provide sufficient information on what actually caused the problem. When you name the constraints when adding them to a table, either during table creation or later with the ALTER TABLE command, the message may look more like the following:

```
*
ERROR at line 1:
ORA-02291: integrity constraint (STUDENT.FK_CLASSENROLLMENT_STUDENTNUM)
violated - parent key not found
```

In this case, we know that the constraint is owned by the user STUDENT and the constraint is probably a foreign key defined on the ClassEnrollment table on the StudentNumber column.

A useful naming convention for constraints, while still adhering to the rules outlined earlier in this chapter, is as follows:

```
constrainttype_table_column
```

Using this convention, when creating a constraint on the Students table that is a primary key on the StudentNumber column, you should name it as follows:

```
PK_Students_StudentNumber
```

Defining constraints

Oracle enables you to define constraints at the time a table is created (that is, when you issue the CREATE TABLE statement) or afterward using the ALTER TABLE command. When you define your constraints at the same time that you create the table, you can specify constraints at the column or table level. Constraints specified for a column at table creation time are also known as *in-line or column constraints* because they are part of the definition of the column. Constraints specified after all columns have been defined are known as *out-of-line or table constraints*.

The syntax for defining constraints while creating a table is as follows:

```
CREATE TABLE [schema.]tablename
  (columnname datatype [DEFAULT expression]
        [[CONSTRAINT constraintname] constrainttype],
   ...
   [CONSTRAINT [constraintname] constraintype (columns),...]);
```

In the first instance of the word CONSTRAINT, you are defining a constraint at the column level that only applies to a specific column. At this point, you need only specify the constraint type, and the constraint is assumed to apply to the column on which it is defined.

In the second instance of the word CONSTRAINT, all of the columns have already been defined, and you are now adding a table constraint. In this case, you must specify the keyword CONSTRAINT to tell Oracle that what follows is a constraint definition. You then have the option to specify a constraint name, which is strongly recommended for the reasons outlined earlier in this chapter. You must specify a constraint type and the column or columns to which the constraint applies. For example, to add two FOREIGN KEY constraints and a PRIMARY KEY constraint to the ClassEnrollment table when the CREATE TABLE statement is issued, you can execute the following command:

```
SQL> CREATE TABLE ClassEnrollment (
  2  ClassID number(5) NOT NULL
  3    CONSTRAINT FK_ClassEnrollment_ClassID
  4      FOREIGN KEY (ClassID) REFERENCES ScheduledClasses (ClassID),
  5  StudentNumber number(5) NOT NULL
  6    CONSTRAINT FK_ClassEnrollment_StudentNum
  7      FOREIGN KEY (StudentNumber) REFERENCES Students (StudentNumber),
  8  Status char (10) NOT NULL ,
  9  EnrollmentDate date NOT NULL,
 10  Price number (9,2) NOT NULL ,
 11  Grade char (4) NULL ,
 12  Comments varchar2 (2000) NULL,
 13    CONSTRAINT PK_ClassID_StudentNumber
 14      PRIMARY KEY (ClassID, StudentNumber);

Table created.

SQL>
```

Note that each NOT NULL found in the preceding table creation statement also specifies a constraint on the ClassEnrollment table. However, because these constraints were not named using the "CONSTRAINT constraintname NOT NULL" syntax, they were each assigned a system-defined name in the form SYS_C*nnnnnn*.

When you define constraints at the column level, you can define any constraint type, but it can apply only to the column itself. In the previous example, each of the FOREIGN KEY constraint created applies only to the column it was specified for.

When you define a constraint at the table level, the constraint can apply to more than one column, as shown previously in the definition of a primary key for the ClassEnrollment table. The primary key is composed of both the ClassID and StudentNumber columns. Because more than one column is specified in the constraint definition, this type of constraint must be specified at the table level. You cannot define a NOT NULL constraint at the table level, but all others may be specified at that level.

A little more about constraint types

As shown previously in Table 7-2, Oracle enables you to use five different constraint types. These include the PRIMARY KEY, UNIQUE, FOREIGN KEY, NOT NULL, and CHECK constraints. Each has its uses and restrictions, which are discussed in the sections that follow.

The NOT NULL constraint

The NOT NULL constraint ensures that the column on which it is defined always has a value. The value can be anything appropriate for the datatype of the column, but the column cannot be NULL. You can define a NOT NULL constraint only at the column level because it applies to the value of the column and not the table as a whole.

To define a NOT NULL constraint on the CourseNumber column of the Courses table, you specify the following at the definition of the column:

```
... CourseNumber number(5) NOT NULL ...
```

Defining a NOT NULL constraint in this manner assigns it a system-generated name in the form of SYS_C*nnnnnn*. When you want to give the constraint a name, you can change the constraint definition to read as follows:

```
... CourseNumber number(5)
   CONSTRAINT NN_Courses_CourseNumber NOT NULL ...
```

The UNIQUE constraint

The UNIQUE constraint ensures that all data within the scope of the constraint does not duplicate an existing set of values, or is NULL. In other words, when you apply a UNIQUE constraint to a column of a table, the value for that column for each row in the table must be different than any other row in the table, or it may be NULL. When applying a UNIQUE constraint, Oracle ensures that values for the columns to which the constraint applies are unique within the table or NULL.

A UNIQUE constraint can be specified at the column level, in which case, the values for the column must be unique within the table or NULL. You can also specify a UNIQUE constraint at the table level by indicating to which columns it should apply. Doing so ensures that the combination of values for the columns are unique within the table.

For example, to ensure that the Email column for each instructor in the Instructors table is different from other instructors, or NULL, you can define a UNIQUE constraint at the column level as follows:

```
...  EMail varchar2(50)
   CONSTRAINT UQ_Instructors_Email UNIQUE ...
```

You also can define the same constraint at the table level after all other columns have been specified, but you need to specify to which column the constraint applies, as shown here:

```
...  CONSTRAINT UQ_Instructors_Email UNIQUE (EMail) ...
```

At the table level, you can also define a UNIQUE constraint that applies to more than one column of a table. When you want to ensure that the combination of FirstName, LastName, and OfficePhone are unique in the Instructors table, you can define a UNIQUE constraint at the table level as in the following:

```
...  CONSTRAINT UQ_Instructors_FNameLNameOTel
        UNIQUE (FirstName, LastName, OfficePhone) ...
```

The way that Oracle enforces uniqueness for the constraint is by creating an index on the table. If an index already exists that has the column or columns to which the UNIQUE constraint applies as the leading part of the index, Oracle does not create another index but uses the existing one. If no index is available, or exists, for the columns to which the UNIQUE constraint applies, Oracle creates an index with the same name as the UNIQUE constraint. It is, therefore, very important that you name your UNIQUE constraints and not allow Oracle to assign them system names in the format SYS_C*nnnnnn* because it becomes more difficult to determine which index may require maintenance as data grows.

While it is not typical to have more than one combination of values to uniquely identify rows in a table, Oracle does enable you to define more than one UNIQUE constraint on the table. In fact, there really is no limit to the number of UNIQUE constraints you can define on a table, but Oracle does limit you to 255 indexes on a table.

A UNIQUE constraint is a valid target of a FOREIGN KEY constraint in another or the same table.

The PRIMARY KEY constraint

The PRIMARY KEY constraint is a combination of the NOT NULL and UNIQUE constraints discussed previously. When you define a PRIMARY KEY on a column or set of columns in a table, the values for those columns specified in the PRIMARY KEY definition must be unique within the table and cannot be NULL. This ensures that every row in the table has a value for the PRIMARY KEY at all times.

Like a UNIQUE constraint, a PRIMARY KEY can be defined at the column or table level. To specify that the CourseNumber column in the Courses table is a PRIMARY KEY, you can define the column in this manner:

```
...    CourseNumber number(5) PRIMARY KEY ...
```

Notice that there is no need to name the constraint or in any other way tell Oracle that data cannot be NULL. This is all automatic with the PRIMARY KEY definition. However, even though you can shortcut the PRIMARY KEY definition as shown previously, you should still follow good practices and define a PRIMARY KEY as any other constraint, as follows:

```
...    CourseNumber number(5)
          CONSTRAINT PK_Course_CourseNumber PRIMARY KEY ...
```

When you need to create a composite PRIMARY KEY (that is, where the uniqueness is true only if the combination of two or more columns are combined), you need to specify the PRIMARY KEY constraint at the table level. To define a PRIMARY KEY on the ClassEnrollment table as the combination of ClassID and StudentNumber, after all the columns for the table have been defined, you specify the following:

```
...    CONSTRAINT PK_ClEnrol_ClassIDStudentNo
          PRIMARY KEY (ClassID, StudentNumber) ...
```

Like a UNIQUE constraint, when a PRIMARY KEY constraint is defined on a table, Oracle tries to use an existing index and, if none is available, Oracle creates an index to enforce the PRIMARY KEY constraint.

Oracle enables you to define only one PRIMARY KEY constraint on a table, as stated in the relational database model. The PRIMARY KEY constraint is a valid target of a FOREIGN KEY constraint in another or the same table.

The FOREIGN KEY constraint

A FOREIGN KEY constraint, also known as a *referential integrity constraint*, specifies that the values in a column or combination of columns in the table where the FOREIGN KEY is defined must already exist in a column or columns defined as a PRIMARY KEY constraint of the same or another table. A FOREIGN KEY creates a parent/child relationship between data in the table with the PRIMARY KEY or UNIQUE constraint (the parent) and the table where the FOREIGN KEY constraint is defined (the child). Just as all children have biological parents, so the data in columns that are part of the FOREIGN KEY must already exist in the table with the PRIMARY KEY or UNIQUE constraint.

The main purpose of a FOREIGN KEY constraint is to ensure that your data makes sense. This is to say that any relationships between tables in your database can be defined using a combination of PRIMARY KEY, or UNIQUE, and FOREIGN KEY constraints. If a user attempts to enter a value in a column on which the FOREIGN KEY has been defined and whose values cannot be found in the table and column to which the FOREIGN KEY points, the value is disallowed, and the operation is rolled back.

For example, if you define a FOREIGN KEY on the LocationID column of the ScheduledClasses table that points to the PRIMARY KEY of the Locations table, which is the LocationID column of the table, and you try to enter a LocationID in the ScheduledClasses table that does not exist in the Locations table, Oracle generates an error and does not allow the insert to take place.

In order to define a FOREIGN KEY constraint, you must specify it either for a column or a table. The FOREIGN KEY constraint must also specify which PRIMARY KEY or UNIQUE constraint columns in another or the same table it references, and, optionally, if you want the data in the child table to be deleted when the parent key value is deleted.

The syntax for defining a FOREIGN KEY constraint at the table level is as follows:

```
CONSTRAINT [constraintname] FOREIGN KEY (column, ...)
     REFERENCES tablename (column, ...)
     [ON DELETE CASCADE]
```

Defining a FOREIGN KEY constraint at the column level is similar, without the need to specify the column to which the FOREIGN KEY applies, as shown here:

```
CONSTRAINT [constraintname] FOREIGN KEY
     REFERENCES tablename {column, ...)
     [ON DELETE CASCADE]
```

For example, to define a FOREIGN KEY on the LocationID column of the ScheduledClasses table, you can define it at the column level as follows:

```
... LocationID number(5) NOT NULL
   CONSTRAINT FK_SchedClass_InstructorID
        REFERENCES Instructors (InstructorID) ...
```

You can also define the same constraint at the table level after all column definitions have taken place as follows:

```
...  CONSTRAINT FK_SchedClass_InstructorID
        FOREIGN KEY (InstructorID)
             REFERENCES Instructors (InstructorID) ...
```

When defining a FOREIGN KEY, three key phrases in the definition tell Oracle how to establish the relationship. They are:

- **FOREIGN KEY:** Tells Oracle that the column names enclosed in parentheses that follow this phrase at a table-level definition of the constraint, or the column where the constraint is defined when done at the column level, must only have values that exist elsewhere.

- **REFERENCES:** Identifies the table and columns that have a PRIMARY KEY or UNIQUE constraint on them and that serve as the source of data verification when a user enters a value in the column or columns to which the FOREIGN KEY applies. If the value being entered into the FOREIGN KEY columns does not exist in the table and columns identified by the REFERENCES clause, Oracle does not allow the entry and generates an error.
- **ON DELETE CASCADE:** Tells Oracle that whenever a value in the table that the FOREIGN KEY REFERENCES is deleted, automatically delete any corresponding child rows in the table on which the FOREIGN KEY is defined. For example, if you delete an InstructorID in the Instructors table, any rows in the ScheduledClasses table with that same InstructorID also are deleted automatically. This ensures that the child table has no values in the FOREIGN KEY columns that do not exist in the parent table (that is, no orphans are in the child table). Without ON DELETE CASCADE, any attempt to delete a parent row with corresponding child rows generates an error. This option should be used with caution.

It is important to note that when you define a FOREIGN KEY constraint, unlike PRIMARY KEY or UNIQUE constraints, Oracle does not automatically create an index. However, the lack of an index on the columns that make up the FOREIGN KEY can lead to performance problems in enforcing the relationship between the parent and the child tables, because Oracle needs to lock the child table or may need to perform full table scans on the child table each time data changes in the parent. For this reason, it is strongly recommended that you create an index on the FOREIGN KEY columns.

Although Oracle introduced support for temporary tables in Oracle 8*i*, it is not possible to create a referential integrity constraint on a temporary table. This means that a temporary table cannot have a FOREIGN KEY constraint defined on it. It is possible to create NOT NULL, CHECK, PRIMARY KEY, and UNIQUE constraints on a temporary table.

The CHECK constraint

A CHECK constraint is a way to enforce simple business rules on all rows of a table, by using an expression. The CHECK constraint can be defined at either the column or table level but cannot perform anything except the most basic of evaluations.

To define a CHECK constraint on the CourseNumber column of the Courses table that ensures that the lowest course number is 1,000, you specify the following:

```
... CourseNumber number(5) NOT NULL
        CONSTRAINT CK_Courses_CourseNumber
             CHECK (CourseNumber >= 1000) ...
```

You can also specify a CHECK constraint at the table level, in which case, the constraint condition can reference other columns in the same row (but not other rows).The following example ensures that the grade of a student in the ClassEnrollment table is NOT NULL when the status of the student is "COMPLETE":

```
... CONSTRAINT CK_ClassElroll_Grade
        CHECK ((Status='COMPLETE' AND Grade is NOT NULL)) ...
```

When defining a CHECK constraint on a column or table, you should keep the following rules in mind:

- ✦ The condition specified by the CHECK constraint is applied to each row of the table as it is inserted or updated. When the row does not meet the CHECK condition, Oracle generates an error and rolls back the statement.
- ✦ The condition can use the same constructs as a WHERE condition and can be complex with a combination of the OR and AND operators and other constructs.
- ✦ References to the CURRVAL, NEXTVAL, LEVEL, and ROWNUM pseudo-columns are not allowed in the CHECK condition.
- ✦ You cannot make calls to the SYSDATE, UID, USER, and USERENV system functions.
- ✦ You cannot reference other rows of the same table or data in other tables. When you need to perform these actions, you must define a trigger on the table.
- ✦ You can reference other columns of the same row only if the CHECK constraint is defined at the table level.
- ✦ You can define more than one CHECK constraint for the same column. There is no real limit to the number of CHECK constraints that can be defined on a column or table, although you should limit the number to only what is needed; otherwise, management of the constraints can become difficult.

Managing constraints

In Oracle, it is possible to add constraints to tables after the tables have been created. Similarly, you can also drop constraints that you no longer need. In some cases, you may also want to turn a constraint off, or disable it, to allow for large bulk operations on the table. After you have performed your bulk operation, you may want to once again enable the constraint to ensure data integrity is enforced. Oracle enables you to perform all of these operations to manage your constraints.

It is important to remember that you can add a constraint to a table after it is created and that you can drop a constraint from a table after it is defined. However, you cannot modify a constraint that already exists on a table. To modify a constraint, you need to drop it and then add it again.

Adding constraints to a table

To add a constraint to an existing table, you use the ALTER TABLE statement, as shown in the following syntax:

```
ALTER TABLE tablename
  ADD [CONSTRAINT constraintname]
        constrainttype (columnname[, ...]);
```

As shown in the preceding syntax example, it is not necessary to name a constraint when adding it to a table, although doing so is highly recommended.

You can add any type of constraint to a table except a NOT NULL constraint. This is because when adding a constraint with the ALTER TABLE syntax, the constraint is considered to be a table-level constraint. Because a NOT NULL constraint applies only to a column, the way to add it is to modify the column definition by using the following syntax:

```
ALTER TABLE tablename
   MODIFY (columnname datatype NOT NULL)
```

When using the preceding syntax to add a NOT NULL constraint to a column, you need to ensure that the table either contains no rows or that all rows have data in the affected column. This is because the addition of a constraint does not modify existing data, and because Oracle will check to see if all rows satisfy the NOT NULL constraint.

When adding constraints to a table using the ALTER TABLE syntax, all other constraint rules apply. This means that only one PRIMARY KEY can be defined on a table, CHECK constraints cannot access data in other rows, and so on.

An example of adding a FOREIGN KEY constraint on the InstructorID column of the ScheduledClasses table after it has already been created is as follows:

```
SQL> ALTER TABLE ScheduledClasses
  2  ADD CONSTRAINT FK_SchedClasses_InstID
  3  FOREIGN KEY (InstructorID)
  4  REFERENCES Instructors (InstructorID);

Table altered.

SQL>
```

To add a NOT NULL constraint to the ChangedBy column of the CourseAudit table, you issue the following command:

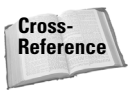
```
SQL> ALTER TABLE CourseAudit
  2  MODIFY (ChangedBy varchar2(15)
  3  CONSTRAINT NN_CourseAudit_ChangedBy NOT NULL);
```

```
Table altered.

SQL>
```

Dropping constraints

Dropping constraints that are already defined on a table can be a simple process or one that requires more thought and effort. The guiding factor in determining which it will be often depends on the constraint type. Dropping CHECK, NOT NULL, and FOREIGN KEY constraints is usually a pretty straightforward process. Dropping PRIMARY KEY and UNIQUE constraints may require more care and additional work to ensure that data integrity is not compromised.

If you properly name constraints when you define them, the process of dropping them will be much easier. If you follow a naming convention when you define your constraints, you do not have to go searching in the USER_CONSTRAINTS and USER_CONS_COLUMNS data dictionary views to figure out what the name of the constraint you wish to drop is — you'll have a pretty good idea already.



The USER_CONSTRAINTS and USER_CONS_COLUMNS data dictionary views are discussed later in the "Viewing Information About Constraints" section of this chapter. To get information on their structure, please refer to Appendix F.

The syntax for dropping a constraint is as follows:

```
ALTER TABLE tablename
  DROP PRIMARY KEY | UNIQUE (column) |
        CONSTRAINT constraintname [CASCADE];
```

As noted in the preceding syntax, to drop a PRIMARY KEY or UNIQUE constraint, you do not need to specify the constraint name. Because a table can contain only one PRIMARY KEY, issuing the following command:

```
ALTER TABLE tablename DROP PRIMARY KEY;
```

automatically drops the PRIMARY KEY constraint on the affected table, as well as any associated index if the index was created by the constraint definition (that is, if the PRIMARY KEY constraint used an existing index, that index is not dropped).

Similarly, you can also drop a UNIQUE constraint without specifying its name by referencing the column or columns on which it is defined. To drop a UNIQUE constraint defined on the EMail column of the Instructors table, you issue the following command:

```
SQL> ALTER TABLE Instructors
  2  DROP UNIQUE (EMail);

Table altered.

SQL>
```

Similar to the PRIMARY KEY, any indexes that were created to enforce the uniqueness also are dropped.

The recommended way to drop any constraint in a table is to specify the name of the constraint that you wish to drop in the ALTER TABLE statement. Using this method, you can drop any constraint, including NOT NULL constraints, as long as you know the constraint name. You do not have to specify what the type of the constraint is — simply provide the name of the constraint — in your ALTER TABLE statement. For example, to drop a NOT NULL constraint defined on the ChangedBy column of the Course Audit table, you issue the following command:

```
SQL> ALTER TABLE CourseAudit
  2  DROP CONSTRAINT NN_CourseAudit_ChangeBy;

Table altered.

SQL>
```

An important consideration when dropping PRIMARY KEY or UNIQUE constraints is that they may be depended upon by FOREIGN KEY constraints in other tables. When this is the case and you attempt to drop a PRIMARY KEY or UNIQUE constraint, you will get an error message from Oracle similar to the following:

```
SQL> ALTER TABLE Instructors
  2  DROP PRIMARY KEY;
ALTER TABLE Instructors
*
ERROR at line 1:
ORA-02273: this unique/primary key is referenced by some foreign keys

SQL>
```

This is a good time to reconsider your decision to drop the PRIMARY KEY because doing so may compromise the referential integrity of the database. If you are certain that you want to drop the PRIMARY KEY or UNIQUE constraint, you can instruct Oracle to also drop any FOREIGN KEY constraints that depend upon the constraint that you are dropping. To do so, include the CASCADE keyword in the ALTER TABLE statement, as in the following example:

```
SQL> ALTER TABLE Instructors
  2  DROP PRIMARY KEY CASCADE;

Table altered.

SQL>
```

The preceding example drops the PRIMARY KEY on the Instructors table (InstructorID) and also drops any FOREIGN KEY constraints that depend upon it, such as the InstructorID column of the ScheduledClasses table.

VALIDATE and NOVALIDATE constraints

In versions prior to Oracle 8*i*, when you enabled a constraint, all data in the table had to satisfy the constraint conditions. If that was not true, you could not enable the constraint and had to correct the problem prior to enabling the constraint on the table. In Oracle 8*i*, two new clauses have been added to the ALTER TABLE syntax dealing with enabling or disabling a constraint — VALIDATE and NOVALIDATE.

In Oracle 8*i*, the default mode for enabling a constraint is VALIDATE, which means that Oracle ensures that all rows in the table satisfy the constraint condition. When you enable a constraint and specify NOVALIDATE, Oracle assumes that all existing rows in the table satisfy the constraint condition and does not read the rows to verify it. In other words, the existing data is not checked for constraint compliance, and it is possible to have rows in the table that do not conform to the constraint condition. However, any new rows added to the table, or any existing rows that are updated, must comply with any constraints on the table.

The syntax for enabling a constraint has also been expanded to include these new clauses. The full syntax is as follows:

```
ALTER TABLE tablename
   ENABLE | DISABLE
        CONSTRAINT constraintname
              [VALIDATE | NOVALIDATE] [CASCADE];
```

When enabling a constraint, the default is VALIDATE (that is, all rows must satisfy the constraint condition). When disabling constraints, the default is NOVALIDATE (the constraint is not enforced).

Enabling a constraint NOVALIDATE allows existing rows to violate the constraint condition, but all new or modified rows must satisfy the constraint condition.

DISABLE VALIDATE drops any indexes, if the constraint was a PRIMARY KEY or UNIQUE constraint, and ensures that all data in the table satisfies the constraint conditions. The net effect of this is to make the table read only, because any attempt to INSERT, UPDATE, or DELETE rows in the table generates an error. This is useful for small lookup tables that do not change frequently and are too small to create an index on.

Disabling and enabling constraints

In many environments, especially those using Oracle for data warehousing or as a decision-support system, it is quite common to perform loads of large amounts of data. Constraints can be problematic when you need to load large amounts of data into one table and a related row does not yet exist in the parent table. Furthermore, if Oracle has to check every row as it is inserted into a table, this can increase the time required to perform the load. This is especially true in those situations where you have defined PRIMARY KEY or UNIQUE constraints, because indexes also have to be updated as the rows are loaded.

To increase the speed of large data loads, Oracle enables you to temporarily turn off, or disable, constraints before you perform the load. You can also enable any constraints that you have turned off after the load has completed. At that point, data in the tables must satisfy constraint conditions.

The "Introduction to Oracle: SQL & PL/SQL" exam tests you on your knowledge of enabling and disabling constraints. You may not be tested on the VALIDATE/NOVALIDATE clause of this syntax, but this information is useful in making better use of your own databases.

Disabling constraints

You may want to disable a constraint prior to a large data load into the table or when you are performing a bulk update that may modify key values where a PRIMARY KEY or UNIQUE constraint exists. Disabling a constraint tells Oracle to no longer enforce the constraint condition and to allow data that does not satisfy the constraint to be added to, changed in, or deleted from the table. If you disable a PRIMARY KEY or UNIQUE constraint, Oracle automatically drops any associated indexes.

The syntax to disable a constraint is:

```
ALTER TABLE tablename
   DISABLE CONSTRAINT constraintname [CASCADE];
```

The CASCADE clause of the ALTER TABLE syntax is used to also automatically disable any FOREIGN KEY constraints that are dependent upon the PRIMARY KEY or UNIQUE constraint being dropped.

The CASCADE clause disables all FOREIGN KEY constraints depending on the PRIMARY KEY or UNIQUE constraint being disabled. However, no corresponding CASCADE clause is available when you ENABLE the constraint. It is always a good idea to manually disable all FOREIGN KEY constraints first and then disable the PRIMARY KEY or UNIQUE constraint without the CASCADE clause. This way, you will be certain that no dependent FOREIGN KEY constraints are dropped without you knowing about it.

Another option is to use the CASCADE clause but to create a script that will re-create the FOREIGN KEY constraints in the appropriate order. You can use the USER_CONSTRAINTS and USER_CONS_COLUMNS data dictionary views to determine on which PRIMARY KEY or UNIQUE constraints a FOREIGN KEY depends.

There is a short form to disable a PRIMARY KEY in a table. The syntax is as follows:

```
ALTER TABLE tablename
   DISABLE PRIMARY KEY [CASCADE];
```

Because a table can have only one PRIMARY KEY, Oracle knows exactly which constraint to disable.

Finding rows not meeting constraint conditions

When you attempt to enable a constraint, it is also possible to determine which rows in the table are causing the operation to fail. This can be done by creating a table to hold exceptions to the constraint that Oracle has found in the table. A script called UTLEXCPT.SQL can be used to create a table called EXCEPTIONS, which has the proper structure. The script can be found in the ORACLE_HOME/RDBMS/ADMIN directory, where ORACLE_HOME is the physical location on the hard drive where you installed the Oracle software. After running the script, you have a table with the following structure:

```
SQL> desc Exceptions
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------
-----------
 ROW_ID                                             ROWID
 OWNER                                              VARCHAR2(30)
 TABLE_NAME                                         VARCHAR2(30)
 CONSTRAINT                                         VARCHAR2(30)

SQL>
```

After creating the Exceptions table in your schema, you can then issue the command to enable the constraint in question and tell Oracle to put information about any rows that do not satisfy constraint conditions in the Exceptions table. The syntax to do so is as follows:

```
ALTER TABLE tablename
   ENABLE CONSTRAINT constraintname
        EXCEPTIONS INTO schema.exceptiontable;
```

By then retrieving the rows from the table in question using the ROWID placed in the Exceptions table, you can correct any errors. After all errors have been corrected, the constraint can then be enabled.

Enabling constraints

After a bulk load or update has been performed, you may want to enable the constraints that you disabled prior to performing the action. The ALTER TABLE command enables you to perform this action.

When you enable a constraint, Oracle, by default, verifies that all data satisfies the constraint condition. If any rows are found not to conform to the requirements of the constraint, Oracle generates an error, and the constraint remains disabled.

To enable a constraint, the syntax is as follows:

```
ALTER TABLE tablename
   ENABLE CONSTRAINT constraintname;
```

Similar to the syntax to disable a PRIMARY KEY constraint, there is also a short form to enable a PRIMARY KEY constraint, as shown here:

```
ALTER TABLE tablename
   ENABLE PRIMARY KEY;
```

When the constraint to be enabled is a PRIMARY KEY or UNIQUE constraint and all rows satisfy the constraint condition, Oracle automatically creates an index to enforce the PRIMARY KEY or UNIQUE constraint, if no other index on the columns making up the constraint already exists.

Viewing information about constraints

After you have defined constraints on your tables, if you want to find out what the constraints are called and on which columns they are defined, you can use the USER_CONSTRAINTS and USER_CONS_COLUMNS views. The corresponding ALL_CONSTRAINTS and ALL_CONS_COLUMNS views also enable you to find out information about constraints defined on tables that you have been granted access to.

For example, to get a list of constraints defined on the ScheduledClasses table, you issue the following command:

```
SQL> SELECT constraint_name, constraint_type,
  2  search_condition
  3  FROM USER_CONSTRAINTS
  4  WHERE table_name = 'SCHEDULEDCLASSES';

CONSTRAINT_NAME                C SEARCH_CONDITION
------------------------------ - --------------------------------
SYS_C001879                    C "CLASSID" IS NOT NULL
SYS_C001880                    C "COURSENUMBER" IS NOT NULL
SYS_C001881                    C "LOCATIONID" IS NOT NULL
SYS_C001882                    C "CLASSROOMNUMBER" IS NOT NULL
SYS_C001883                    C "INSTRUCTORID" IS NOT NULL
SYS_C001884                    C "STARTDATE" IS NOT NULL
SYS_C001885                    C "DAYSDURATION" IS NOT NULL
SYS_C001886                    C "STATUS" IS NOT NULL
PK_CLASSID                     P
FK_SCHEDCLASS_COURSENUM        R
FK_SCHEDCLASSES_LOCATIONID     R
FK_SCHEDCLASSES_INSTID         R

12 rows selected.

SQL>
```

When viewing the preceding output, it is important to note that the Constraint_Type column of the view returns a single character for the type of constraint. The letter C indicates a CHECK constraint, R indicates a FOREIGN KEY

constraint, P indicates a PRIMARY KEY constraint, and U indicates a UNIQUE constraint. As shown previously, NOT NULL constraints are actually CHECK constraints where the condition specifies that the column on which the NOT NULL is defined "is NOT NULL". You can also see that NOT NULL constraints, when defined on the ScheduledClasses table were not named, because they have been assigned system names in the format SYS_C*nnnnnn*.

For PRIMARY KEY, UNIQUE, and FOREIGN KEY constraints, when you want to determine which columns make up the constraint, you can query the USER_CONS_COLUMNS view, as in this example:

```
SQL> SELECT constraint_name, column_name
  2  FROM USER_CONS_COLUMNS
  3  WHERE table_name = 'SCHEDULEDCLASSES';

CONSTRAINT_NAME                COLUMN_NAME
------------------------------ -----------------------------------------
FK_SCHEDCLASSES_INSTID         INSTRUCTORID
FK_SCHEDCLASSES_LOCATIONID     LOCATIONID
FK_SCHEDCLASS_COURSENUM        COURSENUMBER
PK_CLASSID                     CLASSID
SYS_C001879                    CLASSID
SYS_C001880                    COURSENUMBER
SYS_C001881                    LOCATIONID
SYS_C001882                    CLASSROOMNUMBER
SYS_C001883                    INSTRUCTORID
SYS_C001884                    STARTDATE
SYS_C001885                    DAYSDURATION
SYS_C001886                    STATUS

12 rows selected.

SQL>
```

Creating Other Database Objects

Objective

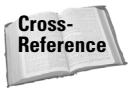
- Describe a view
- Create a view
- Retrieve data through a view
- Insert, update, and delete data through a view
- Drop a view
- Create, maintain, and use sequences
- Create and maintain indexes
- Create private and public synonyms

Aside from tables and the constraints that you can define on them, Oracle also enables you to create other objects that make the use of your database easier or speed up performance of queries. Some of the objects that you can create, and which are discussed in this chapter, include views, indexes, sequences, and synonyms. Oracle also allows the creation of many other objects, including clusters, stored procedures, user-defined functions, packages, and tablespaces.

The "Introduction to Oracle: SQL & PL/SQL" exam tests your knowledge of creating tables, defining constraints, and creating views, indexes, synonyms, and sequences. The creation and use of other Oracle database objects are tested in other exams.

Views

A view is a database object that enables you to present data from one or more tables in a single rowset (or recordset). It is created by specifying a name for the view and assigning that name to a SELECT statement that defines the view. When a user retrieves data from the view, the view appears as if it were a table. In fact, a user cannot tell the difference in the output when using an SQL SELECT statement to retrieve data from a view or a table — the output appears the same with column headings and rows returned as expected.



The syntax of the SELECT statement is discussed in detail in Chapters 2 through 4.

Views are useful to make complex queries easier to write. By creating a view whose definition is the complex SQL SELECT statement for the query, users are able to SELECT from the view and not have to repeat the complex SQL syntax each time they want to see the resulting data.

Views are also useful in restricting access to only certain columns or rows in a database. For example, if you want to create a phone list for your organization with everyone's name, email address, phone number, and office location, you can create a table to hold that information or, preferably, create a view that extracts the necessary columns from the Employees table. In this way, the data is stored only once in the Employees table, and those columns in the Employees table that users should not be allowed to see (such as salary, bonus, and last review results) are not available through the view.

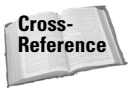
The tables on which a view is based are called *base tables*. A view can be based upon one or more base tables. The number of tables a view is based on, and the types of operations that are being performed in the SELECT statement that defines a view, determine whether the view is considered to be a simple or a complex view.

A simple view is one that is based upon a single table and has the following additional characteristics:

- ✦ No functions, either system or user-defined, are used in the query.
- ✦ No grouping of data is used in the query. This means that the SELECT statement contains no GROUP BY clauses and no group functions in the column list.

A complex view is one that is based upon more than one table, or has functions, groupings, or other elements in the SELECT statement. A view that contains only one base table is considered complex if its definition makes use of functions or groupings of data.

The determination of whether a view is simple or complex is important if you want to modify data through the view. When using simple views, it is possible to perform INSERT, UPDATE, and DELETE operations on the view, and the underlying base table will receive, change, or remove the rows affected, provided that no integrity constraints are violated by the operation.



The topic of performing Data Manipulation Language (DML) operations through a view is covered later in this chapter in the section "Performing DML Through a View."

The CREATE VIEW statement

The CREATE VIEW statement is used to create a view in Oracle8*i*. The syntax is as follows:

```
CREATE [OR REPLACE] [FORCE | NOFORCE] VIEW viewname
   [(columnalias, columalias, ...)]
AS SELECT ...
   [WITH CHECK OPTION [CONSTRAINT constraintname]]
   [WITH READ ONLY]
```

Because the preceding syntax is somewhat long and complex, information on each of the clauses is presented in Table 7-3.

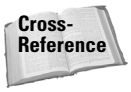
Table 7-3
CREATE VIEW Clauses

Clause	***Description***
OR REPLACE	Oracle 8*i* enables you to modify the definition of a view. Unlike tables, where the ALTER TABLE clause is used to add or remove columns, there is no corresponding ALTER VIEW clause to change a view's definition. To do so, you add the OR REPLACE clause to the CREATE command, and Oracle replaces a view with the same name specified on the command line with the new definition. Using the CREATE OR REPLACE clause creates the view when it does not exist and replaces an existing view when it does. It is a good idea to always use CREATE OR REPLACE to avoid errors.
FORCE/NOFORCE	When a view is created, Oracle checks to see that the underlying base tables already exist in the database. If they do not, Oracle generates an error and does not create the view (the default NOFORCE behavior). If you want to create a view based on tables that do not yet exist, or that you may not have been granted permissions to yet, you can specify the FORCE keyword to tell Oracle to create the view anyway. The view is marked as INVALID. The first time the view is used (that is, a user issues a SELECT statement from the view), Oracle verifies that all the objects in the view definition exist and updates the view's definition in the data dictionary. If the view cannot be verified, Oracle returns an error to the user, and the view remains in an INVALID state until the next time it is accessed and the process is repeated. It is generally not recommended that you use the FORCE option. You should always create all dependent tables first, or ensure that you have permissions to tables in another user's schema upon which your view definition depends.
(columnalias, ...)	When a view is created, the names of the columns for the view are derived from the list of column names specified in the SELECT statement. If you want to specify different names for the columns, you can do so here. This is useful when some of the columns in the SELECT statement are the result of an expression or functional operation. Another way to alias the column names is to specify an alias in the SELECT statement. Of the two, either works equally well, although column aliasing in the SELECT statement is more common.

Continued

Table 7-3 *(continued)*

Clause	*Description*
AS SELECT ...	This is the SELECT statement (that is, the subquery) that defines the view. The names of the columns for the view are derived from column names specified and/or aliased in the SELECT statement. The subquery can be any SELECT statement that is valid in Oracle. Prior to Oracle 8*i*, the subquery could not contain an ORDER BY clause. This is no longer true. When columns in the subquery are the result of an expression or functional operation, they must be aliased either in the subquery itself or in the VIEW definition. The same also holds true for columns with the same name from different tables, if the subquery contains a join condition.
WITH CHECK OPTION	The WITH CHECK OPTION specifies that any changes to data through an INSERT or UPDATE operation cannot be performed if those changes would cause the added or modified row not to be visible through the view. For example, if you create a view for all California customers and you specify the WITH CHECK OPTION, any attempt to insert or modify a row in the base table through the view with a state of Georgia would fail because those rows would not be returned the next time you issued a SELECT statement on the view.
CONSTRAINT constraintname	Specifies the name of the constraint that is defined on the view when the WITH CHECK OPTION is specified. Oracle adds the conditions for the WITH CHECK OPTION to the data dictionary automatically based upon the WHERE clause of the SELECT statement that defined the view. If not named by the users, Oracle assigns the constraint a system-supplied name in the form of SYS_C*nnnnnn*. If you want to provide your own name to the WITH CHECK OPTION constraint, you can specify it by including this clause. This clause is available only if the WITH CHECK OPTION has been specified.
WITH READ ONLY	If you do not want to allow users to add, change, or delete data from the base table through the view, create the view with the WITH READ ONLY clause. Any view with this clause specified generates an error if a user attempts to perform an INSERT, UPDATE, or DELETE operation on the view. Most views should typically be created WITH READ ONLY, unless you create INSTEAD OF triggers to deal with complex updates.



For more information on INSTEAD OF triggers, refer to Chapter 13.

If you want to create a view that returns the names of instructors and the courses they are scheduled to teach, you issue the following command:

```
SQL> CREATE VIEW InstructorClasses AS
  2   SELECT TRIM(FirstName || ' ' || LastName) AS Instructor,
  3     C.CourseNumber AS CourseID,
  4     CourseName, StartDate
  5   FROM Instructors I, Courses C, ScheduledClasses S
  6   WHERE I.InstructorID=S.InstructorID
  7    AND S.CourseNumber=C.CourseNumber;

View created.

SQL>
```

Using the DESCribe command to find the structure of the view provides this information:

```
SQL> DESC InstructorClasses
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 INSTRUCTOR                                         VARCHAR2(61)
 COURSEID                                  NOT NULL NUMBER(38)
 COURSENAME                                NOT NULL VARCHAR2(200)
 STARTDATE                                 NOT NULL DATE

SQL>
```

Querying data by selecting from the view runs the SELECT statement that defines the view and also enables you to further refine or sort your data, as shown here:

```
SQL> SELECT * FROM InstructorClasses
  2  ORDER BY StartDate;

INSTRUCTOR        COURSEID COURSENAME                                STARTDATE
--------------- ---------- ---------------------------------------- ---------
David Ungar            100 Basic SQL                                06-JAN-01
Lisa Cross             200 Database Performance Basics              13-JAN-01
Kyle Jamieson          100 Basic SQL                                14-FEB-01

SQL>
```

If you want to modify the view to sort the output by the StartDate instead of having to specify it when issuing a SELECT statement on the view, you issue the following command:

```
SQL> CREATE OR REPLACE VIEW InstructorClasses AS
  2   SELECT TRIM(FirstName || ' ' || LastName) AS Instructor,
  3     C.CourseNumber AS CourseID,
  4     CourseName, StartDate
  5   FROM Instructors I, Courses C, ScheduledClasses S
  6   WHERE I.InstructorID=S.InstructorID
  7    AND S.CourseNumber=C.CourseNumber
  8   ORDER BY StartDate;

View created.

SQL>
```

Then, issuing the following at the SQL*Plus prompt, provides you with presorted output:

```
SQL> SELECT * FROM InstructorClasses;

INSTRUCTOR        COURSEID COURSENAME                                STARTDATE
--------------- ---------- ---------------------------------------- ---------
David Ungar            100 Basic SQL                                06-JAN-01
Lisa Cross             200 Database Performance Basics              13-JAN-01
Kyle Jamieson          100 Basic SQL                                14-FEB-01

SQL>
```

Performing DML through a view

As mentioned earlier, Oracle, by default, enables users to perform INSERT, UPDATE, and DELETE operations on data that is presented through a view. No restrictions are placed on this for simple views, and it is possible to insert rows that cannot be seen when querying the view if the WITH CHECK OPTION is not specified.

Oracle, to ensure that the data makes sense and to deal with the fact that a view is simply a SELECT statement that has been given a name, has placed several restrictions on performing DML through a view. In fact, depending upon how the subquery is formulated, it may not be possible to perform any DML through a view unless an INSTEAD OF trigger is created.

Any DML performed through a view always modifies the base table that is referenced in the subquery. The view itself has no physical storage because all of its data is derived from the tables that make it up.

Oracle supports the use of snapshots (in versions prior to Oracle 8*i*) and materialized views (in Oracle 8*i*). These are views that have a storage component and are used in large data warehousing environments to precalculate values so that queries work faster. Oracle 8*i* can make use of the data in materialized views instead of or together with the base tables to return the result of a SELECT statement to the user more quickly than by scanning the base table data. For more information on materialized views, refer to the *Oracle 8i Data Warehousing Guide*, a component of the Oracle documentation set. The "Introduction to Oracle: SQL & PL/SQL" exam does not test your knowledge of materialized views.

The rules for performing DML on a view, without the use of INSTEAD OF triggers, are:

- ✦ On a simple view, it is always possible to perform DML because a simple view is based on only a single table and contains no functions, expressions, or GROUP BY, DISTINCT, or other problematic clauses.
- ✦ You cannot delete, update, or insert data through a view if the view definition contains a GROUP BY clause.
- ✦ You cannot delete, update, or insert data through a view if the view definition contains the DISTINCT keyword in the column list.
- ✦ You cannot delete, update, or insert data through a view if the view contains reference to the ROWNUM pseudo-column.
- ✦ You cannot update or insert data through a view if the view contains columns that are the determined by the result of an expression.
- ✦ You cannot insert data through a view if the base tables contain columns not referenced in the view definition that have a NOT NULL constraint and do not have a DEFAULT defined.

If any of the preceding conditions are true, you cannot perform an INSERT, UPDATE, or DELETE except through the use of INSTEAD OF triggers.

The WITH CHECK option

If you allow users to perform DML operations on a view, it may be possible for the users to make changes to the data that would not allow them to view the row modified using the view.

For example, suppose you allow a user to change the State column of the Instructors table through the a view called NewYorkInstructors that lists all New York instructors. The user may be able to change the location of an instructor from New York to Toronto. Using the NewYorkInstructors view, the instructor whose location was changed to Toronto would no longer be visible through the view. If you do not want to allow users to make changes to the data that would make the data not visible using the view, you can create the view using the WITH CHECK OPTION. To create the NewYorkInstructors view, the syntax is:

```
SQL> CREATE OR REPLACE VIEW NewYorkInstructors AS
 2  SELECT TRIM(FirstName || ' ' || LastName) as Name,
 3  City, State, Email, CellPhone, InstructorType
 4  FROM Instructors
 5  WHERE State = 'NY'
 6  WITH CHECK OPTION;

View created.

SQL>
```

When retrieving data, a list of only those instructors residing in New York state is displayed, as shown here:

```
SQL> col name format a40
SQL> SELECT * FROM NewYorkInstructors;

NAME                                     CITY                           ST
---------------------------------------- ------------------------------ --
EMAIL                                                 CELLPHONE       INSTRUCTOR
----------------------------------------------------- --------------- ----------
David Ungar                              New York                       NY
davidungar@trainers.com                                                ORACLE

Kyle Jamieson                            New York                       NY
kylejamieson@trainers.com                             412-987-0423    ORACLE


SQL>
```

If you attempt to change the value of the State column for David Ungar to ON, Oracle returns the following error:

```
SQL> UPDATE NewYorkInstructors SET State='ON' WHERE Name LIKE '%Ungar%';
UPDATE NewYorkInstructors SET State='ON' WHERE Name LIKE '%Ungar%'
       *
ERROR at line 1:
ORA-01402: view WITH CHECK OPTION where-clause violation


SQL>
```

As you can see, Oracle checks the WHERE clause of the view definition to ensure that after the UPDATE, the row being modified is still visible through the view. Because this would not be true should the UPDATE complete, Oracle does not allow the update and returns the ORA-01402 error code.

The READ ONLY option

If you do not want to allow any INSERT, UPDATE, or DELETE statements to be performed on a view, you can create the view using the WITH READ ONLY option. This returns an error whenever an attempt is made to modify data through the view; any changes to the data on which the view is based must be done using the base tables or other views on which the target view depends.

The READ ONLY option is useful for complex views that perform aggregation and/or join many tables. Because performing updates on these views may not be successful anyway if an INSTEAD OF trigger is not used, preventing data modifications completely ensures that there is no possibility of data inconsistency.

To disable updates on the InstructorClasses view, you create the view as follows:

```
SQL> CREATE OR REPLACE VIEW InstructorClasses AS
  2   SELECT TRIM(FirstName || ' ' || LastName) AS Instructor,
  3     C.CourseNumber AS CourseID,
  4     CourseName, StartDate
  5   FROM Instructors I, Courses C, ScheduledClasses S
  6   WHERE I.InstructorID=S.InstructorID
  7    AND S.CourseNumber=C.CourseNumber
  8   WITH READ ONLY;

View created.

SQL>
```

Any attempt to perform an INSERT, UPDATE, or DELETE on the view results in the following:

```
SQL> UPDATE InstructorClasses
  2  SET StartDate='14-APR-01' WHERE CourseID=200;
SET StartDate='14-APR-01' WHERE CourseID=200
    *
ERROR at line 2:
ORA-01733: virtual column not allowed here


SQL>
```

The preceding error message, which is returned whenever an INSERT or UPDATE is attempted on a view created WITH READ ONLY, may not clearly indicate what the problem is, which may be an issue because a user may repeatedly attempt to perform the update but no changes to the data will be allowed.

When attempting to DELETE data through a view that was created using the WITH READ ONLY option, you get the following error:

```
SQL> DELETE FROM InstructorClasses;
DELETE FROM InstructorClasses
            *
ERROR at line 1:
ORA-01752: cannot delete from view without exactly one key-preserved table


SQL>
```

Again, the message may be somewhat cryptic, but the desired result takes place — the DELETE fails.

Dropping views

To remove a view from the database, you can use the DROP VIEW command, as follows:

```
SQL> DROP VIEW InstructorClasses

View dropped.

SQL>
```

When a view is dropped, any objects that reference it, such as other views based upon the view that was dropped, are marked as INVALID by Oracle and no longer work correctly until they are modified to remove the reference to the view. However, dropping a view does not drop the objects that it is based on. In other words, any tables or other objects referenced by the view remain in the database after the view is dropped.

Indexes

Objective

- ✦ Create and maintain indexes

Database users always want to be able to retrieve data in the fastest possible way. In large databases, scanning the entire table to locate a particular value for a row can take a long time. Indexes can be created to make data retrieval quicker.

Indexes store, by default, the value of the column or columns of a table being indexed (also known as the *key*) in the index, as well as a pointer to the physical location of the row or rows that hold the value (the *rowid*). By issuing a query against the table where the index is created, Oracle may decide to scan the index, which is always organized in either ascending or descending order according to the key, and when it finds the entries with the required key value, use the corresponding key's rowid to return only those rows with the appropriate value.

This chapter only discusses the creation of B*Tree indexes, which are covered on the "Introduction to Oracle: SQL & PL/SQL" exam. Oracle also supports another type of index, called a bitmap index. Bitmap indexes are useful in large data warehousing environments and can speed up queries considerably. Bitmap indexes should only be used on tables whose data does not change, as they have a significant overhead in key values are changed, or data is added or deleted to the table.

Indexes can also be used to enforce uniqueness in a table. It is possible to create a unique index on a table (and one is created for you if it does not already exist) when you specify a UNIQUE or PRIMARY KEY constraint on a table. Oracle uses the index to ensure that no duplicate values exist.

You should note that indexes can also increase the time it takes to perform updates of data on a table on which they are defined. This is because as rows are inserted or updated in a table, Oracle may have to insert the data as well as create or update any index keys for that row. Having too many indexes on a table can be detrimental to the speed of data modifications; not having enough indexes can be detrimental to the speed of data retrieval.

The syntax of the CREATE INDEX command is as follows:

```
CREATE INDEX indexname
ON tablename (column [, column] ...);
```

The preceding CREATE INDEX syntax is not complete. Many more options can be specified when an index is created. You are not tested on these additional options on the "Introduction to Oracle: SQL & PL/SQL" exam. If you want more information on these options, consult the *Oracle 8i SQL Reference* manual that is part of the Oracle documentation set.

At a minimum, the syntax to create an index requires you specify a number of key things:

- **Index name:** The name of the index must be unique within the schema in which it is being created.
- **Table name:** The name of the table on which the index is being created.
- **Column list:** The list of columns within the table on which the index is being created. You may specify more than one column in the column list, in which case a "composite" index is created. Separate each column using a comma. The column list may also include an expression, in which case a function-based index is created.

To create an index on the LastName column of the Students table, you issue the following command:

```
SQL> CREATE INDEX StudentLastName
  2  ON Students (LastName);

Index created.

SQL>
```

As when creating any object in Oracle, when creating indexes, you should use a naming convention. The previous example names the index with the table name followed by the column being indexed (that is, Student_LastName). This may or may not be an appropriate way for you to name your index. The convention used in your database is not as much of an issue as the existence and consistent use of a naming convention.

Reasons for and against creating B-tree indexes

You should not create indexes in your databases because you think you need them, but rather, because it makes sense to do so. You should follow a number of rules when determining whether or not an index on a particular column or set of columns makes sense.

Remember that too many indexes on a table can cause inserts and updates to take longer than if the indexes did not exist. This is because indexes occupy physical storage in the database and cause Oracle to update the block where the row is on the hard disk, as well as any index blocks that exist for columns on the table. In environments where data changes frequently, you need to decide carefully if indexes should be created.

Some of the conditions when indexes make sense include the following:

- **Columns used in join conditions:** It is a good idea to create an index on those columns that are used to join two tables in the WHERE clause of a SELECT statement. For example, the LocationID column of the ScheduledClasses table might be an appropriate candidate for an index because you may frequently join the ScheduledClasses and Locations tables on the LocationID column of each table to get information on where the class is held.

  Typically, any column, or collection of columns, used to define a FOREIGN KEY constraint is an appropriate candidate for an index. Oracle does not automatically create an index on FOREIGN KEY columns, so you should.

- **High cardinality columns:** Columns that have a wide range of different values (that is, high cardinality) are appropriate candidates for an index if they are frequently used in the WHERE clause of a SQL statement. If a column's values do not frequently repeat, this means that Oracle can retrieve a single row, or a small number of rows, when using the index instead of a large chunk of the data.

Similarly, columns whose values often repeat are not appropriate candidates for an index. For example, a column called gender is not a good candidate for an index because it can store only three possible values — male, female, and unknown.

- **Columns with many NULLs:** If a column contains a NULL, Oracle does not index it. If you have a column in a table with many NULL values, indexing the column creates a "sparse index," (that is, an index with only those rows containing data having index keys). Oracle does not create an index entry for a NULL value column. In this way, the index may represent a small portion of the actual data in the table, making index scans very efficient when searching for a value.
- **Columns frequently used in a WHERE, GROUP BY, or ORDER BY clause:** Oracle tries to make use of an index when the optimizer finds that one is available and makes sense. Oracle uses the WHERE clause, GROUP BY, and ORDER BY clauses of a SQL statement to determine whether an index can be used and what the most efficient method to retrieve the data would be. Creating indexes on those columns frequently queried by users and making up one of the aforementioned clauses could allow the optimizer to make use of the index in satisfying a query and thereby speed performance. A good way to gauge this is to look at the application code to determine what columns are used in these clauses and then create the index where appropriate.
- **Most data retrieval involves 2 to 4 percent of a large table:** In cases where a table is large and most queries return a small portion of the total data in the table, creating an index when any of the preceding conditions also exist makes sense. However, if most queries on the table return 25 percent or more of the data, creating an index may not help query performance, and a full table scan may actually end up being quicker. This is because indexes cause more disk I/O operations because more than one Oracle object must be read from the disk.

Indexes do not make sense if any of the following are true:

- **Small tables:** If the table you want to create an index on is small and contains very few rows, as might be the case with a lookup table, then creating an index actually hinders performance because it may cause Oracle to perform additional I/O. Small tables may actually perform better when full table scans are performed because Oracle may be able to load all blocks for the table in memory and retrieve data from there.
- **Columns not often used in a query:** If creating an index on columns frequently used in a query is a good idea, the reverse must also be true. In other words, if a user is not very likely to query a column's value, do not create an index on that column. Doing so may cause Oracle to use the index when another index may be more appropriate.

- ✦ **Queries return more than 10 percent of data in a table:** If you find that most queries return a large portion of the data in a large table, don't create the index on the large table but rather create indexes on supporting tables that might be joined to it. Oracle can use those indexes for data retrieval from the supporting tables while doing a full table scan on the one without an index. This may be more efficient than creating indexes on the large table.
- ✦ **Heavy data modifications:** If a table is the target of frequent DML operations, create only as many indexes as make sense. If a situation exists in which a lot of changes take place to tables in a database, fewer indexes are the rule. You will probably want to have indexes to enforce PRIMARY KEY and UNIQUE constraints, as well as create indexes on FOREIGN KEY columns, but do not create too many, if any, additional indexes. Doing so slows down write operations and reduces performance.

The general rule for indexes has always been, and should always be, create enough to make data retrieval faster but not too many to cause poor DML performance. Depending upon your environment, you may need to adjust the number of indexes accordingly. If you have a data warehousing or decision-support system that is query intensive, you will create more indexes than if you have an online transaction-processing environment that is very update intensive. Create indexes when you feel they are warranted, but be prepared to drop some and potentially create others if the ones you have do not provide the performance your users expect.

Function-based indexes

A new feature in Oracle 8*i* is the ability to create an index whose key value is the result of an expression. This is quite handy in those situations where expressions are used in the WHERE clause of a query or in the GROUP BY or ORDER BY clauses.

For example, you need to frequently generate a report of the most expensive instructors that you deal with. The query to do so is as follows:

```
SQL> SELECT LastName, FirstName, (PerDiemCost + PerDiemExpenses) AS DailyCost
  2  FROM Instructors
  3  WHERE (PerDiemCost + PerDiemExpenses) > 500
  4  ORDER BY (PerDiemCost + PerDiemExpenses) DESC;

LASTNAME                       FIRSTNAME                      DAILYCOST
------------------------------ ------------------------------ ----------
Cross                          Lisa                                 1000
Ungar                          David                                 800
Harrison                       Michael                               700
Jamieson                       Kyle                                  700
Keele                          Susan                                 650

SQL>
```

Because you are frequently executing this query, and you expect the number of instructors to grow over time, you may want to create an index on the result of the "(PerDiemCost + PerDiemExpenses)" expression. Doing so allows Oracle to use the index to quickly present the data in the right order. To create this function-based index, you execute the following:

```
SQL> CREATE INDEX InstructorCost
  2  ON Instructors ((PerDiemCost + PerDiemExpenses));

Index created.

SQL>
```

After the function-based index is created, Oracle can make use of the index when you execute the preceding SELECT statement.

Privileges for creating indexes

When you need to create an index on a table and you are the owner of the table, you automatically have permission to create the index because you own both objects. However, you cannot create a function-based index even if you own the table.

In order to be able to create a function-based index on tables you own, you must be granted the QUERY REWRITE privilege by the DBA. This privilege tells Oracle that the optimizer may dynamically rewrite a query to make use of the index instead of reading data from the table to perform the calculation. It is used in conjunction with materialized views introduced in Oracle 8*i*.

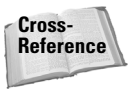
Materialized views are an advanced Oracle 8*i* topic. You are not tested on them or on query rewrites on the "Introduction to Oracle: SQL & PL/SQL" exam. If you want to lean more about materialized views or query rewrites, consult the *Oracle8i Data Warehousing Guide*, a part of the Oracle 8*i* documentation set.

You can also create indexes in another user's schema, or on tables that are in another schema, if you have the CREATE ANY INDEX privilege. This must be granted by the DBA and gives you global carte blanche to create indexes on any table in the database. For obvious reasons, it is not likely to be given to the average user.

If you also want to create function-based indexes on tables in another schema, or in another schema, you must also have the GLOBAL QUERY REWRITE privilege, in addition to the CREATE ANY INDEX privilege. Unless you are responsible for the creation and maintenance of indexes in your database, it is not likely that these privileges will be relinquished to you by the DBA.

Viewing index information

Oracle provides a couple of data dictionary views that you can use to get information on what indexes are created and on which columns of a table they are defined. These are the USER_INDEXES and USER_IND_COLUMNS views.



The structure of the USER_INDEXES and USER_IND_COLUMNS views can be found in Appendix F. This appendix also lists other useful data dictionary views.

To get a listing of all indexes, and the tables and columns on which they have been created, you issue the following command:

```
SQL> col index_name format a25
SQL> col table_name format a20
SQL> col column_name format a18
SQL> SELECT i.index_name, i.table_name, c.column_name, i.uniqueness
  2  FROM USER_INDEXES i, USER_IND_COLUMNS c
  3  WHERE i.index_name = c.index_name
  4  ORDER BY i.table_name;

INDEX_NAME                TABLE_NAME           COLUMN_NAME        UNIQUENES
------------------------- -------------------- ------------------ ---------
BATCHJOBS_JOBID_PK        BATCHJOBS            JOBID              UNIQUE
PK_CLASSID_STUDENTNUMBER  CLASSENROLLMENT      STUDENTNUMBER      UNIQUE
PK_CLASSID_STUDENTNUMBER  CLASSENROLLMENT      CLASSID            UNIQUE
COURSEAUDIT_PK            COURSEAUDIT          COURSENUMBER       UNIQUE
COURSEAUDIT_PK            COURSEAUDIT          CHANGE             UNIQUE
COURSEAUDIT_PK            COURSEAUDIT          DATECHANGED        UNIQUE
PK_COURSENUMBER           COURSES              COURSENUMBER       UNIQUE
INSTRUCTORCOST            INSTRUCTORS          SYS_NC00020$       NONUNIQUE
PK_INSTRUCTORID           INSTRUCTORS          INSTRUCTORID       UNIQUE
PK_LOCATIONID             LOCATIONS            LOCATIONID         UNIQUE
PK_CLASSID                SCHEDULEDCLASSES     CLASSID            UNIQUE
PK_STUDENTNUMBER          STUDENTS             STUDENTNUMBER      UNIQUE
STUDENTLASTNAME           STUDENTS             LASTNAME           NONUNIQUE

13 rows selected.

SQL>
```

As you may have already noticed, the output of this query also returns those indexes that have been automatically created when PRIMARY KEY and UNIQUE constraints were defined on a table. By properly naming the constraints, when you review the output from the preceding query on the data dictionary views, you can easily tell which columns make up the PRIMARY KEY or UNIQUE constraint, as well as the constraint type.

Dropping indexes

When you no longer need an index, you can drop it using the DROP INDEX command. When the index is dropped, Oracle removes any reference to it from the data dictionary and frees any disk space used by the index.

The syntax to drop an index is quite simple:

```
DROP INDEX indexname;
```

To drop the InstructorCost index, you issue the following command:

```
SQL> DROP INDEX InstructorCost;

Index dropped.

SQL>
```

Sequences

Objective

- ✦ Create, maintain, and use sequences

In many applications, it may be necessary to generate chronological numerical values. These numerical values typically are the basis for a PRIMARY KEY or UNIQUE constraint on a table. For example, an order-entry system most likely generates unique numerical order numbers. The order information may be stored in an Orders table with OrderID as a PRIMARY KEY to uniquely identify each order. The challenge is to ensure that unique values are always entered for columns with PRIMARY KEY or UNIQUE constraints. Oracle supports the use of sequences to help in this task.

Sequences are database objects that generate incremental numeric values that are always unique for the named sequence. When you need to insert a new value for a row in a table with a PRIMARY KEY, instead of figuring out what the last number was and adding one to it, you can define a sequence and assign the next value in the sequence. An Oracle database can support many sequences simultaneously so that you can define one for each PRIMARY KEY or UNIQUE column where you need to generate new values.

The benefits of sequences are as follows:

- ✦ **Sequences automatically generate unique numbers:** Each call to assign a sequence's value to a column of a table automatically increments the sequence. There is never a possibility of a duplicate value being used (unless this is what you want and have used the CYCLE option when creating a sequence).

- **Sequences are database objects that can be managed by the DBA or application developer:** If a sequence needs to be restarted or the value range needs to be modified, the DBA re-creates the sequences with the new parameters, and all applications using it get the changes. There is no need to modify application code to take into account the changes.
- **Sequences replace application code:** Instead of writing a procedure to determine the next order number or customer number, using a sequence simply requires a call to increment the sequence value — the number will be unique. Therefore, you do not run the risk of creating a bottleneck in the database by locking rows in a table holding the next number of an invoice or order, or risk having duplicate values when using a query to select the MAX of the order number of invoice number in a table within application code.
- **Sequences can be cached for speed:** Oracle enables you to cache sequence values in memory. Doing so reduces the number of reads Oracle has to perform on the data dictionary to determine the next sequence value. However, in the event of an database instance crash, any cached values are lost, and you could have gaps in the sequence.

Do not cache sequence values in those situations where you cannot live with gaps in the chronological numbering of data. For example, do not use a sequence for generating check numbers since caching the sequence will lose any cached values in the event of an instance crash. In order to reset the sequence to the correct value, you need to re-create the sequence.

The CREATE SEQUENCE command

The Oracle CREATE SEQUENCE command is used to create sequences in the database. The syntax for the command is as follows:

```
CREATE SEQUENCE sequencename
   [INCREMENT BY num]
   [START WITH num]
   [MAXVALUE num | NOMAXVALUE]
   [MINVALUE num | NOMINVALUE]
   [CYCLE | NOCYCLE]
   [CACHE num | NOCACHE]
```

The meaning of the various parameters is described in Table 7-4.

Table 7-4
CREATE SEQUENCE Parameters

Parameter	***Description***
INCREMENT BY *num*	This parameter instructs Oracle to increment the value of the sequence by the number specified by *num***.** The default increment value is 1, which means that each sequence value assigned is 1 greater than the previous. If you want to leave gaps in the chronological numbering of data (that is, you want values to go up in increments of 5), you specify an INCREMENT BY value other than the default of 1. To create a sequence whose values decrement (that is, they go down instead of up), specify an INCREMENT BY of –1 or another negative value. In this case, make sure you specify a high value for the START WITH parameter.
START WITH *num*	Oracle always starts a sequence with 1 as the first value in the sequence, unless a different value is specified for *num* in the START WITH clause. For example, invoice numbers typically start with 1,000 or a higher value. To change the default start value for the sequence to 1,000, you add the START WITH 1000 parameter to the sequence creation command. When you create a decrementing sequence, you should use a very high number for the START WITH parameter.
MAXVALUE *num* NOMAXVALUE	This parameter tells Oracle whether the sequence has a maximum value, after which it may reuse previous values depending upon the setting of the CYCLE/NOCYCLE parameter. The default is not to have a maximum value for a sequence (NOMAXVALUE), which causes it to continue to grow to a maximum of 10^{27} for an ascending sequence, or –1 for a descending sequence.
MINVALUE *num* NOMINVALUE	Specifies the minimum value for a sequence. If NOMINVALUE is specified, 1 is the minimum value for an ascending sequence, while 10^{26} is the minimum value for a descending sequence. In the case of ascending sequences, it is a good idea to specify a MINVALUE that is the same as the START WITH parameter for the sequence. To limit the minimum invoice number to 1,000 for the sequence, you can use the MINVALUE 1000 parameter when creating the sequence.

Continued

Table 7-4 *(continued)*

Parameter	*Description*
CYCLE/NOCYCLE	When an ascending sequence reaches its highest value, or when a descending sequence reaches its lowest value, Oracle does not allow any further values to be generated, and any attempt to do so generates an error. This is the default NOCYCLE behavior. If you want Oracle to continue to reuse sequence values after the sequence limit is reached, you can specify the CYCLE parameter when the sequence is created.
CACHE *num* NOCACHE	Any call to use a sequence value causes a read of data dictionary information. To reduce the overhead of doing this, Oracle enables you to cache sequence values in memory so that they may more easily be retrieved and assigned to data. By default, the value for cache is 20 (that is, Oracle caches 20 sequence values whenever an instance is started). After a sequence value is cached, it is considered by Oracle to be used up. This means that if the instance crashes, any cached values are lost, and you may have a numerical gap in the data. If you do not want to cache sequence values, use the NOCACHE parameter. This has a negative impact on performance because the sequence needs to be read from the data dictionary each time it is used in a value assignment, but using the NOCACHE parameter reduces the possibility of gaps in data using sequences.

To create a sequence that will be used to automatically assign student numbers, starting with 1100 and incrementing the value by 1 for each new student, on the Students table you issue the following command:

```
SQL> CREATE SEQUENCE StudentNumSeq
  2  INCREMENT BY 1
  3  START WITH 1100
  4  MAXVALUE 99999
  5  NOCACHE
  6  NOCYCLE
  7  /

Sequence created.

SQL>
```

Using sequences

The main reason to create sequences is to use them for generating PRIMARY KEY values when INSERTs are made to tables. Oracle provides two pseudo-columns for sequences that make this possible: NEXTVAL and CURRVAL.

Each sequence that you define in your database can have the next value in the sequence assigned to a column by using the NEXTVAL sequence pseudo-column. For example, to add a new student to the Students table and ensure that a unique PRIMARY KEY was generated using the StudentNumSeq sequence, you issue the following command to perform the INSERT:

```
SQL> INSERT INTO Students (StudentNumber, FirstName, LastName, Email)
  2  VALUES (StudentNumSeq.NEXTVAL, 'David','Smith','davids@bradsys.com');

1 row created.

SQL>
```

Using the NEXTVAL pseudo-column in the INSERT statement and prefixing it with the sequence name tells Oracle to get the next value for the sequence and place it in the StudentNumber column during the INSERT operation. The value returned by NEXTVAL is different for each user and is guaranteed to be unique every time, thereby ensuring that no duplicates exist in the data, even when multiple users are adding rows to the table at the same time.

If you want to check the value added to the table, after performing the INSERT, you can check to see what the sequence value used was with the CURRVAL pseudo-column. For example, to see what value was put in the StudentNumber column by the preceding INSERT, you can issue the following command:

```
SQL> SELECT StudentNumSeq.CURRVAL FROM DUAL;

   CURRVAL
----------
      1100

SQL>
```

As you can see, the student is assigned a StudentNumber of 1100, which makes sense because this is the value you said the sequence should start with.

It is important to note that the value of CURRVAL is not populated until after NEXTVAL has been used. In other words, if you try to check the current sequence value by using CURRVAL before assigning the value with NEXTVAL, you get an error from Oracle, as shown here:

```
SQL> SELECT StudentNumSeq.CURRVAL FROM DUAL;
SELECT StudentNumSeq.CURRVAL FROM DUAL
       *
ERROR at line 1:
```

```
ORA-08002: sequence STUDENTNUMSEQ.CURRVAL is not yet defined in this session

SQL>
```

The use of NEXTVAL must be very precise and used only in those situations where you want to increment the sequence value. This means that issuing a statement such as the following causes the sequence to increment, even though it was not assigned to data in a table:

```
SQL> SELECT StudentNumSeq.NEXTVAL FROM DUAL;

   NEXTVAL
----------
      1101

SQL>
```

The rules for using NEXTVAL and CURRVAL are as follows:

- ✦ NEXTVAL and CURRVAL can be used in the column list of a SELECT statement that is not part of a subquery.
- ✦ NEXTVAL and CURRVAL can be used in the column list of a subquery in a INSERT statement, in the form INSERT INTO ... SELECT ... FROM ....
- ✦ NEXTVAL and CURRVAL can be used in the VALUES clause of an INSERT statement to assign values to columns, as shown previously.
- ✦ NEXTVAL and CURRVAL can be used in the SET clause of an UPDATE statement to modify existing data in a table.
- ✦ After NEXTVAL is used, each call to NEXTVAL increments the sequence, and the value used cannot be regained. This means that NEXTVAL always adds the INCREMENT value to the previous sequence value and changes the value of CURRVAL.
- ✦ CURRVAL cannot be used until NEXTVAL has been used.
- ✦ NEXTVAL and CURRVAL cannot be used as part of a view definition. This means that the SELECT statement used to define a view cannot include a reference to NEXTVAL or CURRVAL.
- ✦ You cannot precede NEXTVAL or CURRVAL with the DISTINCT keyword in the column list of a SELECT statement. This generates an error and is not allowed.
- ✦ You cannot use NEXTVAL or CURRVAL in a SELECT statement that includes a GROUP BY, HAVING, or ORDER BY clause.
- ✦ You cannot use NEXTVAL or CURRVAL in a subquery of a SELECT, UPDATE, or DELETE statement. As noted previously, these sequences can be used in the subquery of an INSERT statement.
- ✦ NEXTVAL and CURRVAL cannot be used in the definition of a DEFAULT for a column. Although it would be quite handy to have a PRIMARY KEY with a

DEFAULT that uses a sequence's NEXTVAL, Oracle does not allow it because the sequence is a separate database object, and they cannot be used in the definition of a DEFAULT.

When designing sequences, it is a good idea to create a sequence for each table and column where it will be used. This means that the database schema that you are using for the labs should have a sequence for the StudentNumber, InstructorID, LocationID, and CourseNumber columns. Although not all of these columns in their respective tables require sequences, each is a potential candidate.

Oracle does allow the same sequence to be used to assign values to columns in more than one table. For example, you can create a sequence that would be used to create student numbers and instructor IDs. The uniqueness of the values is spread across both tables, creating gaps in the student number column, as well as in the instructor ID column. This is possible because a sequence is a data dictionary object, which can be accessed by a user and implemented as the user sees fit.

It is possible to have gaps in sequences. This can occur for a number of reasons, including:

- ✦ A user rolled back a transaction after using the sequence in an INSERT with the NEXTVAL pseudo-column. Once the NEXTVAL has been called for a sequence, the value is considered used and cannot be regained. A ROLLBACK does not cause the value used with NEXTVAL to be replaced. Sequence usage does not confirm to the transaction rules of COMMIT, ROLLBACK, and SAVEPOINT. NEXTVAL *always* uses up a sequence value.
- ✦ The Oracle instance has crashed, and the sequence was created with a CACHE parameter. Any cached values for a sequence that are held in memory by Oracle are considered to be used as soon as they are cached. This means that if the instance crashes, those values cannot be reused and are lost. This is the main reason not to cache sequences.
- ✦ If a sequence is used in two or more tables, each table will have some of the sequence values and each will appear to have gaps. This is the way sequences work, indicating that each sequence should be designed to be used in one table and one table only.

Getting information on sequences

Oracle enables you to view information on sequences in your schema by querying the USER_SEQUENCES data dictionary view. There is also a corresponding ALL_SEQUENCES data dictionary view that enables you to view information on sequences that you have been granted access to. For example, if you want to view the name, incremental value, cache number, and next number to be used for the StudentNumSequence, you issue the following command:

```
SQL> SELECT sequence_name, increment_by, cache_size, last_number
  2  FROM USER_SEQUENCES
  3  WHERE sequence_name = 'STUDENTNUMSEQ';
```

```
SEQUENCE_NAME                  INCREMENT_BY CACHE_SIZE LAST_NUMBER
------------------------------ ------------ ---------- -----------
STUDENTNUMSEQ                             1          0        1101

SQL>
```

In the data returned, it is important to note that the LAST_NUMBER column indicates what the next *noncached* sequence value is. This means that if the CACHE_SIZE column has a value other than 0, LAST_NUMBER does not represent the next value that will be assigned to data in the database, but rather the next value that will be assigned to the cache. As mentioned earlier, any sequence values in the cache are considered by Oracle to be used and are retained should the instance crash.

Modifying sequences

Oracle enables you to change a number of the sequence parameters by using the ALTER SEQUENCE command. The following can be changed:

- **The INCREMENT BY value:** Changing the incremental value for a sequence applies to all uses of the sequence after the ALTER SEQUENCE command has been issued.
- **The MAXVALUE for the sequence:** In those situations where you find that you are already reaching the previous maximum value you defined for a sequence, you can raise the bar by modifying the sequence and increasing the ceiling value.
- **The cache status for a sequence:** If the sequence was previously cached, you can increase or decrease the cache size, or turn off caching for the sequence. If the sequence was not cached, you can implement caching to improve performance.
- **The cycle status for a sequence:** If you want to disallow recycling of sequence values after the MAXVALUE or MINVALUE are reached for an ascending or descending sequence, respectively, the ALTER SEQUENCE command enables you to change the cycle parameter.

The syntax for the ALTER SEQUENCE command is as follows:

```
ALTER SEQUENCE sequencename
   [INCREMENT BY num]
   [MAXVALUE num | NOMAXVALUE]
   [MINVALUE num | NOMINVALUE]
   [CYCLE | NOCYCLE]
   [CACHE num | NOCACHE]
```

For example, to modify the StudentNumSeq sequence to cache five values to improve the performance of INSERTs to the Student table, you issue the following command:

```
SQL> ALTER SEQUENCE StudentNumSeq
  2  CACHE 5;

Sequence altered.

SQL>
```

Dropping sequences

To remove a sequence that is no longer required, you issue the DROP SEQUENCE command. For example, to remove the StudentNumSeq sequence, you issue the following command:

```
SQL> DROP SEQUENCE StudentNumSeq;

Sequence dropped.

SQL>
```

After a sequence is dropped, it can no longer be used, and any calls to it fail. Drop sequences only after you have modified any application code that makes use of it.

Synonyms

Objective

- ✦ Create private and public synonyms

Sometime while taking English in school, you were introduced to *Roget's Thesaurus* and told by your English teacher what a wonderful tool it can be. You can look up a word and find other words that have the same or almost the same meaning. These words, you were told, are *synonyms*.

In Oracle, you are also allowed to make use of synonyms. Synonyms are created for objects, such as tables and views. They are used as a form of shorthand for tables, or other objects, with long names (for example, "Emps" for the Employees table or "Teachers" for Instructors). They can also be created to refer to objects that are owned by another user without having to specify the fully qualified name of the object. They are, in short, a form of shorthand.

The syntax to create a synonym is as follows:

```
CREATE [PUBLIC] SYNONYM synonymname
   FOR objectname
```

For example, ScheduledClasses is a long name for a table. If you want to refer to the same table by using the name Classes instead, you can create a synonym as follows:

```
SQL> CREATE SYNONYM Classes
  2  FOR ScheduledClasses;

Synonym created.

SQL>
```

From this point, if you issue a query against the Classes synonym or the ScheduledClasses table, Oracle always operates against the table. For example, a query on the base table ScheduledClasses also has the same result using the synonym, as shown here:

```
SQL> SELECT ClassID, CourseNumber, StartDate, Status
  2  FROM ScheduledClasses;

   CLASSID COURSENUMBER STARTDATE STATUS
---------- ------------ --------- ----------
        50          100 06-JAN-01 Confirmed
        51          200 13-JAN-01 Confirmed
        53          100 14-FEB-01 Hold

SQL> SELECT ClassID, CourseNumber, StartDate, Status
  2  FROM Classes;

   CLASSID COURSENUMBER STARTDATE STATUS
---------- ------------ --------- ----------
        50          100 06-JAN-01 Confirmed
        51          200 13-JAN-01 Confirmed
        53          100 14-FEB-01 Hold

SQL>
```

The use of synonyms is particularly handy when you want to have a shorthand way to refer to a table in another schema. Instead of always qualifying the object name with the schema name, you can create a synonym that does that for you. For example, to access the Customers table in Roger's schema, you can create a synonym as follows (this is only an example and you do not have a Roger schema in your database):

```
SQL> CREATE SYNONYM RogerCusts
  2  FOR Roger.Customers;

Synonym created.

SQL>
```

An important element to consider here is that Oracle does not verify that the object you are creating a synonym for actually exists at the time the synonym is being created. It is possible to create a synonym for an object that does not yet exist in the database. A simple spelling mistake in the creation of a synonym can result in the synonym being created in the database, but an attempt to use it may generate the following error:

```
ERROR at line 1:
ORA-00942: table or view does not exist
```

Always test the synonym immediately after creating it to ensure that it works properly. Failure to do so may make it more difficult to determine the cause of the problem later.

Public synonyms

When you create a synonym in your schema, it is accessible only to you and is considered a private synonym. If you have been granted the CREATE PUBLIC SYNONYM permission, you can create a synonym that can be used by anyone with access to the database. Unlike a private synonym, whose name must be unique within the schema, a public synonym must be unique within the database. For this reason only DBAss have the privilege to create them.

If you want to create a public synonym called Enrollment for the ClassEnrollment table in the Student schema, provided you have the correct permissions, you can issue the following command:

```
SQL>  CREATE PUBLIC SYNONYM Enrollment
  2   FOR Student.ClassEnrollment;

Synonym created.

SQL>
```

If the command failed with the error:

```
CREATE PUBLIC SYNONYM Enrollment
*
ERROR at line 1:
ORA-01031: insufficient privileges
```

it indicates that you have not been granted the CREATE PUBLIC SYNONYM permission.

Getting information on synonyms

Like other objects, information on synonyms may be retrieved by the data dictionary by querying one of the USER_, ALL_, or DBA_ views. In the case of synonyms, the view is USER_SYNONYMS.

To find out the names of all synonyms and the tables they refer to, you can issue the following command:

```
SQL> SELECT synonym_name, table_owner, table_name
  2  FROM USER_SYNONYMS;

SYNONYM_NAME         TABLE_OWNER          TABLE_NAME
-------------------- -------------------- --------------------
CLASSES              STUDENT              SCHEDULEDCLASSES

SQL>
```

The list presented does not include any public synonyms because they are created in the SYS schema so that they may be used by all. To get a list of public synonyms, you must query the ALL_SYNONYMS view. Oracle creates close to synonyms when the database is created; thus, when querying the ALL_SYNONYMS view, you should be precise as to what information you want to retrieve. In the following case, you retrieve all synonyms for objects created in the Student schema:

```
SQL> SELECT owner, synonym_name, table_name, table_owner
  2  FROM ALL_SYNONYMS
  3  WHERE table_owner = 'STUDENT' AND owner='PUBLIC';

SYNONYM_NAME         TABLE_NAME           TABLE_OWNER
-------------------- -------------------- --------------------
ENROLLMENT           CLASSENROLLMENT      STUDENT

SQL>
```

Dropping synonyms

After you have created a private synonym, if you no longer need to make use of it, you can drop it using the DROP SYNONYM command, as follows:

```
SQL> DROP SYNONYM Classes;

Synonym dropped.

SQL>
```

Dropping a synonym removes its reference from the data dictionary.

If you want to drop a public synonym, you must have the DROP PUBLIC SYNONYM permission, which only the DBA has by default. If you attempt to drop a public synonym without the required permission, you get an error as shown here:

```
SQL> DROP PUBLIC SYNONYM Enrollment;
DROP PUBLIC SYNONYM Enrollment
                    *
ERROR at line 1:
```

```
ORA-01031: insufficient privileges

SQL>
```

Key Point Summary

Oracle enables you to create database objects to support your requirements. The objects that you can create include tables, views, indexes, sequences, and synonyms.

- ✦ Object names in Oracle can be up to 30 characters in length and must start with a letter. Object names may also contain numbers (0–9) and the _, #, and $ characters, although the use of # and $ is discouraged.
- ✦ All objects created in Oracle are owned by an Oracle user. Any user who creates database objects is said to be a *schema owner*. The collection of all objects that a user owns is called the *schema*.
- ✦ Object names within a schema must be unique.
- ✦ To create tables to store data, you use the CREATE TABLE command.
- ✦ The ALTER TABLE command enables you to add or remove columns of a table.
- ✦ All columns of a table must have at least two attributes — a name and a datatype. Oracle supports many datatypes including CHAR, VARCHAR2, NCHAR, NVARCHAR2, NUMBER, DATE, BLOB, CLOB, NCLOB, BFILE, RAW, LONG, LONG RAW, ROWID, and UROWID.
- ✦ CHAR, VARCHAR2, and CLOB datatypes store data in the character set of a database. NCHAR, NVARCHAR2, and NCLOB datatypes store data in the National Language character set of a database. Both the character set and National Language character set of a database are specified when the database is created and may not normally be changed.
- ✦ Constraints may be defined on columns of a table and on the table as a whole. Constraints are used to enforce database integrity and business rules. Oracle supports PRIMARY KEY, UNIQUE, FOREIGN KEY, CHECK, and NOT NULL constraints.
- ✦ PRIMARY KEY and UNIQUE constraints automatically create an index on the constraint columns, if one does not already exist, to enforce the uniqueness. FOREIGN KEY constraints do not automatically create an index, but you should create them manually after the constraint is defined.
- ✦ A table may have only one PRIMARY KEY constraint but multiple UNIQUE constraints. PRIMARY KEY constraints ensure that rows are unique in the table and NOT NULL; UNIQUE constraints allow NULLs.
- ✦ Constraints may be dynamically enabled or disabled using the ALTER TABLE command. This capability is useful when you need to perform a mass update or mass insert into a table, which may not be done in such an order to maintain constraint integrity.

- ✦ A DEFAULT may be specified for a column to assign a value, other than NULL, to the column if another value is not specified during an INSERT. DEFAULTs are only applied during INSERT operations and are not maintained thereafter.
- ✦ A VIEW enables you to give a simple or complex SELECT statement a name and store its definition in the data dictionary. Users then can SELECT from the VIEW, which automatically runs the SELECT statement that defines the view.
- ✦ You can perform DML through simple views (that is, views based on a single table with no expressions or functions in the column list), but you may not always be able to perform DML through complex views.
- ✦ To ensure that DML does not put the data outside of the scope of the view, you can create the view using the WITH CHECK OPTION. To prevent DML on a view, create the view with the READ ONLY option.
- ✦ Sequences enable you to automatically generate incremental values, either ascending or descending, which then can be assigned to a column of a table using the NEXTVAL and CURRVAL pseudo-columns for the sequence.
- ✦ Once NEXTVAL is used, the value for the sequence is automatically incremented and cannot be reused. You should use NEXTVAL only when assigning the next value in the sequence to a column in a table.
- ✦ After using NEXTVAL, you can check the value assigned using CURRVAL. CURRVAL has no value until NEXTVAL is used.
- ✦ Caching sequences improves performance because the next set of incremental values for the sequence will be cached in memory. After a sequence value is cached, it is considered used, and an instance crash loses any cached values.
- ✦ Synonyms enable you to create alternate names to reference tables in the database. They can be used as a form of shorthand.
- ✦ Public synonyms can be created only by the DBA, by default, and can be used by anyone connected to the database.
- ✦ Indexes enable you to potentially speed the retrieval of data in the database. Indexes also slow down the performance of DML in the database when they are created on tables that are frequently modified.
- ✦ The right number of indexes to create is as many as are needed for queries to work faster but not too many that DML operations slow to a crawl.
- ✦ Create indexes on columns that form part of a join condition, are frequently referenced in the WHERE clause of a SQL statement, or used in the GROUP BY, HAVING, or ORDER BY clauses.
- ✦ Oracle 8*i* enables you to create function-based indexes, which store the result of an expression instead of a column value. In order to create function-based indexes, you need to be granted the QUERY REWRITE permission.

✦ ✦ ✦

STUDY GUIDE

This section will enhance your understanding of the material presented in this chapter. Answer the questions and then work through the lab exercises in order to feel more comfortable with the material.

Assessment Questions

1. On which line will the following CREATE TABLE statement fail? (Choose the best answer.)

```
1  CREATE TABLE MyTable (
2  IdColumn number(5) NULL,
3  Name varchar2(30),
4  Status varchar2(10);
```

A. 1

B. 2

C. 3

D. 4

E. The statement will not fail.

2. Which permissions are required to create a function-based index on a table in another schema? (Choose three answers.)

A. CREATE INDEX

B. CREATE ANY TABLE

C. SELECT on the table being indexed

D. GLOBAL QUERY REWRITE

E. QUERY REWRITE

F. ALTER TABLE

G. CREATE ANY INDEX

3. If you need to create a view that returns data so that it can be used for "Top-N" analysis, which of the following clauses must you include in the view definition? (Choose the best answer.)

A. HAVING

B. ORDER BY

C. TOP-N

D. ROWNUM

E. GROUP BY

F. GROUPING

4. If you are connected to your Oracle instance as the user "Student", which of the following database objects can you drop, assuming a default configuration? (Choose all correct answers.)

A. Student.Courses

B. Scott.Courses

C. Student.Table2

D. USER_CONSTRAINTS

E. Public Synonym ENROLLMENT defined on Student.ClassEnrollment

5. Which of the following benefits do indexes provide? (Choose two answers.)

A. Speed data inserts.

B. May provide data in sorted order.

C. Speed data updates.

D. Speed data deletes.

E. Speed view creation.

F. Speed data retrieval.

6. Which of the following commands can be used to add a constraint to an existing table called Courses? (Choose all correct answers.)

A. CREATE CONSTRAINT ...

B. ALTER TABLE Courses MODIFY ...

C. ALTER TABLE Courses CONSTRAINT ...

D. ALTER TABLE Courses ADD ...

E. MODIFY TABLE Courses CONSTRAINT ...

F. UPDATE TABLE Courses SET CONSTRAINT ...

7. You need to create a view through which users can update employee records. The view will extract data from a number of tables, but the INSERT, UPDATE, or DELETE can take place only on one of the underlying tables. Which of the following clauses will cause the updates to fail? (Choose all correct answers.)

A. CREATE VIEW

B. SELECT DISTINCT

C. WHERE

D. GROUP BY

E. HAVING

F. ORDER BY

8. You create a sequence to be used to assign incremental order numbers in the Orders table. You create the sequence using the following definition:

```
CREATE SEQUENCE OrderNumSequence
START WITH 1000
INCREMENT BY 1
MAXVALUE 100000000
NOCYCLE
```

The instance accessing the orders database crashes. After instance restart, you determine that the highest order number is 1277. The LAST_NUMBER column of the USER_SEQUENCES view has a value of 1320. What is the next order number to be assigned when using the NEXTVAL psuedo-column for the sequence? (Choose the best answer.)

A. 1280

B. 1300

C. 1301

D. 1278

E. 1281

9. In the following ALTER TABLE command, which line causes the command to fail? (Choose the best answer.)

```
1    ALTER TABLE Employees
2    MODIFY (EmployeeID number
3          PRIMARY KEY NOT NULL
4          DEFAULT EmpIDSequence.NEXTVAL)
```

A. 1

B. 2

C. 3

D. 4

E. The statement will process correctly.

10. Which of the following constraint options is the default when you attempt to add a constraint to an existing table? (Choose two answers.)

A. ENABLE

B. DISABLE

C. FORCE

D. NOFORCE

E. VALIDATE

F. NOVALIDATE

11. Which of the following system views can you query to determine which views are in your schema? (Choose all correct answers.)

A. USER_TABLES

B. USER_VIEWS

C. ALL_VIEWS

D. USER_CATALOG

E. MY_VIEWS

12. If you want to remove references to a column in an existing table without reorganizing the entire table, which command do you issue? (Choose the best answer.)

A. ALTER TABLE ... DROP COLUMN ...

B. ALTER TABLE ... DROP UNUSED COLUMNS

C. ALTER TABLE ... UNUSED COLUMN ...

D. ALTER TABLE ... SET UNUSED ...

E. ALTER TABLE ... MODIFY ...

Scenarios

1. You have just been hired as a junior DBA for a national training organization. As one of your first tasks, you have been asked to assist the sales department in keeping better track of their data. They need to be able to perform the following:

- Create a listing of the instructors who have taught the most courses over the last six months.
- Create a listing of the courses generating the highest revenue over the last year, with the output showing a subtotal by course and month.

- The reporting functions, which will grow as the company increases in size, must be performed so that they do not impact other users making changes to the existing data.

What database objects would you create to help support these requirements?

2. A major distributor of houseware products has been using a part-numbering scheme in their Products table based upon the manufacturer and product description of the products they sell. This has caused problems when manufacturers release products with similar names because the clerks entering the new products often were informed that the product code they were creating already exists.

   The DBA has also mentioned to you that the objects in the database are owned by many users, and the objects have names close to the 30-character limit that Oracle allows. Referencing those objects and performing queries requires a lot of typing and is time consuming.

   What suggestions would you make to solve these problems?

Lab Exercises

In the lab exercises, you will create and manage tables, views, constraints, and other database objects. You should use the same database used in labs in previous chapters because you need to reference the existing tables in the Student schema.

Lab 7.1 Creating and Managing Tables

1. Open SQL*Plus and connect to your instance using the Student account with the password oracle.
2. Create a table called SalesPersons that will hold salesperson information. The table should have the following structure:

Column Name	*Datatype*	*Length*	*Null?*
SalesPersonID	number	5	No
FirstName	varchar2	15	No
LastName	varchar2	20	No
Address	varchar2	30	Yes
City	varchar2	25	Yes
State	char	2	Yes

Continued

Column Name	*Datatype*	*Length*	*Null?*	*(continued)*
PostalCode	varchar2	10	Yes	
Telephone	varchar2	15	Yes	
Email	varchar2	40	Yes	
Salary	number	9,2	Yes	
Commission	number	3	No	
Comments	varchar2	2000	Yes	

3. Insert data into the Salespersons table with the following information:

Column Name	*Datatype*	*Length*	*Null?*
SalesPersonID	1001	1002	1003
FirstName	Adrian	Natasha	Erin
LastName	Newey	Konkle	Smith
Address	101 Williams Rd.	117 Hennesy Crt.	
City	Dorsett	Toronto	
State	NH	ON	
PostalCode	12123	L4M 3C7	
Telephone	(555) 244-5523	(416) 555-1313	
EMail	Anewey@bradsys.com		
Salary	2000.00	2200.00	
Commission	20	17	35
Comments			

Add more records into the table if you wish, but ensure that the preceding information is inserted. Verify your data by querying the table.

4. Modify the SalesPersons table so that a MiddleInitial column is added with a datatype of VARCHAR2 and length 3, and that NULLs are allowed.

5. Modify the Email address column so that if another email address is not specified when a record is created, an email address of "sales@bradsys.org" is automatically entered.

Insert a new record of your choosing without an email address to verify that the email address is being assigned properly.

6. Create a temporary table called TempClassEnroll that will be used by your application to hold a class enrollment record while a salesperson is on the phone with a student. The information in the temporary table should have the exact same structure as that ClassEnrollment table.

   Verify that the structure of both tables is the same by querying the data dictionary.

Lab 7.2 Creating and Managing Constraints

1. Modify the TempClassEnroll table to include the exact same FOREIGN KEY constraints as those of the ClassEnrollment table. What happened? Why?
2. Create a PRIMARY KEY constraint on the SalesPersonID column of the SalesPersons table.
3. You need to ensure that a commission rate for a salesperson is no more than 50 percent of the sale amount. Modify the SalesPersons table to include this constraint. Attempt to modify the commission for Adrian Newey to 65 to test your constraint

Lab 7.3 Creating Other Database Objects

1. You need to ensure that each row in the SalesPersons table has a unique SalesPersonID and that these IDs increment chronologically by a value of 1. Create the appropriate database object to satisfy this requirement.
2. You determine that users frequently query information in the Students, Instructors, and SalesPersons tables using the LastName columns in the respective table. Create a database object that will speed queries on the table when this column is referenced in the WHERE clause.
3. Your manager has asked for an enrollment count for each class in all locations. This information must be generated each week. Create a database object that can be queried by your manager to more easily generate this data when needed.

Answers to Chapter Questions

Chapter Pre-Test

1. When creating a table, you must specify a table name that is unique in the schema of the user creating the table. The table must also contain at least one column, for which a name and datatype must be specified.

2. When you define a DEFAULT on a database column, Oracle uses only the DEFAULT when an INSERT is made into the table and only if another value for the column was not explicitly specified in the INSERT statement. DEFAULTs are used to assign values to a database column if no other value is specified at INSERT time. The DEFAULT is not used during an UPDATE operation.

3. You should create sufficient indexes on a table so that queries perform well, but not too many indexes so that DML operations suffer greatly in performance. In general, indexes should be created on columns used to join tables (that is, all columns in a PRIMARY KEY/FOREIGN KEY relationship), columns frequently used in a WHERE clause, or columns often used in the GROUP BY, HAVING, or ORDER BY clauses of a SQL statement. You should always create indexes on FOREIGN KEY columns. Oracle automatically creates indexes on columns with PRIMARY KEY or UNIQUE constraints.

4. In Oracle 8*i* it is now possible to include the ORDER BY clause in a view definition. This is useful if you need to perform "top-N" analysis, such as finding the ten best customers or the ten worst salespeople. You can define a view with the SELECT statement required to return the data in the appropriate order, and then use the ROWNUM pseudo-column to retrieve only the number of sorted rows you desire from the view.

5. Because you have three tables (Customers, Order, Suppliers) and each table must have a unique number generated to identify its contents and support PRIMARY KEY or UNIQUE constraints, you should create three sequences — one for each table's data. This way you do not have gaps in sequences, and the data makes chronological sense for the unique identification columns. You can create a single sequence, but this distributes the sequence values across all three tables creating gaps in each.

6. By default, only the DBA has the CREATE PUBLIC SYNONYM permission. This means that only DBAs are allowed to create PUBLIC synonyms. In order for other users to be able to create or drop a PUBLIC synonym, they must be granted the CREATE PUBLIC SYNONYM or DROP PUBLIC SYNONYM permission by the DBA.

7. Caching sequences improves performance in environments where rapid DML is common, such as order-entry systems and other online transaction-processing environments. Instead of requiring Oracle to get the next sequence value from the data dictionary and then increment it, caching a sequence preincrements a number of sequence values and touches the data dictionary only when the cache is exhausted. This reduces the calls to the data dictionary and improves performance.

   Caching of sequences can cause gaps in the sequence values, because an instance crash loses any cached values. After a sequence value is cached, it is considered used by Oracle and cannot be regained.

8. A PRIMARY KEY constraint ensures that the values for the column or columns on which the constraint is defined are unique within the table and NOT NULL. A UNIQUE constraint ensures that the values for the column or columns on which the constraint is defined are unique within the table or NULL. You can define only a single PRIMARY KEY constraint on a table, whereas a table may support multiple UNIQUE constraints.
9. Oracle automatically assigns a name to a constraint when you do not do so when defining the constraint. The constraint name is in the format of SYS_C*nnnnnn*, such as SYS_C002037. Not naming constraints is a bad idea because Oracle returns a constraint name in the error message when the constraint is violated. Naming the constraint enables you to more easily determine which constraint has been violated when you receive an error message.
10. Columns defined using the CHAR datatype store data using the character set specified when the database was created. Columns using the NCHAR datatype store data using the National Language character set specified when the database was created. Both the CHAR and NCHAR datatypes store character data but differ in which characters they can store.
11. Every object you create in Oracle must start with a letter (a–z or A–Z). Other valid characters for naming objects include numbers (0–9) and the symbols _, #, and $. The use of # and $ is discouraged. Object names in Oracle can be up to 30 characters long and are not case sensitive. In fact, Oracle converts all names to uppercase when storing them in the data dictionary.

Assessment Questions

1. **D** — The syntax for the CREATE TABLE statement is not properly terminated with a closing parenthesis ")". For this reason, the last line of the command fails. You should also explicitly specify whether or not a column will allow NULL for each column of the table, as was done for IdColumn.
2. **C, D, G** — In order to create an index in a schema other than your own, you need the CREATE ANY INDEX privilege to create the index in another user's schema, and the SELECT permission on the table on whose columns the index will be created. Because the index will also be function based in somebody else's schema, you need the GLOBAL QUERY REWRITE privilege.

   If the index were in your schema, you would need the CREATE INDEX and QUERY REWRITE privileges. If the table on whose columns the index was being created was not in your schema, you would also need SELECT permissions on the table whose columns would be referenced in the index.
3. **B** — "Top-N" analysis requires that the data be presented in sorted order. For this reason, you need an ORDER BY clause in the SELECT statement portion of the view definition. In most cases, a GROUP BY and a HAVING clause also are used, but they are not required.

4. **A, C** — Assuming a default configuration where you have not been provided any extraordinary privileges, you can drop objects in your own schema. The only two objects listed that are owned by Student are Student.Courses and Student.Table2. The Scott.Courses table is owned by another user, while USER_CONSTRAINTS and public synonyms can be dropped only by the DBA. It is not recommended, even if you have DBA privileges, that you drop data dictionary objects, such as USER_CONSTRAINTS.

5. **B, F** — Indexes are primarily used to speed data retrieval operations. Using an index in a query can also retrieve the data in sorted order, assuming the query is not too complex or specifies another sort order with an ORDER BY. Indexes always reduce the speed of data modifications because Oracle must update the data and index blocks for each data modification.

6. **B, D** — In order to add a constraint to an existing table, you can use the ALTER TABLE ... ADD command, which can be used to add a PRIMARY KEY, UNIQUE, CHECK, or FOREIGN KEY constraint to the table. You can also use the ALTER TABLE ... MODIFY syntax to add a NOT NULL constraint to an existing column. All of the other commands are not valid in Oracle.

7. **B, D** — If you want to perform DML through a view, you cannot have the DISTINCT keyword in the column list of the view definition. You are also not allowed to have a GROUP BY in the view definition because this clause causes data aggregation and does not allow individual records to be projected through the view.

8. **B** — The next value that will be used when the NEXTVAL pseudo-column is used to assign a value to a table is 1300. This is because USER_SEQUENCES reported 1320 in the LAST_NUMBER column. The sequence has an incremental value of 1 and a default CACHE value of 20. Because 20 values are being cached, the LAST_NUMBER column reports only the next uncached value (in this case, 1320), which means that the first cached value is 1300.

9. **C** — The line reading PRIMARY KEY NOT NULL will cause the ALTER TABLE command to fail. This is because you cannot modify a column to include a PRIMARY KEY constraint. In order to add the PRIMARY KEY constraint, you need to use the ADD clause of the ALTER TABLE command.

   The definition of the DEFAULT also uses a NEXTVAL pseudo-column of the EmpIDSequence sequence, which is also not allowed but is not the primary cause of the failure. The PRIMARY KEY definition is interpreted and deemed invalid by Oracle before it even parses the DEFAULT clause.

10. **A, E** — The default state of all constraints added to an existing table is always ENABLE and VALIDATE. This is to ensure backward compatibility with previous versions of Oracle, as well as to ensure that all data satisfies constraint conditions. FORCE and NOFORCE are not valid options for constraint definition.

11. **B, C, D** — The USER_VIEWS view contains all the views in a your schema. The ALL_VIEWS view contains information about views in your schema and all other views in other schemas that you have been granted permissions to. The USER_CATALOG view is an ANSI-standard view that provides information on all tables and views in your schema.
12. **D** — The ALTER TABLE ... SET UNUSED ... command can be used to mark a column as unusable without forcing a structural reorganization of the table. The ALTER TABLE ... DROP COLUMN and ALTER TABLE ... DROP UNUSED COLUMNS commands remove the column as well as reorganize the data. None of the other examples are valid Oracle commands.

Scenarios

1. In order to satisfy the requirements that have been outlined, the best course of action includes:
    - Create a view that with the proper SQL syntax to provide a listing of instructors who have taught courses over the last six months. Your view definition might look something like this:

```
SQL> CREATE OR REPLACE VIEW InstructorUsage AS
  2  SELECT     I.InstructorID, Name,
  3           COUNT(DISTINCT S.ClassID) "Classes Taught"
  4  FROM Instructors I, ScheduledClasses S, ClassEnrollment C,
  5           (SELECT InstructorID, (FirstName || ' ' || LastName) Name
  6                   FROM Instructors) N
  7  WHERE      I.InstructorID = S.InstructorID
  8  AND        I.InstructorID = N.InstructorID
  9  AND        S.ClassID = C.ClassID
 10  AND        UPPER(S.Status) = 'CONFIRMED'
 11  AND        S.StartDate BETWEEN ADD_MONTHS(SYSDATE,-6) AND SYSDATE
 12  GROUP BY I.InstructorID, Name
 13  ORDER BY COUNT(DISTINCT S.ClassID) DESC, Name;

View created.

SQL> col name format a40
SQL> SELECT * FROM InstructorUsage;

INSTRUCTORID NAME                                     Classes Taught
------------ ---------------------------------------- --------------
         100 David Ungar                                           1
         200 Lisa Cross                                            1

SQL>
```

    - Create a view that summarizes the revenue generated from each course by querying the ClassEnrollment table and joining it to the ScheduledClasses table to report only on those classes that have run and been confirmed. You also need to exclude those enrollments that were canceled. The view definition might look something like this:

```
SQL> CREATE OR REPLACE VIEW ClassRevenue AS
  2  SELECT     S.CourseNumber,  CourseName,
  3           TO_CHAR(SUM(Price), '$99,999.99') "Course Revenue"
  4  FROM       ScheduledClasses S, ClassEnrollment C,
  5           (SELECT CourseNumber, CourseName
  6                   FROM Courses) N
  7  WHERE      S. CourseNumber = N.CourseNumber
  8  AND        S.ClassID = C.ClassID
  9  AND        UPPER(S.Status) = 'CONFIRMED'
 10  AND        S.StartDate BETWEEN ADD_MONTHS(SYSDATE,-12) AND SYSDATE
 11  GROUP BY S.CourseNumber, CourseName
 12  ORDER BY SUM(Price) DESC, CourseName
 13  /

View created.

SQL> col CourseName format a40
SQL> SELECT * FROM ClassRevenue;

COURSENUMBER COURSENAME                                Course Reve
------------ ---------------------------------------- -----------
         200 Database Performance Basics               $11,500.00
         100 Basic SQL                                  $5,750.00

SQL>
```

- In order to reduce the impact on users making changes to the database, you should create copies of the tables with the data you need and create the views on the copied tables. You can do this by using the CREATE TABLE ... AS SELECT ... syntax of the CREATE TABLE command for each relevant table. You then populate those tables with the data from the production system periodically.

2. In order to make the entry of new products more efficient, you should recommend that a numeric code be used to uniquely identify products for the company. The part number column should be changed to include a primary key. You should also create a sequence that allows for the population of part number column with the next incremental part number using the NEXTVAL pseudo-column of the sequence in the INSERT statement.

   To make it easier to reference objects in other schemas with shorter names, synonyms can be created for each object that must be referenced. If all users need to manipulate many objects, the DBA can create public synonyms that can be accessed by all users. Permissions on the tables to which the synonyms refer will prevent users seeing data that they should not be allowed to see.

Lab Exercises

Lab 7.1 Creating and Managing Tables

1. Open SQL*Plus and connect to your instance using the Student account with the password oracle.
2. Create a table called SalesPersons that holds salesperson information. The table should have the following structure:

Column Name	*Datatype*	*Length*	*Null?*
SalesPersonID	number	5	No
FirstName	varchar2	15	No
LastName	varchar2	20	No
Address	varchar2	30	Yes
City	varchar2	25	Yes
State	char	2	Yes
PostalCode	varchar2	10	Yes
Telephone	varchar2	15	Yes
EMail	varchar2	40	Yes
Salary	number	9,2	Yes
Commission	number	3	No
Comments	varchar2	2000	Yes

```
SQL> CREATE TABLE SalesPersons (
  2  SalesPersonID    number(5)       NOT NULL,
  3  FirstName        varchar2(15)    NOT NULL,
  4  LastName         varchar2(20)    NOT NULL,
  5  Address          varchar2(30)    NULL,
  6  City             varchar2(25)    NULL,
  7  State            char(2)         NULL,
  8  PostalCode       varchar2(10)    NULL,
  9  Telephone        varchar2(15)    NULL,
 10  Email            varchar2(40)    NULL,
 11  Salary           number(9,2)     NULL,
 12  Commission       number(3)       NOT NULL,
 13  Comments         varchar2(2000)  NULL
 14  );

Table created.

SQL>
```

3. Insert data into the Salespersons table with the following information:

Column Name	*Datatype*	*Length*	*Null?*
SalesPersonID	1001	1002	1003
FirstName	Adrian	Natasha	Erin
LastName	Newey	Konkle	Smith
Address	101 Williams Rd.	117 Hennesy Crt.	
City	Dorsett	Toronto	
State	NH	ON	
PostalCode	12123	L4M 3C7	
Telephone	(555) 244-5523	(416) 555-1313	
EMail	Anewey@bradsys.org		
Salary	2000.00	2200.00	
Commission	20	17	35
Comments			

Add more records into the table if you wish, but ensure that the preceding is inserted. Verify your data by querying the table.

```
SQL> INSERT INTO SalesPersons VALUES
  2  (1001, 'Adrian', 'Newey', '101 Williams Rd.', 'Dorsett', 'NH',
  3  '12123', '(555) 244-5523', 'Anewey@bradsys.org', 2000, 20, NULL);

1 row created.

SQL> INSERT INTO SalesPersons VALUES
  2  (1002, 'Natasha', 'Konkle', '117 Hennesy Crt.', 'Toronto', 'ON',
  3  'L4M 3C7', '(416) 555-1313', NULL, 2200.00, 17, NULL);

1 row created.


SQL> INSERT INTO SalesPersons (SalesPersonID, FirstName, LastName, Commission)
  2  VALUES (1003, 'Erin', 'Smith', 35);

1 row created.


SQL> set pagesize 100
SQL> SELECT * FROM SalesPersons;
```

```
SALESPERSONID FIRSTNAME       LASTNAME
------------- --------------- --------------------
ADDRESS                        CITY                      ST POSTALCODE
------------------------------ ------------------------- -- ----------
TELEPHONE       EMAIL                                        SALARY COMMISSION
--------------- ---------------------------------------- ---------- ----------
COMMENTS
--------------------------------------------------------------------------------
         1001 Adrian           Newey
101 Williams Rd.               Dorsett                   NH 12123
(555) 244-5523  Anewey@bradsys.com                             2000         20


         1002 Natasha          Konkle
117 Hennesy Crt.               Toronto                   ON L4M 3C7
(416) 555-1313                                                 2200         17

         1003 Erin             Smith

                                                                             35

SQL>
```

4. Modify the SalesPersons table so that a MiddleInitial column is added with a datatype of VARCHAR2 and a length 3 and that NULLs are allowed.

```
SQL> ALTER TABLE SalesPersons ADD
  2  (MiddleInitial varchar2(3) NULL);

Table altered.

SQL>
```

5. Modify the email address column so that if another email address is not specified when a record is created, an email address of "sales@bradsys.org" is automatically entered.

```
SQL> ALTER TABLE SalesPersons MODIFY
  2  (EMail varchar2(40) DEFAULT 'sales@bradsys.org');

Table altered.

SQL>
```

Insert a new record of your choosing without an email address to verify that the email address is being assigned properly.

```
SQL> INSERT INTO SalesPersons
  2  (SalesPersonID, FirstName, LastName, Commission)
  3  VALUES (1004, 'Arthur', 'Jones', 35);
```

```
1 row created.

SQL> SELECT SalesPersonID, FirstName, LastName, Email
  2  FROM SalesPersons;

SQL> SELECT FirstName, LastName, EMail FROM SalesPersons;

FIRSTNAME       LASTNAME             EMAIL
--------------- -------------------- -------------------------
Adrian          Newey                Anewey@bradsys.org
Natasha         Konkle
Erin            Smith
Arthur          Jones                sales@bradsys.org

SQL>
```

6. Create a temporary table called TempClassEnroll that can be used by your application to hold a class enrollment record while a salesperson is on the phone with a student. The information in the temporary table should have the exact same structure as that ClassEnrollment table.

```
SQL> CREATE GLOBAL TEMPORARY TABLE TempClassEnroll AS
  2  SELECT * FROM ClassEnrollment WHERE 0=1;

Table created.

SQL>
```

Verify that the structure of both tables is the same by querying the data dictionary.

```
SQL> DESC ClassEnrollment;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 CLASSID                                   NOT NULL NUMBER(38)
 STUDENTNUMBER                             NOT NULL NUMBER(38)
 STATUS                                    NOT NULL CHAR(10)
 ENROLLMENTDATE                            NOT NULL DATE
 PRICE                                     NOT NULL NUMBER(9,2)
 GRADE                                              CHAR(4)
 COMMENTS                                           VARCHAR2(2000)

SQL> DESC TempClassEnroll;
 Name                                      Null?    Type
 ----------------------------------------- -------- ----------------------------
 CLASSID                                   NOT NULL NUMBER(38)
 STUDENTNUMBER                             NOT NULL NUMBER(38)
 STATUS                                    NOT NULL CHAR(10)
 ENROLLMENTDATE                            NOT NULL DATE
 PRICE                                     NOT NULL NUMBER(9,2)
 GRADE                                              CHAR(4)
 COMMENTS                                           VARCHAR2(2000)

SQL>
```

Another query that you can use to compare columns visually is:

```
SQL> SELECT table_name, column_name, data_type, data_length, nullable
  2  FROM USER_TAB_COLUMNS
  3  WHERE table_name IN ('CLASSENROLLMENT','TEMPCLASSENROLL')
  4* ORDER BY column_name;

TABLE_NAME           COLUMN_NAME                    DATA_TYPE  DATA_LENGTH N
-------------------- ------------------------------ ---------- ----------- -
CLASSENROLLMENT      CLASSID                        NUMBER              22 N
TEMPCLASSENROLL      CLASSID                        NUMBER              22 N
CLASSENROLLMENT      COMMENTS                       VARCHAR2          2000 Y
TEMPCLASSENROLL      COMMENTS                       VARCHAR2          2000 Y
CLASSENROLLMENT      ENROLLMENTDATE                 DATE                 7 N
TEMPCLASSENROLL      ENROLLMENTDATE                 DATE                 7 N
CLASSENROLLMENT      GRADE                          CHAR                 4 Y
TEMPCLASSENROLL      GRADE                          CHAR                 4 Y
CLASSENROLLMENT      PRICE                          NUMBER              22 N
TEMPCLASSENROLL      PRICE                          NUMBER              22 N
CLASSENROLLMENT      STATUS                         CHAR                10 N
TEMPCLASSENROLL      STATUS                         CHAR                10 N
CLASSENROLLMENT      STUDENTNUMBER                  NUMBER              22 N
TEMPCLASSENROLL      STUDENTNUMBER                  NUMBER              22 N

14 rows selected.

SQL>
```

Lab 7.2 Creating and Managing Constraints

1. Modify the TempClassEnroll table to have the exact same FOREIGN KEY constraints as those of the ClassEnrollment table. What happened? Why?

```
SQL> ALTER TABLE TempClassEnroll
  2  ADD CONSTRAINT TempEnroll_ClassID
  3* FOREIGN KEY (ClassID) REFERENCES Classes.ClassID;

ALTER TABLE TempClassEnroll
*
ERROR at line 1:
ORA-14455: attempt to create referential integrity constraint
on temporary table

SQL>
```

You cannot create FOREIGN KEY constraints on temporary tables. FOREIGN KEY constraints can only be created on permanent tables.

2. Create a PRIMARY KEY constraint on the SalesPersonID column of the SalesPersons table.

```
SQL> ALTER TABLE SalesPersons
  2  ADD CONSTRAINT SalesPersons_PK
  3  PRIMARY KEY (SalesPersonID);

Table altered.

SQL>
```

3. You need to ensure that a commission rate for a salesperson is no more than 50 percent of the sale amount. Modify the SalesPersons table to include this constraint. Attempt to modify the commission for Adrian Newey to 55 to test your constraint.

```
SQL> ALTER TABLE SalesPersons
  2  ADD CONSTRAINT SalesPersons_Comm_CK
  3* CHECK (Commission <= 50);

Table altered.

SQL> UPDATE SalesPersons
  2  SET Commission = 65
  3   WHERE SalesPersonID = 1001;
UPDATE SalesPersons
*
ERROR at line 1:
ORA-02290: check constraint (STUDENT.SALESPERSONS_COMM_CK)
violated

SQL>
```

Lab 7.3 Creating Database Objects

1. You need to ensure that each row in the SalesPersons table has a unique SalesPersonID and that these IDs increment chronologically by a value of 1. Create the appropriate database object to satisfy this requirement.

   Insert a new row into the SalesPersons table to ensure that the object you created is working properly.

```
SQL> CREATE SEQUENCE SalesPersonIDSequence
  2  START WITH 1005
  3  INCREMENT BY 1
  4  MAXVALUE 99999
  5  NOCYCLE
  6  NOCACHE;

Sequence created.

SQL> INSERT INTO SalesPersons
  2  (SalesPersonID, FirstName, LastName, Commission)
  2  VALUES
  3  (SalesPersonIDSequence.NEXTVAL, 'Robert', 'Miller', 30);
```

```
1 row created.

SQL> commit;

Commit complete.

SQL> SELECT SalesPersonID, LastName FROM SalesPersons;

SALESPERSONID LASTNAME
------------- --------------------
         1001 Newey
         1002 Konkle
         1003 Smith
         1004 Jones
         1005 Miller

SQL>
```

2. You determine that users frequently query information in the Students, Instructors, and SalesPersons tables using the LastName columns in the respective table. Create a database object that can speed queries on the table when this column is referenced in the WHERE clause.

```
SQL> CREATE INDEX Instructors_LastName_IDX
  2  ON Instructors (LastName);

Index created.

SQL> CREATE INDEX SalesPersons_LastName_IDX
  2  ON SalesPersons (LastName);

Index created.

SQL> CREATE INDEX Students_LastName_IDX
  2  ON Students (LastName);

Index created.

SQL>
```

3. Your manager has asked for an enrollment count for each class in all locations. This information must be generated each week. Create a database object that can be queried by your manager to more easily generate this data when needed.

```
SQL> CREATE OR REPLACE VIEW ClassEnrollmentView AS
  2  SELECT S.LocationID Location, S.ClassRoomNumber Room,
  3          S.CourseNumber Course, S.StartDate,
  4          COUNT(StudentNumber) Enrolled
  5  FROM ScheduledClasses S, ClassEnrollment C
  6  WHERE S.ClassID = C.ClassID
  7  GROUP BY S.LocationID, S.ClassRoomNumber,
  8          S.ClassID, S.CourseNumber, S.StartDate
  9  ORDER BY S.LocationID, S.StartDate;
```

```
View created.

SQL> SELECT * FROM ClassEnrollmentView;

  LOCATION       ROOM     COURSE STARTDATE   ENROLLED
---------- ---------- ---------- --------- ----------
       100          4        100 06-JAN-01          3
       300          1        200 13-JAN-01          3
       300          2        100 14-FEB-01          1

SQL>
```

CHAPTER 8

Configuring Security in Oracle Databases

EXAM OBJECTIVES

- Controlling user access
  - Create users
  - Create roles to ease setup and maintenance of the security model
  - Use the GRANT and REVOKE statements to grant and revoke object privileges

CHAPTER PRE-TEST

1. What is the difference between a schema user and a nonschema user?
2. When a user is granted permission to update data in a table, is it possible to limit which columns the user is able to change?
3. What is a role, and how can it be used to streamline security administration?
4. Is it possible to grant a role to another role?
5. After a user creates a table, who is able to query the table?
6. What happens when permission on a view is granted WITH GRANT OPTION?
7. Which data dictionary view do you query to determine what permissions you have been granted on tables created by other users?
8. When another user references your table in a FOREIGN KEY constraint, what happens if you attempt to remove the user's privileges on your table?
9. What command is used to remove system privileges from a user?
10. How do you change your Oracle password?

✦ Answers to these questions can be found at the end of the chapter. ✦

Are databases accessed by a single person or by more than one individual? If the answer is more than one, which is typically the case in most organizations, then it is necessary to control access to those objects through the use of permissions. Furthermore, each individual that requires access should have a user account that may be used to track his or her access and to ensure that a user cannot perform actions that are not permitted. Security in an Oracle database is responsible for all of these things.

By default, Oracle does not allow anyone, except the Oracle users that exist at database creation time, to perform any actions on the database. In fact, the basic Oracle security model is that "all actions that are not explicitly permitted are implicitly denied." This means that Oracle disallows anything that you try to do unless the DBA (database administrator — an Oracle user) or the owner of an object has permitted you to perform the action.

In this chapter, you learn how to create users in the Oracle database and then how to enable those users to perform certain actions in the database, such as create or modify tables or other database objects. You also learn how you can grant and manage permissions on database objects to enable a user to insert, update, and delete data in another user's schema.

Users and Schemas

By default, Oracle creates two users when you create your database. They are called SYS (with a password of "change_on_install") and SYSTEM (with a password of "manager"). The SYS user owns the data dictionary and all of its associated objects. The user SYSTEM has access to all objects in the database. The distinction between a user owning objects and a user having access only to objects that are owned by another user is an important one in Oracle.

Any user that has been given the permission to create objects and does so is said to *own a schema*. The schema is a collection of all objects that are owned by a user. The schema has the same name as the user.

For example, if Bob creates a table called Orders, at that point, Bob also creates his schema. The schema has the same name as the owner of the Orders table, namely Bob. Bob is said to own Bob's schema and is therefore a *schema owner*. Bob is also sometimes referred to as a *schema user* because he owns objects.

If Susan wants to be able to read data in Bob's Orders table, she must be granted permission by Bob to do so. Susan does not own any objects of her own, and consequently, she does not have a schema. She reads data in other schemas, such as Bob's. For this reason, Susan is also referred to as a *nonschema user*.

If Susan wants to query the contents of the Orders table that Bob created, she has to begin the name of the table with the name of the schema (that is, the name of the owner), in the form of Bob.Orders. In this way it is possible for her to distinguish an Orders table that Bob created from an Orders table created by another user. The full name of the table (Bob.Orders) is called the *fully qualified name* of the table.

The reason for using the fully qualified name instead of just the table name is that Susan may have access to a table called Orders in another schema. If this is so, Oracle returns an error indicating that reference to the Orders table is ambiguous, and Oracle is unable to determine which of the Orders tables that Susan has access to should be used to satisfy the query. Whenever creating multiple schemas by allowing more than one user to create database objects, it is important to use schema names in references to Oracle objects to avoid ambiguity.

Taking the concept of schema users and nonschema users to a real-world example, it is similar to you still living with your parents. Your parents own the house that you live in, just like an Oracle user owns a schema. Your parents allow you to live in the house (with or without cost), which makes you the nonschema user because you do not own the house. In fact, they have given you permission to access objects in the house (your room, the bathroom, the kitchen, and — most important — the fridge), which is similar to a schema owner granting privileges to other Oracle users to access a table or other object. The schema owner (that is, your parents), always has the option to revoke any privileges granted to objects in the database, just like parents have the option to kick you out of the house (but they won't because you're a good person).

Creating and managing users

- ✦ Create users

Gaining access to an Oracle database requires that you have a user account in the database. For this to happen, the person who controls the database has to create an Oracle user account for you. That person is the Oracle DBA.

The DBA is automatically granted permissions to do anything he or she wants in the database. This is because the DBA is the one that creates the database in the first place and, therefore, inherits all permissions in it. The Oracle user that the DBA corresponds to is called SYS. Another Oracle user that has been granted full DBA privileges is SYSTEM. As mentioned earlier, these two Oracle user accounts are created automatically by Oracle when the database is created. The password for the SYS user is always "change_on_install" at database creation time, and the password for the SYSTEM user is always "manager" at creation time. These passwords should be changed to prevent users with some Oracle knowledge from gaining access to the database.

Changing user passwords is covered in the section "Using the ALTER USER Command to Change Passwords," later in this chapter.

Connecting to an Oracle Instance

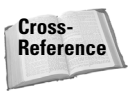
In order to connect to an Oracle instance, you need to provide a username and password, as well as the name of the instance to connect to. Most tools, including SQL*Plus, prompt you for this information and hide the password by echoing asterisks instead of the password characters. Another syntax that you can use in SQL*Plus, or any command line utility, is *username/password@instance*. If you use this syntax, it is important to remember that the password must be specified unencrypted. However, if you omit the password, you will be prompted to enter it, at which time no characters will be echoed and the password will not be visible to prying eyes. Although using the *username/password@instance* combination when executing the CONNECT command can be useful in having batch job runs successfully at off-peak hours, it is not recommended as it introduces a big security hole.

After the database has been created, the DBA connects to the instance as either SYS or SYSTEM (typically SYSTEM) and creates additional users to become schema owners, or to grant them some or all of the privileges that the DBA has inherited. Only the DBA can create users by default.

The CREATE USER command

The command issued by the DBA to create a user is the CREATE USER Oracle command. The simplified syntax for the CREATE USER command is as follows:

```
CREATE USER username
   IDENTIFIED BY password;
```



The syntax of the CREATE USER command outlined here is a subset of the complete syntax allowed by Oracle. For a description of the complete syntax, refer to the *Oracle 8i SQL Reference* in the Oracle documentation set.

For example, to create a user called Bob with a password of "mypass", you issue the following command:

```
SQL> CREATE USER Bob
  2  IDENTIFIED BY mypass;

User created.

SQL>
```

It is important to note that usernames and passwords in Oracle are not case sensitive. When attempting to connect to an Oracle database instance, whether you specify the username/password combination as "Bob/mypass" or "BOB/MYPASS" does not matter to Oracle because it ensures only that the right combination of characters are sent across, not whether they are uppercase or lowercase.

After creating a user, if you try to connect to the instance using the user account created, you get the following error:

```
SQL> connect bob/mypass@orcl.delphi.bradsys.com
ERROR:
ORA-01045: user BOB lacks CREATE SESSION privilege; logon denied

Warning: You are no longer connected to ORACLE.
SQL>
```

This error is returned because the act of creating a user does not mean that the user can actually access a database. In order to do so, the user must be granted a system privilege called CREATE SESSION. System privileges are discussed later in this chapter in the "System Privileges" section.

The DROP USER command

If a DBA determines that a user should no longer have access to the database, he or she can remove the user from the database using the DROP USER command. The syntax for the command is as follows:

```
DROP USER username [CASCADE];
```

For example, if you were a DBA, you issue the following command to remove Susan from the database:

```
SQL> DROP USER Susan;

User Dropped.

SQL>
```

If Susan owned objects in the database, you receive an error as follows:

```
SQL> DROP USER Susan;
DROP USER Susan
*
ERROR at line 1:
ORA-01922: CASCADE must be specified to drop 'SUSAN'

SQL>
```

The reason for the error is that Oracle does not allow a user who owns database objects to be removed because this orphans the objects. This is because the user's schema cannot be separated from the user. If you want to drop Susan and all of the objects she owns, you can, as the message indicates, specify the CASCADE option.

You should completely back up the database before specifying the CASCADE option while issuing the DROP USER command. Oracle does not have a mechanism to undo the DROP USER command because it is a Data Definition Language (DDL) command. Any DDL command, including DROP USER, automatically commits its work to the database. To reverse the action of the command, you must restore a backup copy of your database.

The "Introduction to Oracle: SQL & PL/SQL" exam does not test your knowledge of how to remove users from the database. This is because a normal user, even one with a schema, is not normally allowed to create other users. This task is one that DBAs keep for themselves.

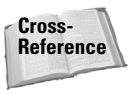
Using the ALTER USER command to change passwords

It is likely that after a user is created, the user will need to change his or her password to comply with the organization's security policy, or a DBA may need to change a user's password in case it has been forgotten. The command to accomplish this is the ALTER USER command.

Oracle 8*i* provides many account management features that can be used to enforce a company's security policy with regard to user passwords. Included is the ability to specify a maximum time that a user can be idle before being automatically disconnected, the maximum connect time, and the number of invalid logon attempts before the account is locked out. A discussion of these features is beyond the scope of this book. For more information, refer to profiles in the *Oracle 8i Administrators Guide* in the Oracle documentation set.

To change a user's password, you issue the following command:

```
ALTER USER username
   IDENTIFIED BY password;
```



The syntax of the ALTER USER command outlined here is a subset of the complete syntax allowed by Oracle. For a description of the complete syntax, refer to the *Oracle 8i SQL Reference* in the Oracle documentation set.

In order to change another user's password, you must be the DBA, or have been granted the ALTER USER system privilege. Any user can change his or her own password using the preceding syntax.

ALTER USER and other system privileges are covered later in this chapter in the "System Privileges" section.

For example, if Bob is logged into the database as the user Bob, to change his password from "mypass", which was specified when the user was created, to "newpass", Bob issues the following command:

```
SQL> ALTER USER Bob
  2  IDENTIFIED BY newpass;

User altered.

SQL>
```

If the DBA wanted to change Bob's password, the command that the DBA would issue is the same.

Granting and Administering User Privileges

After you have created user accounts in Oracle, you need to enable those users to perform certain actions in the database or to access and manipulate objects in the database. This is accomplished through the granting and revoking of different privileges (or permissions).

Oracle has two different types of privileges that can be granted — system privileges and object privileges. System privileges enable a user to perform certain database actions, such as create a table or an index or even connect to the instance. Object privileges enable a user to manipulate objects, such as read data through a view, execute a stored procedure, or change data in a table. Generally, system privileges are granted to and revoked from users by the DBA, while object privileges are granted to and revoked from users by the owner of the object.

System privileges

Oracle 8*i* includes over 100 system privileges that can be granted to users. They include the capability to create various database objects and modify the configuration of the database. The granting of these privileges is restricted to the DBA by default, but it is possible to enable users who have been granted a privilege to grant it to others as well. Some of the most commonly granted system privileges are listed in Table 8-1. For a complete listing of all system privileges, refer to the *Oracle 8i Administrators Guide*, a part of the Oracle 8i documentation set.

Table 8-1
Common System Privileges

Privilege	*Description*
CREATE SESSION	Enables a user to connect to the database instance. After creating a new user, you need to grant the CREATE SESSION privilege; otherwise, the user cannot connect. Granting the CREATE SESSION privilege is not done automatically by Oracle and must be granted explicitly.
CREATE TABLE	Enables a user to create a table in his or her schema.
CREATE VIEW	Enables a user to create a view in his or her schema.
CREATE SYNONYM	Enables a user to create a private synonym in his or her schema.
CREATE PUBLIC SYNONYM	Enables a user to create a synonym in the SYS schema that can be used by any user in the database.
CREATE PROCEDURE	Enables a user to create a stored procedure or function is his or her schema.
CREATE SEQUENCE	Enables a user to create a sequence in his or her schema.
CREATE TRIGGER	Enables a user to create a trigger in his or her schema on a table in his or her schema.
CREATE USER	Enables a user to create another user in the database and specify the password and other settings at creation time.
ALTER USER	Enables a user who has been granted the privilege to modify the user information of another user in the database, including changing the user's password.
DROP ANY TABLE	Enables a user to drop any table in any schema in the database.
ALTER ANY TABLE	Enables a user to alter any table in any schema in the database.
BACKUP ANY TABLE	Enables a user to make a copy of any table in the database using the Export utility (exp).
SELECT ANY TABLE	Enables a user to issue a SELECT statement against any table in the database, whether or not permissions to the table have been explicitly granted to the user performing the action.
INSERT ANY TABLE	Enables a user to issue an INSERT statement against any table in the database, whether or not permissions to the table have been explicitly granted to the user performing the action.

Continued

Table 8-1 *(continued)*

Privilege	*Description*
UPDATE ANY TABLE	Enables a user to issue an UPDATE statement against any table in the database, whether or not permissions to the table have been explicitly granted to the user performing the action.
DELETE ANY TABLE	Enables a user to issue a DELETE statement against any table in the database, whether or not permissions to the table have been explicitly granted to the user performing the action.

Granting system privileges

In order to assign system privileges to users, you must be connected to the instance as a user who is a DBA, or as a user who has been given permissions to assign the system privilege to others. The syntax for assigning system privileges is as follows:

```
GRANT privilege [, privilege, ...]
TO username [, username, ...]
[WITH ADMIN OPTION];
```

As you can see by this syntax, it is possible to grant multiple privileges to multiple users at the same time. The privileges granted to a user are immediately available to the user. This means that the user does not have to disconnect from the instance and log in again in order for the privilege change to take effect. Simply granting the privilege enables the user to make use of it right away.

For example, if you as the DBA want to grant the CREATE SESSION privilege to Bob and Susan, you issue the following command:

```
SQL> GRANT CREATE SESSION
  2  TO Bob, Susan;

Grant succeeded.

SQL>
```

If you want to enable Bob to create tables, views, triggers, indexes, synonyms, and sequences in his schema, you can issue the following command:

```
SQL> GRANT CREATE TABLE, CREATE VIEW, CREATE SYNONYM,
  2  CREATE SEQUENCE, CREATE TRIGGER, CREATE INDEX
  3  TO Bob;
```

```
Grant succeeded.

SQL>
```

Just because you have been granted a system privilege does not always mean that you can perform the action that you have been granted permissions to. For example, if Bob connects to the instance and attempts to create a table, the following results:

```
SQL> connect Bob/newpass@orcl.delphi.bradsys.com
Connected.
SQL> CREATE TABLE BobTable
  2  (BobID number,
  3  Name varchar2(40));
CREATE TABLE BobTable
*
ERROR at line 1:
ORA-01950: no privileges on tablespace 'USERDATA'

SQL>
```

The error message that Oracle returns indicates that Bob does not have privileges on tablespace "USERDATA". A tablespace is a logical unit of storage that is made up of one or more operating system files. Tables, indexes, and any other Oracle object that requires storage are located on tablespaces.

In order for users to be able to create tables and indexes, they must also be granted a quota on the tablespace that will be used to store the data. When Bob tried to create the table BobTable, Oracle checked to see if Bob had been granted a quota on the tablespace where the table will be stored (specified by the DBA). When no quota was found, the command returned the error shown previously.

The "Introduction to Oracle: SQL & PL/SQL" exam does not test your knowledge of how to assign quotas to users or how to define a default tablespace for a user.

The DBA typically assigns a default and temporary tablespace to the user, and assigns the user a quota on the tablespace. The quota specifies the maximum amount of disk space that a user's object may take up on the tablespace and can be set to unlimited if needed (although it should not be). For more information on the management of user quotas or tablespaces, refer to the *Oracle 8i Administrators Companion* in the Oracle 8*i* documentation set.

The WITH ADMIN OPTION privilege

When a user is granted system privileges, the grantor (that is, the person granting the privilege — typically the DBA) also has the option to enable the grantee (the person receiving the privilege, typically the user) to grant the same privilege to other users. If this is the result desired, the grantor can grant the privilege using the WITH ADMIN OPTION.

If the DBA wants to enable Bob, who is the development manager, to grant the privileges to create tables, indexes, and other database objects to other users, the DBA grants Bob the privilege WITH ADMIN OPTION as shown here:

```
SQL> GRANT CREATE TABLE, CREATE VIEW, CREATE SYNONYM,
  2  CREATE SEQUENCE, CREATE TRIGGER, CREATE PROCEDURE
  3  TO Bob WITH ADMIN OPTION;

Grant succeeded.

SQL>
```

In turn, if Bob then wants to grant Susan the privilege to create tables, sequences, and synonyms, Bob can issue the following command:

```
SQL> connect Bob/newpass@orcl.delphi.bradsys.com
Connected.
SQL> GRANT CREATE TABLE, CREATE SEQUENCE, CREATE SYNONYM
  2  TO Susan;

Grant succeeded.

SQL>
```

If Bob had not been granted the privileges he then granted to Susan, the following error would be presented:

```
SQL> GRANT CREATE TABLE, CREATE SEQUENCE, CREATE SYNONYM
  2  TO Susan;
GRANT CREATE TABLE, CREATE SEQUENCE, CREATE SYNONYM
*
ERROR at line 1:
ORA-01031: insufficient privileges


SQL>
```

Granting system privileges WITH ADMIN OPTION is not recommended in practice. By allowing others to grant the privileges they have received, you may lose control over certain aspects of managing the database. In real-world situations, few DBAs grant system privileges to other Oracle users WITH ADMIN OPTION.

Revoking system privileges

If you do not want anyone to continue to have a system privilege granted to them, you can use the REVOKE command to remove the privileges granted. The syntax to revoke system privileges is very similar to that of granting them, as follows:

```
REVOKE privilege [, privilege, ...]
FROM username [, username, ...];
```

As with the GRANT command, the REVOKE command also accepts multiple privileges and/or users in the syntax. In order for the REVOKE command to be issued to remove system privileges from users, the person executing the command must be a DBA or have been granted the privilege being revoked WITH ADMIN OPTION.

For example, if the DBA no longer wanted Bob to be able to create stored procedures in the database, the DBA can issue the following command:

```
SQL> REVOKE CREATE PROCEDURE FROM Bob;

Revoke succeeded.

SQL>
```

After the command is processed, Bob no longer is able to issue the CREATE PROCEDURE command. Furthermore, if Bob were granted the privilege to create procedures in the database WITH ADMIN OPTION, Bob also would not be able to grant the privilege to others (because he does not have it himself).

It is important to note one side effect that is the result of specifying WITH ADMIN OPTION at the time a system privilege is granted. While the DBA may revoke the privilege granted to the user WITH ADMIN OPTION, if the user (in this case, Bob) granted that privilege to others, it is not removed from those users that were granted the privilege.

For example, if you give the key to your apartment to a friend and tell him that he can make copies of the key, when you ask for the key back from your friend, you cannot, at the same time, get back all copies that were made by him and given to others. In order to retrieve the other copies of the key, you must find out who has them. Similarly, in Oracle, you must query the data dictionary to determine which other users were granted the permission being revoked by the user from which it is being revoked.

Determining system privileges granted

If you want to find out which system privileges you have been granted, Oracle provides the USER_SYS_PRIVS view. The view enables you to see which privileges you have been granted and whether or not they have been granted WITH ADMIN OPTION. For example, if Bob queried the USER_SYS_PRIVS views, the output might be similar to the following:

```
SQL> SELECT * FROM USER_SYS_PRIVS;

USERNAME                       PRIVILEGE                                ADM
------------------------------ ---------------------------------------- ---
BOB                            CREATE SEQUENCE                          YES
BOB                            CREATE SESSION                           NO
BOB                            CREATE SYNONYM                           YES
BOB                            CREATE TABLE                             YES
BOB                            CREATE TRIGGER                           YES
BOB                            CREATE VIEW                              YES
```

```
6 rows selected.

SQL>
```

Only those system privileges that have been granted appear on the list. Any privileges that have been revoked are not listed because Oracle does not keep track of permissions denied a user. The default for Oracle is to deny all actions unless those explicitly granted; therefore, only those explicitly granted are listed.

Object privileges

Objective

- Use the GRANT and REVOKE statements to grant and revoke object privileges

The second type of privileges that can be granted to a user in Oracle are object privileges. Object privileges enable a user to manipulate data in the database or perform an action on an object, such as execute a stored procedure. Unlike system privileges, which are granted by the DBA, object privileges must be granted by the owner of the object.

The syntax to assign object privileges is as follows:

```
GRANT privilege [,privilege, ...] | ALL [(column[, column,
...])]
   ON objectname
   TO user | role | PUBLIC
   [WITH GRANT OPTION];
```

The major difference in the syntax between system and object privileges is that the keyword ON must be specified to determine which object the privileges apply to. Furthermore, object privileges for views and tables can also specify which column of the view or table they should be applied to. The keyword ALL specifies that all privileges that apply to the object should be granted. The privilege can be granted to a user, role (to be discussed later), or the keyword PUBLIC, which means all users in the database.

For example, to enable all users in the database to query the Courses table, the owner of the table (Student) issues the following command:

```
SQL> GRANT SELECT ON Courses TO PUBLIC;

Grant succeeded.

SQL>
```

At this point, any user that exists in the database is able to query the Student.Courses table.

The types of privileges that can be granted depend on the object on which they are being granted. For example, it makes no sense to grant the SELECT privilege to a stored procedure, while the SELECT privilege makes perfect sense on a table. The object privileges that can be granted and the object they can be granted to are outlined in Table 8-2. The options listed in the table make sense. For example, you cannot issue an ALTER VIEW command, so therefore, the ALTER privilege cannot apply to a view.

Table 8-2
Object Privilege Application

Privilege	*Granted On*
SELECT	Table, view, sequence
INSERT	Table, view
UPDATE	Table, view
DELETE	Table, view
ALTER	Table, sequence
INDEX	Table
REFERENCES	Table
EXECUTE	Procedure, function, package

Most of the privileges shown in Table 8-2 are self-explanatory, with the possible exception of the REFERENCES privilege. The REFERENCES privilege can be granted to a user to create a FOREIGN KEY constraint on a column or columns of a table, or to create a view on the table. By granting users the REFERENCES privilege, you do not have to enable the users to see that data, as they would with the SELECT privilege, but are allowing them only to reference the data in the FOREIGN KEY or the view. The SELECT permission alone is not sufficient to create a FOREIGN KEY or view that references a column in the table; the REFERENCES permission is also required. Even when the user has the SELECT permission on the table, the creation of a FOREIGN KEY or view fails without the REFERENCES permission.

One of the options available when granting the INSERT, UPDATE, and REFERENCES privileges on a table or view is to restrict the columns available for modifying or referencing in the table. This is done by providing a list of columns in the GRANT statement, as shown here:

```
SQL> GRANT UPDATE (CourseName)
  2  ON Courses
  3  TO Susan;
```

```
Grant succeeded.

SQL>
```

This enables Susan to update the CourseName column of the Courses table. Susan cannot update any other columns in the table.

Granting object privileges at the column level is generally not recommended, and often frowned upon. The management of many column-level privileges can become quite time consuming, as well as confusing. When you need to assign privileges to only certain columns of a table, it is generally recommended that you create a view including only those columns and grant the appropriate permission on the view itself. This way, if you drop the view, or remove permission from the view for a user, the management is easier and cleaner.

The WITH GRANT OPTION privilege

Similar to the WITH ADMIN OPTION system privilege, the WITH GRANT OPTION object privilege enables a user granted the privilege to grant it to someone else. The reason for doing this is to minimize the administrative burden of granting object privileges. To grant the privilege to query class enrollment information to Bob and also to enable Bob to grant this privilege to other users, you issue the following command:

```
SQL> GRANT SELECT ON ClassEnrollment TO Bob
  2  WITH GRANT OPTION;

Grant succeeded.

SQL>
```

If Bob needs to grant this same privilege to Susan, he issues the following command:

```
SQL> connect Bob/newpass@orcl.delphi.bradsys.com
Connected.
SQL> GRANT SELECT ON Student.ClassEnrollment TO Susan
  2  WITH GRANT OPTION;

Grant succeeded.

SQL>
```

Note that Bob had to begin the name of the ClassEnrollment table with the owner name; otherwise, he receives the following message:

```
SQL> GRANT SELECT ON ClassEnrollment TO Susan
  2  WITH GRANT OPTION;
GRANT SELECT ON ClassEnrollment TO Susan
                *
```

```
ERROR at line 1:
ORA-00942: table or view does not exist


SQL>
```

Notice also that Bob is able to grant Susan SELECT permissions on the Student.ClassEnrollment table. This is by design and is permitted so that Bob can further lighten his workload by enabling Susan to grant access to the Student.ClassEnrollment table to others.

Determining the object privileges granted

As is the case with system privileges, Oracle enables a user to determine which object privileges have been granted to him or her by querying the data dictionary. Table 8-3 identifies the various data dictionary views and the information available through them.

Table 8-3
Object Privilege Data Dictionary Views

View	*Description*
USER_TAB_PRIVS_MADE	Lists the object privileges granted to others on objects in the user's schema. This includes privileges granted by the user to others and granted by users that have been assigned the privilege WITH GRANT OPTION.
USER_TAB_PRIVS_RECD	Lists the object privileges that were granted to the user on objects in other schemas.
USER_COL_PRIVS_MADE	Lists the object privileges made on columns of objects owned by the user.
USER_COL_PRIVS_RECD	Lists the object privileges on columns of objects in other schemas granted to the user.

To view the list of privileges that you have granted to others on tables in your schema, you issue the following command:

```
SQL> col grantee format a10
SQL> col grantor format a10
SQL> col table_name format a20
SQL> col privilege format a20
SQL> SELECT * FROM USER_TAB_PRIVS_MADE;
```

```
GRANTEE    TABLE_NAME           GRANTOR    PRIVILEGE            GRA
---------- -------------------- ---------- -------------------- ---
BOB        CLASSENROLLMENT      STUDENT    SELECT               YES
SUSAN      CLASSENROLLMENT      BOB        SELECT               YES
PUBLIC     COURSES              STUDENT    SELECT               NO

SQL>
```

Notice that the fact that Bob granted Susan the SELECT privilege on the ClassEnrollment table is clearly indicated in the output presented. This enables the owner of an object to quickly determine who has been granting permissions to other users on his or her objects.

Revoking object privileges

Revoking object privileges has similar syntax to granting them. The full syntax is:

```
REVOKE privilege [,privilege, ...] | ALL [(column[, column,
...])]
   ON objectname
   FROM user | role | PUBLIC
   [CASCADE CONSTRAINTS];
```

To revoke the SELECT privilege grant to Bob on the ClassEnrollment table, the table owner executes the following command:

```
SQL> REVOKE SELECT ON ClassEnrollment FROM Bob;

Revoke succeeded.

SQL>
```

Querying the USER_TAB_PRIVS_MADE data dictionary view after executing the preceding command results in the following:

```
SQL> SELECT * FROM USER_TAB_PRIVS_MADE;

GRANTEE    TABLE_NAME           GRANTOR    PRIVILEGE            GRA
---------- -------------------- ---------- -------------------- ---
PUBLIC     COURSES              STUDENT    SELECT               NO

SQL>
```

If you compare this output with the results of the same query on the previous page, you note that the SELECT privilege has been removed on the ClassEnrollment table from both Bob *and* Susan. This is because any object privileges that were granted to a user using WITH GRANT OPTION are revoked from that user and any other user (Susan) that he (Bob) granted them to. Contrast this to the behavior of revoking system privileges granted to a user WITH ADMIN OPTION, where the cascading delete of permissions does not take place.

An option of the REVOKE command for object privileges is CASCADE CONSTRAINTS. This option is required in those situations where a user has been granted REFERENCES permission on a table in your schema, and he or she has used this privilege to create a table with a FOREIGN KEY constraint depending upon the table you own. Attempting to revoke the REFERENCES privilege generates an error, as shown here:

```
SQL> REVOKE REFERENCES ON Courses FROM Bob;
REVOKE REFERENCES ON Courses FROM Bob
*
ERROR at line 1:
ORA-01981: CASCADE CONSTRAINTS must be specified to perform this revoke

SQL>
```

To correct the problem, simply reissue the command using the CASCADE CONSTRAINTS option, as follows:

```
SQL> REVOKE REFERENCES ON Courses FROM Bob CASCADE CONSTRAINTS;

Revoke succeeded.

SQL>
```

At this point, the permission is revoked, and the FOREIGN KEY that Bob created that referenced the Courses table in the Student schema is also dropped.

When using the CASCADE CONSTRAINTS option of the REVOKE command for object privileges, verify that any FOREIGN KEY created in another schema referencing the table is no longer required. One of the things you do not want to do is break the database and invite inconsistent data by simply revoking an object privilege. If you do get the ORA-01981 error, query the data dictionary to determine which objects are referencing the table and notify those table's owners of your intentions prior to performing the action.

Roles

- ✦ Create roles to ease setup and maintenance of the security model

Up to this point, you have learned how to assign system and object privileges to users or, in the case of object privileges, PUBLIC. In small environments, assigning permissions directly to users may be sufficient, especially if new users must be created in the database only occasionally. However, as things can always change and to support volatile and large environments, Oracle provides a mechanism to group permissions together and then assign the whole group of permissions to a user. This mechanism is called a *role*.

A role is a container that holds privileges. Just as a soft drink can is a container for a tasty and refreshing beverage, a role holds privileges. Many individuals at the same time can enjoy a soft drink, just as many users in Oracle can be granted the role. A user may hold many roles at the same time, just as a user can drink many different soft drinks.

The main benefit of a role is that it simplifies the process of granting privileges to users. To make the process efficient, a DBA creates a role and then grants all of the privileges required by a user to perform a task to the role. If another user comes along that needs to perform the same task, instead of granting that user the permission explicitly, the DBA grants the user the role. Any privileges that have been granted to a role that has been granted to a user automatically apply to the user. Furthermore, it is possible to grant a role to another role, in which case the second role will inherit all of the privileges of the first. However, granting roles to other roles can make it difficult to track down problems with permissions and should be carefully planned.

Creating and granting roles

Roles are created by the DBA using the CREATE ROLE command, as shown here:

```
SQL> CREATE ROLE OrderEntry;

Role created.

SQL>
```

After the role has been created, the DBA, or the owner of a database object, can assign permissions to a role, as follows:

```
SQL> GRANT CREATE SESSION TO OrderEntry;

Grant succeeded.

SQL> connect student/oracle@orcl.delphi.bradsys.com;
Connected.
SQL> GRANT SELECT ON Student.Courses TO OrderEntry;

Grant succeeded.

SQL> GRANT SELECT, INSERT, UPDATE
  2  ON Student.ClassEnrollment TO OrderEntry;

Grant succeeded.

SQL>
```

As shown by the preceding output, both system privileges and object privileges can be granted to a role. The full syntax for assigning privileges to a role is:

```
GRANT privilege | role [, privilege | role, ...]
TO rolename;
```

The next step is to grant the role to a user, at which point the user inherits all permissions granted the role. In the following example, Bob attempts to query the ClassEnrollment table before and after being granted the role OrderEntry by the DBA:

```
SQL> connect bob/newpass@orcl.delphi.bradsys.com;
Connected.
SQL> SELECT * FROM Student.ClassEnrollment;
SELECT * FROM Student.ClassEnrollment
                      *
ERROR at line 1:
ORA-00942: table or view does not exist


SQL> connect system/manager@orcl.delphi.bradsys.com;
Connected.
SQL> GRANT OrderEntry TO Bob;

Grant succeeded.

SQL> connect bob/newpass@orcl.delphi.bradsys.com;
Connected.
SQL> SELECT ClassID, StudentNumber, Status, TO_CHAR(Price,'$99,999.99') AS Price
  2  FROM Student.ClassEnrollment;

   CLASSID STUDENTNUMBER STATUS     PRICE
---------- ------------- ---------- -----------
        50          1001 Confirmed    $2,000.00
        50          1002 Confirmed    $1,750.00
        50          1005 Confirmed    $2,000.00
        51          1003 Cancelled    $4,000.00
        51          1004 Confirmed    $4,000.00
        51          1008 Confirmed    $3,500.00
        53          1003 Hold         $1,500.00

7 rows selected.

SQL>
```

You should note that it is possible to grant a role to another role. In this way, the assignment of permissions can be broken down further. You can first create a role to accomplish a very specific task, such as to enter or update an order. You then create another role to correspond to the individual who will perform one or more tasks, such as an order-entry clerk. You then assign each task-specific role to the

user-specific role. In this way, when another task must be added to the list of tasks a user has to perform, another task-specific role can be created and then granted to the user-specific role. Similarly, when a task has to be removed from the job description for a group of individuals, you simply revoke the task-specific role from the user-specific role.

While the "Introduction to Oracle: SQL & PL/SQL" exam may not test whether or not you are aware that roles can be granted to other roles, it is a good idea to keep this bit of information in mind when studying for the exam. You may be tested on this knowledge in an indirect manner.

Determining privileges and roles granted

After privileges have been granted to a role, and after that role has been granted to a user, the user can query the data dictionary to determine which roles have been granted and their associated permissions. The data dictionary views providing this information are outlined in Table 8-4.

Table 8-4
Data Dictionary Views Describing Roles

View	*Description*
ROLE_SYS_PRIVS	Lists the system privileges that have been assigned to roles and available to the currently logged in user.
ROLE_TAB_PRIVS	Lists the object privileges granted to the role on tables in the database. All of these privileges are available to the current user.
USER_ROLE_PRIVS	Lists the roles that have been granted to the user.

While connected to the instance as Bob and you want to see which roles are available, you issue the following command:

```
SQL> SELECT GRANTED_ROLE FROM USER_ROLE_PRIVS;

GRANTED_ROLE
------------------------------
ORDERENTRY

SQL>
```

To determine which system privileges have been granted to the roles that are currently active for Bob, he can issue the following command:

```
SQL> SELECT * FROM ROLE_SYS_PRIVS;

ROLE                           PRIVILEGE            ADM
------------------------------ -------------------- ---
ORDERENTRY                     CREATE SESSION       NO

SQL>
```

To find out which privileges he has been granted on objects in the database, Bob can query the ROLE_TAB_PRIVS view, as shown here:

```
SQL> col role format a15
SQL> col owner format a15
SQL> col table_name format a20
SQL> col privilege format a10
SQL> SELECT ROLE, OWNER, TABLE_NAME, PRIVILEGE
  2  FROM ROLE_TAB_PRIVS;

ROLE            OWNER           TABLE_NAME           PRIVILEGE
--------------- --------------- -------------------- ----------
ORDERENTRY      STUDENT         CLASSENROLLMENT      INSERT
ORDERENTRY      STUDENT         CLASSENROLLMENT      SELECT
ORDERENTRY      STUDENT         CLASSENROLLMENT      UPDATE
ORDERENTRY      STUDENT         COURSES              SELECT

SQL>
```

Revoking roles

The syntax to revoke a role granted to a user is similar to what you have seen when revoking system and object privileges, as shown here:

```
REVOKE ROLE rolename FROM user | role;
```

Only the DBA is allowed to revoke roles from users and other roles.

For example, to revoke the OrderEntry role from Bob, the DBA issues the following command:

```
SQL> REVOKE OrderEntry FROM Bob;

Revoke succeeded.

SQL>
```

At this point, if Bob queried the data dictionary to determine what object privileges were available through roles granted him, he would get the following result:

```
SQL> connect bob/newpass@orcl.delphi.bradsys.com;
Connected.
SQL> col role format a15
SQL> col owner format a15
SQL> col table_name format a20
SQL> col privilege format a10
SQL> SELECT ROLE, OWNER, TABLE_NAME, PRIVILEGE
  2  FROM ROLE_TAB_PRIVS;

no rows selected

SQL>
```

Key Point Summary

Security is a very important element of ensuring that your data remains safe and can only be seen or changed by those individuals authorized to do so. The bullet points that follow summarize the salient points presented in this chapter.

- ✦ Each individual that must create or access database objects must be provided with an Oracle username and password.
- ✦ The DBA is the only one who can create a user, by default. The DBA can grant the CREATE USER privilege to others, which can be used to create more Oracle database users. This is not normally done.
- ✦ Any user can change his or her password by using the ALTER USER command.
- ✦ A user must be granted the CREATE SESSION privilege in order for a connection to the instance to be established.
- ✦ Oracle has two types of privileges: system privileges and object privileges.
- ✦ System privileges enable a user to execute a particular command, such as CREATE TABLE or CREATE VIEW. They are granted to users so that the administrative burden on the DBA is lessened.
- ✦ System privileges can be granted WITH ADMIN OPTION, which enables the person granted a system privilege in this manner to grant it to others.
- ✦ When a system privilege granted WITH ADMIN OPTION is revoked, the revoke does not cascade to all users granted the privilege through the option.
- ✦ Object privileges enable a user to manipulate objects, such as SELECT data from a table, sequence, or view, or EXECUTE a stored procedure.

- Object privileges can be granted WITH GRANT OPTION, which enables the grantee to grant the privilege to others.
- Revoking object privileges granted WITH GRANT OPTION will cascade and also will be revoked from others they have been granted to.
- A DBA can create roles and grant system privileges, object privileges, or other roles to the roles created.
- Roles simplify the administration of privileges by reducing the assignment of the same privileges to multiple users to an assignment of the role only, with all permissions granted the role being inherited by all users granted the role.

✦ ✦ ✦

STUDY GUIDE

This section will enhance your understanding of the material presented in this chapter. Answer the questions and then work through the labs in order to feel more comfortable with the material.

Assessment Questions

1. When attempting to revoke a user's privilege on your Orders table, you receive an error. What is the most likely cause of the error? (Choose the best answer.)
   - **A.** Your Orders table contains data.
   - **B.** Another user has a primary key defined on your Orders table.
   - **C.** You have defined a FOREIGN KEY constraint on your Orders table that references another user's table's PRIMARY KEY constraint.
   - **D.** Another user has defined a FOREIGN KEY constraint that references the PRIMARY KEY constraint of your Orders table.
   - **E.** Users are making changes to the data in the table.
2. You have been granted the CREATE TABLE privilege WITH ADMIN OPTION. Which of the following can you perform? (Choose all correct answers.)
   - **A.** Grant the privilege to other users.
   - **B.** Revoke the privilege from the DBA.
   - **C.** Revoke the privilege from other users.
   - **D.** Revoke the privilege from yourself.
   - **E.** Grant the privilege to other users WITH ADMIN OPTION.
3. What users exist in an Oracle database after it has been created?
   - **A.** DBA
   - **B.** SYS
   - **C.** BOB
   - **D.** SCOTT
   - **E.** SYSTEM

4. Which of the following cannot be granted to a role? (Choose all correct answers.)

A. System privileges

B. System privileges WITH ADMIN OPTION

C. Roles

D. Roles WITH GRANT OPTION

E. Roles WITH ADMIN OPTION

F. Object privileges

G. Object privileges with GRANT OPTION

5. Which of the following clauses of the CREATE USER command specify a password for the user? (Choose the best answer.)

A. CREATE USER

B. PASSWORD

C. ROLE

D. IDENTIFIED BY

E. Users do not have password in Oracle.

6. In the following grant of system privileges, which line will cause the command to fail? (Choose the best answer.)

```
1    GRANT CREATE TABLE, CREATE VIEW,
2    CREATE PROCEDURE, CREATE SEQUENCE
3    TO SCOTT, BOB, DeveloperRole
4    SUSAN;
```

A. 1

B. 2

C. 3

D. 4

E. None of the above

7. You have been granted the CREATE VIEW privilege by the DBA WITH ADMIN OPTION. You grant the privilege to Susan. When the DBA revokes the CREATE VIEW privilege from you, which of the following is true? (Choose all correct answers.)

A. You no longer can create views.

B. Susan no longer can create views.

C. You are able to create views.

D. Susan is able to create views.

E. You are able to grant the CREATE VIEW privilege to others.

F. You are not able to grant the CREATE VIEW privilege to others.

8. If you want to see what users Bob granted the SELECT privilege on your Courses table to, which data dictionary view do you query? (Choose the best answer?)

A. USER_TAB_PRIVS

B. USER_TAB_PRIVS_MADE

C. USER_TAB_PRIVS_RECD

D. ROLE_TAB_PRIVS

E. USER_ROLE_PRIVS

9. If you want to completely remove all privileges on your ClassEnrollment table from Susan, which of the following commands should you execute? (Choose the best answer.)

A. REVOKE ALL ON ClassEnrollment FROM Susan;

B. REVOKE PRIVILEGES ON ClassEnrollment FROM Susan;

C. REVOKE ALLPRIVS ON ClassEnrollment FROM Susan;

D. REVOKE ALLPRIVS ON ClassEnrollment FROM Susan CASCADE CONSTRAINTS;

E. REVOKE ALL ON ClassEnrollment FROM Susan CASCADE CONSTRAINTS;

F. REVOKE PRIVILEGES ON ClassEnrollment FROM Susan CASCADE CONSTRAINTS;

10. Which of the following combinations of commands is the best way to assign privileges to the FirstName and LastName columns of the Instructors table to Susan? (Choose two correct answers.)

A. GRANT ALL (FirstName, LastName) ON Instructors TO Susan;

B. CREATE VIEW InstructorNames AS SELECT FirstName, LastName FROM Instructors;

C. GRANT ALL ON InstructorNames TO Susan;

D. GRANT SELECT, INSERT, UPDATE, DELETE, REFERENCES (FirstName, LastName) ON Instructors TO Susan;

E. GRANT SELECT, INSERT, UPDATE, DELETE, REFERENCES ON Instructors (FirstName, LastName) TO Susan;

F. GRANT ALL ON Instructors (FirstName, LastName) TO Susan

Scenarios

1. You have just been hired as a consultant by a major consumer goods distributor to assist the distributor in making it easier to assign permissions to database objects. The problems the client has identified include the following:
   - Assignment of permissions for new order-entry clerks is time consuming because many tables and other database objects are involved.
   - Managers of departments want an easy mechanism to grant additional privileges to their employees without involving the DBA.
   - The DBAs in the company are spending way too much time dealing with the assignment or revocation of privileges.
   - Users' actions on the database can be logically organized into a series of tasks.

   What recommendations do you recommend to make it easier to assign and maintain permissions?
2. As the senior DBA of a multinational retailer of books, music, and video products, you are swamped with requests to create additional databases. You are spending a great deal of your time managing users and creating database objects at the request of users in any one of your company's locations. You now are typically one to two weeks behind in responding to requests, and management and users are complaining. What can you do to lessen your workload and carry out the creation of databases and database objects, and the management of users, in a timely manner?

Lab Exercises

Lab 8.1 Creating and Managing Users

1. Invoke SQL*Plus and connect to your database instance as the user SYSTEM with a password of manager. If you previously changed the password for the user SYSTEM, make sure you specify the new password when connecting.
2. Create three new users with usernames of Carol, Ted, and Alice. Make the password for all three users: oracle.
3. Attempt to connect to the instance as any one of the users. What happens and why?
4. Connect as SYSTEM and grant each of the users the appropriate privileges to connect to the instance. Connect to the instance as each of the users to ensure that the assignment worked correctly.

5. Connect to the instance as Alice and change your password to newpass. Disconnect and reconnect to the instance as Alice to test the password change.
6. Connect to the instance as SYSTEM and change Ted's password to password. Attempt to connect to the instance as Ted with both the old and new password. What happens when you try the old password?
7. Connect to the instance as SYSTEM and drop Alice's user account. Attempt to connect to the instance as Alice. What happens?

Lab 8.2 Creating and Managing Roles and Permissions

1. Connect to the instance as system and create two new roles called DeveloperRole and UserRole.
2. Grant the following privileges to the DeveloperRole:

```
CREATE TABLE
CREATE VIEW     WITH ADMIN OPTION
CREATE SEQUENCE WITH ADMIN OPTION
CREATE SYNONYM
```

3. Grant the DeveloperRole to Carol and the UserRole to Ted.
4. Connect as Ted and verify what privileges you have.
5. Connect to the instance as Student and grant the following privileges to the UserRole:

```
SELECT ON Student.Courses
SELECT, INSERT, UPDATE ON Student.ClassEnrollment
SELECT ON Student.ScheduledClasses
ALL ON Student.Students
```

6. Connect as Ted and verify what privileges you now have. What has changed?
7. While still connected as Ted, attempt to create a view that includes the StudentNumber, FirstName, and LastName columns of the Students table. What happens?
8. Connect to the instance as Carol and attempt to create the same view as in the previous step. What happens?
9. Connect to the instance as student and create the view mentioned in step 7 in such a way that users cannot change data through the view. Grant the UserRole all permissions on the view.
10. Connect to the instance as Ted and attempt to query the StudentNameView in the Student schema. What happens?
11. Connect to the instance as SYSTEM and grant the appropriate privileges to enable Carol to grant the DeveloperRole to other users.

12. Connect to the instance as Carol and grant the DeveloperRole to Ted.
13. Connect to the instance as Ted and attempt to create a synonym for the Student.Students table. What happens? Use the synonym in a query.
14. Connect to the instance as SYSTEM and revoke the DeveloperRole from Carol.
15. Connect to the instance as Ted and drop and then attempt to create the synonym from step 13. What happens and why?

Answers to Chapter Questions

Chapter Pre-Test

1. A schema user is one who owns objects in the database. As soon as a user has been given permission to create objects, and the user exercises that permission and creates an object, a schema with the same name as the user is defined in the database. Schema users own objects; nonschema users do not own objects but may be granted permissions to access the objects.
2. Yes, it is possible to limit which columns a user is able to modify in a table when granting the user permissions to update the table. This is done by specifying the columns that the user can update in the GRANT statement. A better way is to create a view with only those columns included and then grant the user permission to update the view.
3. A role is a container for permissions. You can assign any statement and/or object privileges that are required for a user to perform a specific task to the role, and then grant the role to the user. Any privileges that the role has the user will inherit. Furthermore, when you add new privileges to or revoke privileges from the role, all users granted the role also inherit the changes immediately.
4. Yes, it is possible to grant a role to a role. To avoid the problem of tracking permissions, you should limit the number of levels to one or two at most. In other words, don't grant a role that has been granted another role to a role.
5. After a user creates a table, only the user who created the table and the DBA can query it. The user is the owner and has full privileges on the table; the DBA has full privileges on all objects in the database.
6. When you grant permission on a view to a user WITH GRANT OPTION, this means that the user to whom the permission has been granted may also grant it to another user. When the permission is granted to a role, any user to whom the role has been granted may also grant the same permission to other users.
7. To determine which permissions have been granted to you on tables other than your own, you query the USER_TAB_PRIVS_RECD view.

8. When another user has referenced your table in a FOREIGN KEY constraint, you cannot remove the REFERENCES privilege on the table. In order to do so, you must specify the CASCADE CONSTRAINTS option on the REVOKE command.
9. The REVOKE command is used to remove system privileges from a table. By default, only the DBA is allowed to revoke system privileges. Users that have been granted a system privilege WITH ADMIN OPTION may revoke it from those other users to whom they granted the privilege.
10. The ALTER USER command is used to change your password.

Assessment Questions

1. **D** — The most likely reason that you are unable to revoke a user's privileges on your table is that he or she has referenced your table in a FOREIGN KEY constraint definition. Whether or not your Orders table contains data, or you create a FOREIGN KEYconstraint on the table referencing another user's table's PRIMARY KEYconstraint, or users are making changes to the data does not prevent you from revoking a user's privileges on the table.
2. **A, C, E** — When you have been granted the CREATE TABLE privilege WITH ADMIN OPTION, you can grant the same privilege to other users with or without the ADMIN OPTION. You can also revoke the privilege from other users as long as you were the one who granted it to them in the first place. You cannot revoke the privilege from the DBA or yourself.
3. **B, E** — When an Oracle database is first created, only the SYS and SYSTEM users are defined by default. The DBA is not a user in Oracle; "DBA" is the name of a role. The user Scott may be created by a script that is used to create a default starter database in an Oracle environment but is not created by default — only as a result of adding sample schemas and data to the database. Bob is never created unless the DBA creates the user.
4. **D** — The only one that cannot be granted to roles is another role WITH GRANT OPTION. This is because WITH GRANT OPTION is not valid for roles — only WITH ADMIN OPTION is. All of the others can be granted to a role.
5. **D** — The IDENTIFIED BY clause of the CREATE USER command specifies a password for a user.
6. **C** — This line causes the GRANT command to fail because a comma is missing after DeveloperRole. The command should read as follows:

```
GRANT CREATE TABLE, CREATE VIEW
CREATE PROCEDURE, CREATE SEQUENCE
TO SCOTT, BOB, DeveloperRole,
SUSAN;
```

7. **A, D, F —** When you have been granted the CREATE VIEW privilege WITH ADMIN OPTION and then grant it to Susan and the privilege is revoked from you, you no longer can create views (A), and you no longer can grant the CREATE VIEW privilege to others (E). Each of these limitations are because the privilege has been revoked.

   Susan still can create views (D) because WITH ADMIN OPTION does not cascade when the permission is revoked from a user. The DBA also must revoke the privilege from Susan, if this is the intention.

8. **B —** The USER_TAB_PRIVS_MADE view shows to whom permissions on tables have been granted and who granted them. The GRANTOR column of the view lists Bob's username whenever he did the granting. The USER_TAB_PRIVS_RECD view shows which tables in other schemas you have been granted permissions to. All other options are not data dictionary views in Oracle 8*i*.

9. **E —** The REVOKE command specifying ALL privileges and CASCADE CONSTRAINTS ensures that any privileges granted, and any constraints that are created by Susan referencing the ClassEnrollment table are removed. If you do not want to automatically drop the constraints and be informed of their existence, option A would work, but the question asked how you can completely remove all privileges.

10. **B, C —** The best way to assign privileges to only the FirstName and LastName columns of the Instructors table to Susan is to create a view with only those columns and then grant Susan all privileges on the view. You also can assign privileges on the columns directly, but you were asked to choose to answers, so the view was the only available option.

    The "Introduction to Oracle: SQL & PL/SQL" exam may present questions similar to number 10 requesting a specific number of answers to solve the problem. Other answers may also work, but you should pay attention to the number of responses the question is looking for. In other words, read the question completely.

Scenarios

1. In order to solve the problems outlined in the scenario, you should recommend that the client create a role for each task that users will be performing. Then other roles should be created according to the job functions of the people in the company, and these roles would be assigned the task-oriented roles previously created. You can grant these user roles to the respective managers WITH ADMIN OPTION so that they can then grant those roles to new users as they join the company.

   Using roles at the task level and user level streamlines the assignment of permissions and also enables delegation. This reduces the workload of the DBAs and also make managers feel that they have control over aspects of the database that relate to their jobs.

2. While you cannot grant other users privileges to create additional databases through roles, the other problems you are experiencing can be solved by using roles. To lessen your workload and to make the creation of database objects more streamlined across a multinational enterprise, you should designate junior DBAs in each key geographic region. Those individuals can be granted a role you create in each database. The role can be granted all system privileges needed to assign permissions to create all relevant database objects. You do not specify WITH ADMIN OPTION on either the role or the system privileges granted to the role because you want the junior DBAs to perform the work and not enable others to do so. From the way the DBA is currently controlling access to any database, control is a very important issue and still needs to be maintained.

Lab Exercises

Lab 8.1 Creating and Managing Users

1. Invoke SQL*Plus and connect to your database instance as the user SYSTEM with a password of manager. If you previously changed the password for the user SYSTEM, make sure you specify the new password when connecting.

```
SQL> connect system/manager@orcl.delphi.bradsys.com
Connected.
SQL>
```

2. Create three new users with usernames of Carol, Ted, and Alice. Make the password for all three users oracle.

```
SQL> CREATE USER Carol IDENTIFIED BY oracle;

User created.

SQL> CREATE USER Ted IDENTIFIED BY oracle;

User created.

SQL> CREATE USER Alice IDENTIFIED BY oracle;

User created.

SQL>
```

3. Attempt to connect to the instance as any one of the users. What happens and why?

```
SQL> connect ted/oracle@orcl.delphi.bradsys.com
ERROR:
ORA-01045: user TED lacks CREATE SESSION privilege; logon
denied


Warning: You are no longer connected to ORACLE.
SQL>
```

In order for a user to be able to connect to an Oracle instance, he or she also needs to be granted the CREATE SESSION system privilege, as the error message indicates.

4. Connect as system and grant each of the users the appropriate privileges to be able to connect to the instance. Connect to the instance as each of the users to ensure that the assignment worked correctly.

```
SQL> connect system/manager@orcl.delphi.bradsys.com
Connected.
SQL> GRANT CREATE SESSION
  2  TO Carol, Ted, Alice;

Grant succeeded.

SQL> connect carol/oracle@orcl.delphi.bradsys.com
Connected.
SQL> connect ted/oracle@orcl.delphi.bradsys.com
Connected.
SQL> connect alice/oracle@orcl.delphi.bradsys.com
Connected.
SQL>
```

5. Connect to the instance as Alice and change your password to newpass. Disconnect and reconnect to the instance as Alice to test the password change.

```
SQL> connect alice/oracle@orcl.delphi.bradsys.com
Connected.
SQL> ALTER USER Alice
  2  IDENTIFIED BY newpass;

User altered.

SQL> disconnect
Disconnected from Oracle8i Enterprise Edition Release 8.1.7.0.0 -
Production
With the Partitioning option
JServer Release 8.1.7.0.0 - Production
SQL> connect alice/newpass@orcl.delphi.bradsys.com
Connected.
SQL>
```

6. Connect to the instance as system and change Ted's password to password. Attempt to connect to the instance as Ted with both the old and new password. What happens when you try the old password?

```
SQL> connect system/manager@orcl.delphi.bradsys.com
Connected.
SQL> ALTER USER Ted
  2  IDENTIFIED BY password;

User altered.
```

```
SQL> connect ted/oracle@orcl.delphi.bradsys.com
ERROR:
ORA-01017: invalid username/password; logon denied


Warning: You are no longer connected to ORACLE.
SQL> connect ted/password@orcl.delphi.bradsys.com
Connected.
SQL>
```

7. Connect to the instance as SYSTEM and drop Alice's user account. Attempt to connect to the instance as Alice. What happens?

```
SQL> connect system/manager@orcl.delphi.bradsys.com
Connected.
SQL> DROP USER Alice;

User dropped.

SQL> connect alice/newpass@orcl.delphi.bradsys.com
ERROR:
ORA-01017: invalid username/password; logon denied


Warning: You are no longer connected to ORACLE.
SQL>
```

Alice is not able to connect to the instance because her user account no longer exists. It was dropped by the DBA.

Lab 8.2 Creating and Managing Roles and Permissions

1. Connect to the instance as SYSTEM and create two new roles called DeveloperRole and UserRole.

```
SQL> connect system/manager@orcl.delphi.bradsys.com
Connected.
SQL> CREATE ROLE DeveloperRole;

Role created.

SQL> CREATE ROLE UserRole;

Role created.

SQL>
```

2. Grant the following privileges to the DeveloperRole:

```
CREATE TABLE
CREATE ANY VIEW WITH ADMIN OPTION
CREATE SEQUENCE WITH ADMIN OPTION
CREATE SYNONYM
```

```
SQL> GRANT CREATE TABLE, CREATE SYNONYM
  2  TO DeveloperRole;

Grant succeeded.

SQL> GRANT CREATE ANY VIEW, CREATE SEQUENCE
  2  TO DeveloperRole WITH ADMIN OPTION;

Grant succeeded.

SQL>
```

3. Grant the DeveloperRole to Carol and the UserRole to Ted.

```
SQL> GRANT DeveloperRole TO Carol;

Grant succeeded.

SQL> GRANT UserRole TO Ted;

Grant succeeded.

SQL>
```

4. Connect as Ted and verify the privileges you have.

```
SQL> connect ted/password@orcl.delphi.bradsys.com
Connected.
SQL> SELECT * FROM ROLE_SYS_PRIVS;

no rows selected

SQL> SELECT * FROM ROLE_TAB_PRIVS;

no rows selected

SQL> col username format a10
SQL> col granted_role format a12
SQL> SELECT * FROM USER_ROLE_PRIVS;

USERNAME   GRANTED_ROLE ADM DEF OS_
---------- ------------ --- --- ---
TED        USERROLE     NO  YES NO

SQL>
```

5. Connect to the instance as Student and grant the following privileges to the UserRole:

```
SELECT ON Student.Courses
SELECT, INSERT, UPDATE ON Student.ClassEnrollment
SELECT ON Student.ScheduledClasses
```

```
     ALL ON Student.Students

SQL> connect student/oracle@orcl.delphi.bradsys.com
Connected.
SQL> GRANT SELECT ON Student.Courses TO UserRole;

Grant succeeded.

SQL> GRANT SELECT, INSERT, UPDATE
  2  ON Student.ClassEnrollment
  3  TO UserRole;

Grant succeeded.

SQL> GRANT SELECT ON Student.ScheduledClasses
  2  TO UserRole;

Grant succeeded.

SQL> GRANT ALL ON Student.Students TO UserRole;

Grant succeeded.

SQL>
```

6. Connect as Ted and verify the privileges that you now have. What has changed?

```
SQL> connect ted/password@orcl.delphi.bradsys.com
Connected.
SQL>  SELECT * FROM ROLE_SYS_PRIVS;

no rows selected
SQL> col role format a10
SQL> col owner format a12
SQL> col table_name format a25
SQL> col privilege format a15
SQL> col column_name noprint
SQL> SELECT * FROM ROLE_TAB_PRIVS;

ROLE       OWNER        TABLE_NAME                PRIVILEGE       GRA
---------- ------------ ------------------------- --------------- ---
USERROLE   STUDENT      CLASSENROLLMENT           INSERT          NO
USERROLE   STUDENT      CLASSENROLLMENT           SELECT          NO
USERROLE   STUDENT      CLASSENROLLMENT           UPDATE          NO
USERROLE   STUDENT      COURSES                   SELECT          NO
USERROLE   STUDENT      SCHEDULEDCLASSES          SELECT          NO
USERROLE   STUDENT      STUDENTS                  ALTER           NO
USERROLE   STUDENT      STUDENTS                  DELETE          NO
USERROLE   STUDENT      STUDENTS                  INSERT          NO
USERROLE   STUDENT      STUDENTS                  SELECT          NO
```

```
USERROLE   STUDENT      STUDENTS                  UPDATE         NO

10 rows selected.

SQL> SELECT * FROM USER_ROLE_PRIVS;

USERNAME   GRANTED_ROLE ADM DEF OS_
---------- ------------ --- --- ---
TED        USERROLE     NO  YES NO

SQL>
```

7. While still connected as Ted, attempt to create a view that includes the StudentNumber, FirstName, and LastName columns of the Students table. What happens?

```
SQL> CREATE VIEW NyStudents AS
  2  SELECT StudentNumber, FirstName, LastName
  3  FROM Student.Students;
FROM Student.Students
             *
ERROR at line 3:
ORA-00942: table or view does not exist


SQL> SELECT COUNT(*) FROM Student.Students;

  COUNT(*)
----------
        12

SQL>
```

You are told that the table Student.Students does not exist and the view is not created. However, querying the table directly to return a count of the number of rows works fine. The problem must be that Ted does not have permissions to create the view or reference the table in the creation of the view.

8. Connect to the instance as Carol and attempt to query data in the Student.Students table. What happens?

```
SQL> connect carol/oracle@orcl.delphi.bradsys.com;
Connected.
SQL> SELECT COUNT(*) FROM Student.Students;
SELECT COUNT(*) FROM Student.Students
                              *
ERROR at line 1:
ORA-00942: table or view does not exist

SQL>
```

Carol is not able to query the Student.Students table. This is because even though the DeveloperRole grants her the privilege to create any view, she has not been granted the UserRole to be able to SELECT from the table.

9. Connect to the instance as student and create the view mentioned in step 7 in such a way that users cannot change data through the view. Grant the UserRole all permissions on the view.

```
SQL> connect student/oracle@orcl.delphi.bradsys.com
Connected.
SQL> CREATE VIEW StudentNameView AS
  2  SELECT StudentNumber, FirstName, LastName
  3  FROM Student.Students
  4  WITH READ ONLY;

View created.

SQL> GRANT ALL ON StudentNameView TO UserRole;

Grant succeeded.

SQL>
```

10. Connect to the instance as Ted and attempt to query the StudentNameView in the Student schema. What happens?

```
SQL> connect ted/password@orcl.delphi.bradsys.com;
Connected.
SQL> SELECT * FROM Student.StudentNameView;

STUDENTNUMBER FIRSTNAME                      LASTNAME
------------- ------------------------------ ----------------
         1000 John                           Smith
         1001 Davey                          Jones
         1002 Jane                           Massey
         1003 Trevor                         Smith
         1004 Mike                           Hogan
         1005 John                           Hee
         1006 Susan                          Andrew
         1007 Roxanne                        Holland
         1008 Gordon                         Jones
         1009 Sue                            Colter
         1010 Chris                          Patterson
         1100 David                          Smith

12 rows selected.

SQL>
```

Because Ted has been granted the UserRole role, and the role has been granted the SELECT permission on the view, Ted is able to see data through the view.

11. Connect to the instance as SYSTEM and grant the appropriate privileges to enable Carol to grant the DeveloperRole to other users.

```
SQL> connect system/manager@orcl.delphi.bradsys.com
Connected.
SQL> GRANT DeveloperRole TO Carol WITH ADMIN OPTION;

Grant succeeded.

SQL>
```

12. Connect to the instance as Carol and grant the DeveloperRole to Ted.

```
SQL> connect carol/password@orcl.delphi.bradsys.com
Connected.
SQL> GRANT DeveloperRole TO Ted;

Grant succeeded.

SQL>
```

13. Connect to the instance as Ted and attempt to create a synonym for the Student.StudentNameView view. Use the synonym in a query.

```
SQL> connect ted/password@orcl.delphi.bradsys.com
Connected.
SQL> CREATE SYNONYM StudentView
  2  FOR Student.StudentNameView;

Synonym created.

SQL> SELECT * FROM StudentView;

STUDENTNUMBER FIRSTNAME                      LASTNAME
------------- ------------------------------ ----------------
         1000 John                           Smith
         1001 Davey                          Jones
         1002 Jane                           Massey
         1003 Trevor                         Smith
         1004 Mike                           Hogan
         1005 John                           Hee
         1006 Susan                          Andrew
         1007 Roxanne                        Holland
         1008 Gordon                         Jones
         1009 Sue                            Colter
         1010 Chris                          Patterson
         1100 David                          Smith

12 rows selected.

SQL>
```

14. Connect to the instance as SYSTEM and revoke the DeveloperRole from Carol.

```
SQL> connect system/manager@orcl.delphi.bradsys.com
Connected.
SQL> REVOKE DeveloperRole FROM Carol;

Revoke succeeded.

SQL>
```

15. Connect to the instance as Ted and drop and then attempt to create the synonym from step 13. What happens and why?

```
SQL> connect ted/password@orcl.delphi.bradsys.com
Connected.
SQL> DROP SYNONYM StudentView;

Synonym dropped.

SQL> CREATE SYNONYM StudentView
  2  FOR Student.StudentNameView;

Synonym created.

SQL>
```

Ted is able to both drop and re-create the synonym because the DeveloperRole was not revoked from him, even though it was revoked from Carol. This is because the WITH ADMIN OPTION does not cascade.

PART

III

Using PL/SQL

This part of the book deals with PL/SQL, an Oracle-specific set of language extensions that enable flow-of-control and logic, looping, conditional branching, handling of errors, and the use of different types of identifiers. None of these elements are defined in the SQL language, so vendors of relational database management systems (RDBMSes) have implemented their own set of constructs. Oracle's is PL/SQL, which stands for Procedural Language Extensions to SQL.

Chapter 9 provides a basic understanding of PL/SQL and where it can be used in Oracle. We discuss the requirements for PL/SQL on the client and the server, the different types of PL/SQL blocks, the basic structure of a block, and the types and use of variables within PL/SQL.

Chapter 10 introduces the basic constructs available in PL/SQL to control the execution of your programs. You first learn about the different types of loops and then about the use of IF statements and related Oracle syntax. We discuss nesting PL/SQL blocks, followed by a discussion of the use of transaction control statements in PL/SQL.

Chapter 11 shows you how to interact with the Oracle database using PL/SQL. We introduce cursors and the special types of variables that are typically used to store database information. You learn how to use DML statements to update the data in your database, followed by more advanced concepts for using cursors, and how to eliminate their declaration altogether using cursor FOR loops.

In Chapter 12 we show you how Oracle handles errors in PL/SQL code and the types of errors that you may encounter. You learn how you can trap errors in PL/SQL and the rules for error propagation. Finally, we briefly discuss coding standards, as well as information on how to debug your PL/SQL programs.

Chapter 13 contains information on how to create stored procedures, triggers, and packages in Oracle. While the "Introduction to Oracle: SQL & PL/SQL" exam does not test your knowledge of this information directly, it is information that is worth knowing because it enables you to make your knowledge of PL/SQL immediately useful.

In This Part

Chapter 9
Introduction to PL/SQL

Chapter 10
Controlling Program Execution in PL/SQL

Chapter 11
Interacting with the Database Using PL/SQL

Chapter 12
Handling Errors and Exceptions in PL/SQL

Chapter 13
Introduction to Stored Programs

CHAPTER 9

Introduction to PL/SQL

EXAM OBJECTIVES

- Declaring variables
  - List the benefits of PL/SQL
  - Describe the basic PL/SQL block and its sections
  - Describe the significance of variables in PL/SQL
  - Declare PL/SQL variables
  - Execute a PL/SQL block
- Writing executable statements
  - Describe the significance of the executable section
  - Write statements in the executable section
  - Execute and test a PL/SQL block
  - Use coding conventions

CHAPTER PRE-TEST

1. What are the benefits of using PL/SQL?
2. What are the various types of PL/SQL blocks and how do they differ?
3. How many sections are in the anonymous block and which keyword starts each section?
4. Name the different types of variables that can be used in PL/SQL, and which ones must be declared.
5. List the scalar datatypes available in PL/SQL.
6. How can a variable be declared based upon the datatype of another previously declared variable?
7. How can comments be added to document the code in a PL/SQL block?
8. What support for SQL is available in PL/SQL?
9. What are two ways of displaying the values of PL/SQL variables on the SQL*Plus screen?
10. Compare and contrast PL/SQL records and tables.

✦ Answers to these questions can be found at the end of the chapter. ✦

While SQL is a powerful language for retrieving and manipulating database data, it lacks the flexibility and processing power of a procedural programming language. This chapter introduces the PL/SQL programming language, which was developed by Oracle in order to extend the functionality of SQL.

PL/SQL is structured much like other procedural languages such as C or PASCAL. In fact, it is based on the Ada programming language, but it also includes seamless support for SQL. This means that it is possible to include, natively within the PL/SQL code, certain SQL statements and most of the SQL functions, operators, and datatypes.

Uses and Benefits of PL/SQL

- List the benefits of PL/SQL

Before the integration of the PL/SQL language in Oracle, applications were limited in how they could retrieve and manipulate database information. The two methods were to send a number of SQL statements to the server in a script file from an interactive tool like SQL*Plus or to embed those statements into a language precompiler called Pro*C. The latter provided the desired processing power but was not trivial to implement. It took several lines of code to explain how to connect to the database, what statement to run, and how to use the results from that statement. There were also differences in the datatypes available in SQL and the precompiled language. PL/SQL addresses the limitations of both of these methods.

The benefits of using PL/SQL include:

- ✦ Modularity
- ✦ Variables
- ✦ Control structures
- ✦ Superior performance
- ✦ Error handling
- ✦ Support for SQL
- ✦ Portability
- ✦ Support for object orientation

Modularity

PL/SQL is a modular language; this means that code is broken down into a number of smaller portions, called *modules*. Modularity provides a number of benefits over creating large scripts with many commands:

- ✦ Each module is smaller, which usually makes it easier to develop and debug.
- ✦ Several developers can work simultaneously on different parts of the program, speeding development time.
- ✦ These modules perform very specific tasks that may be repeated elsewhere in that application, or in another, so they can be reused. This again serves to speed development time.

Variables

SQL does not incorporate the idea of variables for the temporary storage of data. When a SELECT statement is issued from SQL*Plus, for example, the results are simply displayed on the screen or written to an output file, depending on the environment settings. In PL/SQL, you are able to hold onto that result in a variable for manipulation within the block of code. This enables you to do more complex computations than you can do in a single SQL statement.

Control structures

In SQL*Plus script files, each statement must execute before the next, and there is no way to conditionally control whether a statement should run, or how many times it should run. PL/SQL includes statements for looping and making decisions, giving you more control over the flow of the program.

Superior performance

Often an application must include more than one SQL statement to complete a task. For example, you may want to retrieve some information from the database using a SELECT statement, then make a decision based on that data retrieved and make some sort of change accordingly. In a client-server environment, each of these individual calls to the server result in costly network traffic. With PL/SQL, this can all be encapsulated in one block of code and sent to the server at once, resulting in better performance of the application.

Error handling

If an application, say in SQL*Plus, runs individual SQL statements, there is no provision for what to do in the case of an Oracle server error occurring. Within PL/SQL, you can code error handlers to control what statements will run when a particular error is encountered. This makes the PL/SQL code run more smoothly.

Support for SQL

Unlike precompiled languages, PL/SQL enables you to natively embed SELECT, Data Manipulation Language (DML), and transaction processing statements within its code. These statements appear as one executable line of code each. New to Oracle

version 8*i* is native dynamic SQL, which allows SQL statements to be written "on the fly" (at runtime), and even includes Data Definition Language (DDL) and Data Control Language (DCL) statements. PL/SQL also supports all of the SQL operators, datatypes, and most of the SQL functions.

Portability

PL/SQL has no platform-specific statements so it can be run on any platform that Oracle runs on. This means that the code can be written for one database and then transferred to another platform with no rewrite required.

Support for object orientation PL/SQL includes packages, which support many of the characteristics of the object-oriented programming paradigm. Since Oracle version 8, PL/SQL also supports object types, which are constructs that encapsulate operations on data with the data itself. Object-oriented design and programming enables you to better map your real-world objects to constructs in your program, and the result is that it is easier to relate one object to another.

The PL/SQL Engine and Statement Processing

Objective

- Describe the basic PL/SQL block and its structure

PL/SQL code must be written in specific pieces called *blocks*. Because PL/SQL is a compiled language, these blocks must be processed by a compiler before they can execute. Compilation is the process of checking to ensure that the objects referred to in the code exist and that the statements have a valid syntax. After this process is completed, the code can then run, but it must run within a PL/SQL engine.

The PL/SQL engine is not a separate product from the Oracle server, but instead a technology that takes a PL/SQL block and executes it. It can be in one of two places:

- **On the client:** This is the case with Oracle's own development suite of tools called Oracle Developer.
- **On the server:** This means that the block of code is run by the engine that resides on the Oracle server. This is the case when you enter a block of code through SQL*Plus.

Once the block of code starts running in the PL/SQL engine, that engine performs all of the instructions contained within the executable lines *except* any SQL statements that it encounters. These SQL statements must be sent to the database to retrieve or manipulate data from there.

The existence of a PL/SQL engine on the client of an Oracle Developer application enables you to balance your client-server load by deciding on which engine the code should run, client or server.

Types of PL/SQL blocks

PL/SQL blocks come in two varieties:

- ✦ Anonymous blocks
- ✦ Named blocks

Regardless of the type of block, they can be nested within one another to any level.

Anonymous blocks

These are pieces of PL/SQL code that have no header with a name. As such, you send them to the PL/SQL engine through an interactive tool like SQL*Plus, and they run once. Remember that PL/SQL is a compiled language so the block is compiled, run, and then disappears. If you want to run it again, you have to send the entire block to the engine again, where it once again is compiled, run, and then disappears. These anonymous blocks can be saved to script files in the operating system to make rerunning them easier.

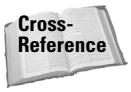
Named blocks or subprograms

A named block can be "called" one or more times by its name. For this reason, the named blocks are often used to implement modularity within a program. The program can be broken down into several modules or subprograms, which can be called one or more times. There are four types of named subprograms:

- ✦ **Procedures:** Perform a task.
- ✦ **Functions:** Carry out a calculation and return a value.
- ✦ **Packages:** Created as one object, these are collections of related procedures and functions.
- ✦ **Triggers:** Code that runs automatically when an event, such as DML, occurs in the database.

Procedures and functions are subprograms that can be created as objects in the database, in the schema of their owner. When this happens, they are called *stored subprograms*. The advantage of using stored subprograms is that they are compiled at creation time and then can be run many times without the processing overhead of recompiling. Packages and triggers are also schema objects, but they cannot be explicitly called by a user. Because a package is a collection of procedures and functions, some of the individual subprograms within the package can be called. Triggers are blocks of code that do not need to be called directly. Simply performing a database event, such as inserting a row into a table, causes the code in a trigger to run, if the trigger is defined for that particular event on that particular table.

The remainder of this discussion of PL/SQL focuses on the structure of anonymous blocks.



For more information on the named subprograms, consult Chapter 13, "Introduction to Stored Programs."

The exam includes questions dealing only with anonymous blocks. Named blocks, or subprograms, are included here and in Appendix B for more information on the "real world" implementation of PL/SQL.

Block structure

The basic structure of an anonymous block is outlined in Figure 9-1.

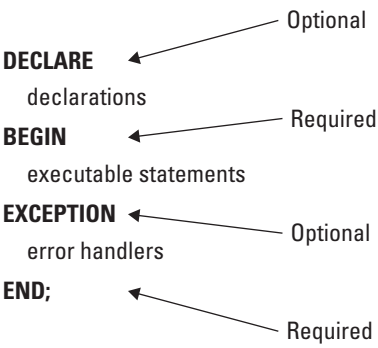


Figure 9-1: Structure of an anonymous block

Note that the DECLARE and EXCEPTION keywords are optional. If you have no variables to declare and you are not performing any error handling, then the block can be as simple as:

```
BEGIN
   statement;
   statement;
   .
   .
   .
END;
```

Remember that the PL/SQL block is free format, so each statement can take up one or more lines and include tabs and spaces. Therefore, to indicate to the compiler where one statement ends and another begins, you *must* end each statement with a semicolon (;).

The convention used in these examples is that each statement is placed on a separate line, and indenting is used to clarify where the block begins and ends. This simply improves readability.

The following is a more detailed look at each of the three sections outlined in Figure 9-1.

Declare section

The DECLARE keyword begins the declare section in an anonymous block. This section is necessary only when identifiers are used in the body of the block. An *identifier* is anything that needs to be declared before it can be used. Examples of identifiers are variables, constants, exceptions, cursors, and user-defined types.

In some programming languages, a variable can be assigned a value without the program having first reserved some area in memory. This usually results in excess storage being used because the datatype of that variable is unknown, and it must be made large enough to store all possible values. PL/SQL does not enable you to waste resources in that fashion. Instead, every variable that is going to be used within the block *must* be declared before it is referenced.

The declaration of each identifier must take place on its own line of code and end with a semicolon (;). The following illustrates a valid declaration of a numeric variable named v_salary, with a precision of 13 and a scale of 2:

```
DECLARE
  v_salary NUMBER(13,2);
...
```

Note that the name of the identifier is given, followed by the datatype. The same method is used for the declaration of constants, collections, and exceptions. Examples of these follow the discussion on each of these identifiers. It is important to remember that you do not have to include the DECLARE keyword for each identifier being declared. Often PL/SQL novices make this mistake:

```
DECLARE
  v_salary NUMBER(13,2);
DECLARE
  v_name VARCHAR2(15);
...
```

when it should look like this:

```
DECLARE
  v_salary NUMBER(13,2);
  v_name VARCHAR2(15);
...
```

Another common mistake made by programmers that are new to PL/SQL is that they try to declare more than one identifier on one line. While this is allowed in many programming languages, it is strictly forbidden in PL/SQL. Therefore, the following code will fail:

```
DECLARE
  v_old_salary, v_new_salary NUMBER(13,2);
...
```

It is also possible to give variables an initial value when they are first declared. This initialization and other declaration options are discussed in detail in the "Variables" section, later in this chapter.

Executable section

The executable section of the block starts with the keyword BEGIN. It is here that you include the main body of your code. Valid statements include PL/SQL variable assignments, loops, decision structures, cursor and subprogram calls, as well as the supported SQL statements: SELECT, INSERT, UPDATE, DELETE, COMMIT, ROLLBACK, and SAVEPOINT.

Expressions in PL/SQL are very much like those in SQL: Date or character literals must be enclosed in single quotes, while numeric literals are not. Furthermore, date literals must be in your Oracle default date format. PL/SQL also supports numbers in scientific notation. For example, 2E5 means $2*10^5$. Expressions are often used in the assignment of values to variables, and this is done using the assignment operator. The assignment operator is := , not just an equal sign as it is in many languages. Following is an example of several variable declaration and assignments using literals:

```
DECLARE
   x NUMBER;
   v_message VARCHAR2(20);
   v_today DATE;
BEGIN
   x := 10;
   x := 2E5;
   v_message := 'Hello';
   v_today := '01-JAN-00'; -- default date format DD-MON-YY
   v_today := SYSDATE;     -- SQL function
END;
```

Notice that expressions can also include SQL functions, like SYSDATE. All of the single-row SQL functions are supported except for DECODE, GREATEST, and LEAST. Group functions are not allowed in PL/SQL expressions, but are still allowed within the SQL statements embedded in the code.

The operators available in PL/SQL expressions include:

- **Arithmetic:** Concatenation (denoted by a double pipe — ||), SQL arithmetic operators (+, -, *, /), plus the exponential (**)
- **Relational:** =, <=, >=, <, >, <>, BETWEEN, IN, LIKE, IS NULL
- **Logical:** AND, OR, NOT

Exception section

The exception section is where error handling occurs. When an error condition is encountered in the PL/SQL engine, an exception is raised. These exceptions may be predefined by Oracle or declared by the programmer (in the declare section). When an exception is raised, the normal flow of the executable section is halted, and control shifts to the exception section for that block, where it searches for a handler for that particular exception. An exception handler is simply a piece of code that indicates what is to happen when a specific exception is raised.

Here is a sample of the flow of control in a simple anonymous block with error handling:

```
DECLARE
  x  NUMBER;
BEGIN
  x := 10/0;   -- this will cause a predefined exception
  y := x + 1;  -- this line will never run
EXCEPTION

  WHEN ZERO_DIVIDE THEN  -- this is the handler
    X := 0;
END;
```

When this block begins, a variable named x is created that will hold numeric data. Then the PL/SQL engine attempts to assign x the value ten divided by zero. Any fifth-grade math student will tell you that this is mathematically impossible so the engine raises an exception. This is a common exception so it has been predefined in the server as ZERO_DIVIDE. Immediately upon raising this exception, control shifts to the exception section where it searches for an appropriate handler, finding it to be the second one. Then the value of x is set to 20, and the flow goes to the next executable line after this block.

If no exceptions are raised, then the lines of code contained in the exception handlers are never executed. If an error is raised and no appropriate handler is found, control returns to the calling environment from which this block was called, with an error condition.

For more information on exceptions, consult Chapter 12, "Handling Errors and Exceptions in PL/SQL."

Comments

The PL/SQL code within any type of block is both case insensitive and free format. This means that you can include tabs and blanks (known as *white space*) as you see fit to make the code more readable. You may also want to include comments about what the code is doing. Comments can take the form of notes embedded in the code for you or other developers, but they must be denoted as comments so that the compiler and PL/SQL engine ignore it.

Some PL/SQL editors recognize that a section of text in your code is a comment and display it in a different color.

The two syntaxes for adding comments in PL/SQL are:

- **Single-line comment:** Denoted by two dashes (–) immediately before the comment, which makes everything to the right of the dashes on that line a comment.
- **Multi-line comment:** This begins with /* and ends with */. Everything in between the opening and closing characters becomes a comment, regardless of how many lines it encompasses.

In this example, both types of comments are used:

```
DECLARE
   x NUMBER := 10; -- x will be used to store numbers
BEGIN
   /* this block will reassign the
      value of the variable x */
   x := 20;
END;
```

Multi-line comments are often used to "comment out" a particular piece of code so that it will no longer be used. This can be more effective than simply deleting it because it shows the evolution of the block of code, and it is simple to revert to an earlier version—just remove the comment markings. Another way in which comments are used is to comment out sections of code while debugging, in order to test smaller sections of the code at one time.

Variables

- Describe the significance of variables in PL/SQL
- Declare PL/SQL variables

Because the storage and manipulation of values is one of the main purposes of database programming, variables are the backbone of any programming language. PL/SQL provides several different types of variables:

- Scalar
- Bind
- Composite
- User-defined
- Reference

With the exception of bind variables, these variables are included in the declare section of the block to reserve space in memory. The values in those variables can then be changed in the executable or exception sections. The names of PL/SQL variables must follow these identifier naming guidelines:

- ✦ Names must be between 1 and 30 characters long.
- ✦ Names must begin with a letter, but the subsequent characters can be letters, digits, or the special characters $, _, or #.
- ✦ Names cannot be PL/SQL reserved words, such as BEGIN or SELECT. For a complete list of reserved words, consult the Oracle documentation.
- ✦ Using double quotes (“) around the name enables you to include characters other than those stated previously, as well as spaces and reserved words. This is not a common practice and can detract from the readability of the code.

Scalar variables

A scalar variable is a placeholder for one value of a specific datatype. It can hold one number, date, character string, or logical value, depending on how it was declared. Declaring a scalar variable is accomplished simply by specifying the name of the variable and its datatype in the declare section of the block in which it is going to be referenced. You can optionally give it an initial value or even enforce that its value can never be NULL. The syntax for declaring scalar variables is:

```
variable_name [CONSTANT] datatype[(size)] [NOT NULL] [{:= or
DEFAULT} initial_value];
```

Note that the word CONSTANT, the size, NOT NULL, and the initialization are optional. Initialization is the assignment of a value to a variable when it is declared, and if a variable is not initialized, its value is NULL. Table 9-1 illustrates several different declarations of a numeric scalar variable x.

Although Oracle provides both the assignment operator (:=) and the DEFAULT keyword for initializing variables, in practice, most PL/SQL developers use just the assignment operator for consistency between the declare and executable sections.

Table 9-1
Variable Declaration Options

Declaration	*Comment*
X NUMBER;	This creates a numeric variable that has no specified size. Because it has not been given an initial value, x has the value NULL.

Declaration	***Comment***
X NUMBER(11,2);	This again has no initial value, but it specifies a size. The method you use to include the optional size element depends on which datatype is used. Here with a NUMBER, you can specify a precision and scale.
X NUMBER(11,2) := 10;	This example includes an initialization using the assignment operator (:=).
X NUMBER DEFAULT 0;	An alternative to the assignment operator is the DEFAULT keyword, which is syntactically equal in the declaration section of the block. The DEFAULT keyword may be used for actual physical defaults to distinguish them from situations where you just happen to be setting the value in the declare section.
X NUMBER NOT NULL := 20;	This example also includes the optional NOT NULL specification, which ensures that this variable can never be assigned NULL as its value. Any attempt to do so results in a runtime error being raised. This means that you *must* initialize it (otherwise, it gets the default value of NULL).
X CONSTANT NUMBER := 3.14;	This declares a constant called x. Unlike variables, constants cannot be assigned new values in the executable section. This means that you *must* also initialize them.

Constants can be used instead of repeating values throughout the executable section. The advantage of using constants is that if you later need to change the value, you only make the change once in the declaration of the constant instead of each time in the code that the value is referenced.

In order to declare a scalar variable, you must specify its datatype. A judicious choice of datatype allows your variable to hold the necessary values, without consuming more resources than necessary. The datatypes available in PL/SQL include all of those available in the Oracle database and fall into one of four datatype families:

- ✦ Numeric
- ✦ Character
- ✦ Date
- ✦ Boolean

While the SQL datatypes are all available in PL/SQL, their range of values may not be the same as they are in database columns. For example, in declaring PL/SQL variables of datatype VARCHAR2, you can specify a maximum size of up to 32,676 bytes, whereas the VARCHAR2 datatype in the Oracle database can only hold up to a maximum of 4,000 bytes,

Numeric datatypes

Though several datatypes can be used to store numeric data, most are subtypes of the NUMBER, BINARY_INTEGER, or PLS_INTEGER datatypes. Subtypes share some, if not all, of the properties of their "base" type. If a subtype shares all of its properties with its base type, then it is equivalent. This subtype name might be just used for clarity or for compliance with a standard from some other database.

The NUMBER datatype holds floating-point representations of positive or negative numbers with magnitudes from 10^{-130} to 10^{125}. It uses 38 digits of precision, although some platforms do not allow this much accuracy in calculations. Some subtypes of NUMBER that are equivalent are FLOAT and FLOATING POINT.

If you know that your variable does not need 38 digits of precision and you do not want to unnecessarily consume your memory resources, then you may want to specify a precision and scale for the variable. For this, you can use the NUMBER(p, s) syntax. For example, the following declares a variable that holds 7 digits, two of which may be decimals:

```
x NUMBER(7,2);
```

The maximum precision that you can specify is 38, while the scale can range from –84 to 127. Specifying a precision with no scale is equivalent to specifying a scale of zero. Then a variable with datatype NUMBER(3) holds integers with three digits. In fact, the subtypes INTEGER, INT, and SMALLINT are all equivalent to NUMBER with a precision specified but scale of zero.

Other subtypes of the NUMBER datatype that are equivalent to NUMBER with a precision and scale are DEC, DECIMAL, and NUMERIC. The REAL subtype is similar to NUMBER(p,s) but the maximum precision is about 18 digits.

The other two numeric base types are BINARY_INTEGER and PLS_INTEGER. While they both hold integers in the range -2147483647 to 2147483647, the PLS_INTEGER performs calculations faster. It is a relatively new datatype in PL/SQL so it is not supported in older versions of the Oracle database, but it should be used in new applications because of the improved performance.

The BINARY_INTEGER datatype has several subtypes, namely POSITIVE, NATURAL, POSITIVEN, NATURALN, and SIGNTYPE. Both POSITIVE and NATURAL denote the subset of integers that are non-negative, while POSITIVEN and NATURALN also add that the variable cannot accept a NULL value. This is usually enforced with the NOT NULL specification in the declaration of the variable, however. Finally, variables declared as SIGNTYPE can have values only of –1, 0, or 1.

Character datatypes

Character datatypes are used to declare variables that hold strings of alphanumeric characters. These strings are either fixed-length or variable-length, and the two main datatypes used are CHAR(n) and VARCHAR2(n). For both of these, the maximum size for n is 32,676 bytes. Keep in mind that the database column datatypes of the same names have a much smaller maximum — 4000 bytes. This can cause problems when writing information to the database from PL/SQL variables that are too large.

For variables declared as CHAR(n), the n defaults to one character if it is excluded, and the n represents the actual size of the variable, padded with blanks. For VARCHAR2, n is mandatory, and it is the maximum size of the variable. If the value assigned to the variable is not n-long, then it does not necessarily take up n bytes of space. For n < 2000, however, it reserves 2,000 bytes of space.

The CHAR datatype has an equivalent subtype CHARACTER, while the VARCHAR2 has equivalent subtypes VARCHAR and STRING. Other types available are NCHAR(n) and NVARCHAR2(n), in which the size n is specified in either bytes or characters, depending on the National Language Support (NLS) settings for your Oracle server. These are included because not all languages can be stored in the same way — languages such as Japanese or Arabic have many symbols and may require more than one byte to store each character.

Similar to the VARCHAR2 datatype is the LONG datatype. It is again used to hold alphanumeric strings up to 32,676 bytes long but has limited usage in PL/SQL. That is, LONG datatypes cannot be used in expressions, SQL function calls, and in many clauses of the INSERT, UPDATE, DELETE, and SELECT statements. They are included only to support the SQL datatype of the same name. This is also true of the RAW and LONG RAW datatypes, which hold binary data such as sound or picture information and also have a maximum size of 32,676 bytes. Problems can occur when retrieving LONG or LONG RAW database columns, which can hold up to 2 GB of data, into their PL/SQL variable counterparts that can only hold 32,676 bytes. For this reason, is it recommended that BLOB, CLOB, or NCLOB PL/SQL datatypes be used as they correspond to the appropriate database datatypes.

The final character datatypes are ROWID and UROWID. These are not supported in SQL but exist only to store the unique address of a database table (called the *rowid*) in a PL/SQL variable. The UROWID, or universal rowid, is a newer datatype that is more flexible in what it stores. For example, a UROWID variable may be able to store rowid information from a non-Oracle database. ROWID and UROWID variables are generally filled up using a SELECT statement such as the following example:

```
DECLARE
   v_selection ROWID;
BEGIN
   SELECT    ROWID
   INTO      v_selection
   FROM      Courses
   WHERE     CourseNumber = 870;
```

```
.
.
.
   UPDATE    Courses
   SET       RetailPrice = 500
   WHERE     ROWID = v_selection;
END;
```

Date datatypes

Used for holding date and time information, the DATE datatype includes the century, year, month, day, hours, minutes, and seconds. It has a range from January 1, 4712 BC to December 31, 9999 AD. If an assignment is made but portions of the value are missing, the defaults are current century, year, and month with day 01 and time 00:00:00 (midnight). For example, the following declaration of a DATE variable includes a partial initialization:

```
DECLARE
   v_EnrollmentDate DATE := TO_DATE('07/11','DD/MM');
```

The day and month have been given, but the century and year will default to the current century and year at runtime, while the time portion will be midnight, the beginning of that day.

Boolean datatypes

Variables declared as BOOLEAN hold only the logical values TRUE, FALSE, and NULL. Unlike other languages where you may be able to use values such as 1, 0, ON, OFF, or others, PL/SQL BOOLEAN datatypes accept only TRUE, FALSE, NULL, or some logical expression that evaluates to one of those values.

```
DECLARE
   x NUMBER := 10;
   y NUMBER := 20;
   v_flag BOOLEAN;
BEGIN
   v_flag := (x > y);
END;
```

In this example, the expression evaluates to FALSE because the value x is not greater than that of y. Before that line executes, the value of v_flag is NULL. These assignments can include any valid logical expression and can use the logical operators AND, OR, and NOT.

The logic tables for the AND, OR, and NOT operators are Tables 9-2, 9-3, and 9-4, respectively. You can use these to determine the outcome of a logical expression, but keep in mind that the operators are evaluated in the following order: NOT, AND, then OR. This means that in the following example, the value of v_flag is TRUE. The logic to this is laid out in Figure 9-2. The OR is the last operator evaluated, so it

splits the entire expression into two cases. In the first, the AND evaluates to FALSE because x is not greater than y and x is not equal to zero. Then the OR evaluates to TRUE because the first case is FALSE, but the second is TRUE:

```
DECLARE
   x NUMBER := 10;
   y NUMBER := 20;
   v_flag BOOLEAN;
BEGIN
   v_flag := (x > y AND x = 0 OR Y IN (20,22,24));
END;
```

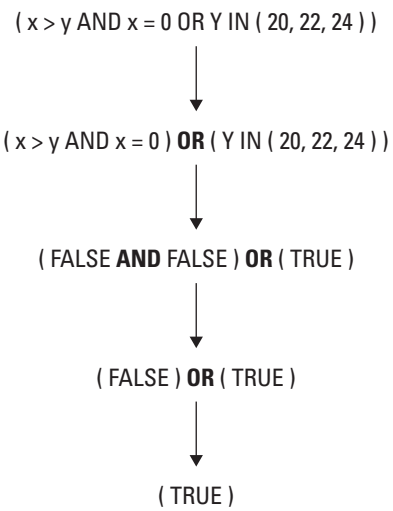


Figure 9-2: Logical steps for the Boolean example

Table 9-2
AND Operator

	TRUE	***FALSE***	***NULL***
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

Table 9-3
OR Operator

	TRUE	*FALSE*	*NULL*
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

Table 9-4
NOT Operator

	TRUE	*FALSE*	*NULL*
NOT	FALSE	TRUE	NULL

All of the datatypes are included here, but the only types that are included on the exam are NUMBER, BINARY_INTEGER, PLS_INTEGER, CHAR, VARCHAR2, LONG, RAW, LONG RAW, DATE, BOOLEAN, and possibly ROWID.

Datatype conversion

The datatype conversion functions TO_CHAR, TO_DATE, and TO_NUMBER are available for use in PL/SQL expressions, just as they are in SQL. These can be used to explicitly change the datatype of an expression, but PL/SQL also does some implicit datatype conversion. For example, the following block runs without exception:

```
DECLARE
   y VARCHAR2(10);
BEGIN
   y := 100;
END;
```

However, for clarity and a slight improvement in performance, you can explicitly change the number *100* to a character string, using:

```
   y := '100';
```

or

```
   y := TO_CHAR(100);
```

This explicit datatype conversion becomes necessary when you have dates in a nonstandard format, such as the following:

```
DECLARE
   v_EnrollmentDate DATE;
BEGIN
   v_EnrollmentDate := TO_DATE('January 01, 2000',
'Month DD, YYYY');
END;
```

Implicit datatype conversion occurs only between numeric and character, or date and character scalar and bind variable expressions.

%TYPE attribute

In some programming languages, you are able to declare more than one variable of the same datatype with one line of code. This is not possible in PL/SQL, but you can declare scalars based on the datatype of a previously declared variable. The %TYPE attribute of a variable or database column returns its datatype and size. The following example illustrates the use of %TYPE:

```
DECLARE
   x NUMBER(7,2) := 100;
   y x%TYPE;
BEGIN
   y := 200;
END;
```

In this example, the datatype of the variable y is determined by the previously declared variable x. Of all the specifications that can be made in the declare statement, only the datatype and size, not the initialized value or the optional NOT NULL specification, is imparted on this second variable. This is convenient for making changes to the datatype at a later date; you need to change only the datatype of the first variable, and all of the variables that reference its datatype with %TYPE are automatically updated.

The %TYPE attribute can also be prefixed by a database column name in order to declare a variable that has the same datatype and size as that column. The syntax for this declaration is as follows:

```
variable_name table_name.column_name%TYPE;
```

As with the previously declared variable, the %TYPE takes only the datatype and size of that column, and not any default value or NOT NULL constraint on it. Therefore, the following statement declares a variable that is able to hold a student's status from the Students table:

```
   v_status students.status%TYPE;
```

The advantage of using %TYPE instead of "hard-coding" the datatype is that you are ensured that no datatype mismatch occurs when you retrieve a value from the database into the variable.

The use of %TYPE makes the variable declaration dynamic. If the datatype or size of a column happens to change, you do not need to change your code to account for the corresponding change in the variable. You do, however, need to recompile your code.

Bind variables

Bind variables are the only type of variables that do not need to be included in the declare section of the block. This is because bind variables are simply placeholders that are "filled in" at runtime with values from the calling environment, or host. For this reason, bind variables are also referred to as *host variables*.

Because a bind variable is used in much the same way as a scalar PL/SQL variable, and it can have the same name as one, you have to distinguish the bind variable by prefixing it with a colon (:). Here is an example of creating a numeric SQL*Plus variable named x that will be used as a bind variable within a PL/SQL block:

```
SQL> VARIABLE x NUMBER
SQL> BEGIN
  2    :x := 100;
  3  END;
  4  /

PL/SQL procedure successfully completed.

SQL> PRINT x

         X
---------
       100
```

Notice that within the PL/SQL block, the variable x is prefixed by the colon (:), whereas outside of the block, in SQL*Plus commands such as VARIABLE and PRINT, it is not. Because this SQL*Plus variable exists for the duration of the session, it can be referenced in more than one PL/SQL block. Therefore, it is said that bind variables are not "local" to one block; they are global variables.

Using a naming convention for global variables, like g_name for a global variable that will hold a name, can further help to distinguish them from local variables.

For more information on SQL*Plus variables consult Chapter 6, "The SQL*Plus Environment."

Composite datatypes

Your PL/SQL blocks of code usually contain many variables for storing and manipulating values, but often these variables are logically related to one another. For example, you may have variables that contain the total sales for each month, or the course number, name, and retail price of a particular course. PL/SQL provides several composite datatypes that are collections of related values:

- ✦ Records
- ✦ VARRAYs
- ✦ Nested tables
- ✦ Index-by tables

Records

A PL/SQL record is a collection of one or more logically related elements called *fields*. Each field must have a name and datatype, and may have a default value and NOT NULL specification. The declaration of a record is similar to scalar variables:

```
record_name  record_type;
```

but these record types are not predefined like the scalar datatypes are. This means that you must first declare the record type as follows:

```
TYPE record_type IS RECORD
(field_name datatype[(size)] [NOT NULL] [:= value],
 field_name datatype[(size)] [NOT NULL] [:= value],
   .
   .
   .);
```

Notice that each field declaration looks like a scalar variable declaration. You can also use the DEFAULT keyword instead of the assignment operator and %TYPE instead of specifying a datatype and size. This is an example of a simple record that holds a person's name, phone number, and birth date:

```
DECLARE
   TYPE person_rec_type IS RECORD
    (first_name    VARCHAR2(20),
     last_name     VARCHAR2(20),
     phone_number  NUMBER(10),
     birthdate     DATE);
person_rec      person_rec_typ;
BEGIN
   SELECT FirstName, LastName, HomePhone, NULL
   INTO   person_rec
   FROM   Instructors
   WHERE  InstructorID = 210;
```

```
.
.
.
   person_rec.first_name := 'Larry';
.
.
.
END;
```

The simplest way to declare a record is to use the %ROWTYPE attribute. Similar to the %TYPE attribute, this enables you to created a record without explicitly stating the datatype of each field. The %ROWTYPE begins with a table name and retrieves the names and datatypes of each of the fields from the columns in that table. For example, the following declares a record that has five fields for holding all of the information on a particular course from the COURSES table:

```
DECLARE
   course_rec courses%ROWTYPE;
BEGIN
   SELECT    *
   INTO       course_rec
   FROM       courses
   WHERE      CourseNumber = 870;

   course_rec.RetailPrice := 900;
.
.
.
END;
```

Note that this record enables you to treat the fields as one logical unit for some operations, such as the SELECT statement, but it also enables you to get and set the values of a particular field using the recordname.fieldname syntax. When using the %ROWTYPE attribute, you have no control over the names of the fields; they are the same as the names of the columns in the table.

VARRAYs, nested tables, index-by tables

While records are useful for storing related information of different datatypes (such as a row of information from a database table), the PL/SQL collection types VARRAY, nested table, and index-by table are better for storing related information that is all of the same datatype.

All three of these enable you to reference one element in the collection using an index value as in this example:

```
total_sales(3) := 10000;
```

If the total_sales collection is declared as a VARRAY type, it is a variable-size array, and when you declare it, you give it an upper bound, like this:

```
TYPE total_sales_typ IS VARRAY(12) of NUMBER(11,2);
total_sales total_sales_typ;
```

This upper bound can later be changed, so it is a variable-size array, but it is not unbounded as the table types are:

```
TYPE total_sales_typ IS TABLE of NUMBER(11,2);  -- nested
```

Or

```
TYPE total_sales_typ IS TABLE of NUMBER(11,2) INDEX BY
   BINARY_INTEGER;     -- index-by
```

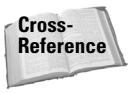
The other main difference between VARRAY and table types is that a VARRRAY always must be dense, whereas tables can be sparse. This means that in a VARRAY with an upper bound of 12, elements 1 through 12 all exist, though their value may be NULL. Nested tables can have elements that have been deleted, and index-by tables have only the elements that have been referenced. The following example shows some of the differences between nested and index-by tables:

```
DECLARE
   TYPE name_nested_typ IS TABLE OF VARCHAR2(20);
   first_name_table name_nested_typ;
   TYPE name_index_by_typ IS TABLE OF VARCHAR2(20) INDEX BY
        BINARY_INTEGER;
   last_name_table name_index_by_typ;
BEGIN

   first_name_table := name_nested_typ('John', 'Paul',
        'George');
   first_name_table.EXTEND;
   first_name_table(4) := 'Ringo';

   last_name_table(3) := 'Smith';
END;
```

This example includes two tables, one nested table that holds first names and one index-by table that holds last names.

Notice that the nested table must be initialized using the constructor method name_nested_typ. This is the same name as the type of the nested table and is a common structure in object-oriented programming that creates an instance of the object with initial values. The first three elements of the nested table have index values 1, 2, and 3. To add a fourth value, you extend the table by one element using the EXTEND method. Then any of the four elements can be deleted or have their values changed.

By contrast, the index-by table that holds last names does not need to be initialized. There are no elements until the value is set for a particular element using its index. The indexes can be any BINARY_INTEGER and do not have to start with one.



The syntax for declaring and using VARRAYs, records, and tables is covered in detail in Chapter 11, "Interacting with the Database Using PL/SQL."

Of the composite datatypes discussed, only records and index-by tables, also known as *PL/SQL tables*, are included on the exam.

User-defined types

In addition to the composite datatypes of records, VARRAYs, nested tables, and index-by tables, PL/SQL enables you to create your own composite datatypes called *object types*. These can have one or more elements, called *attributes*, which are scalars, composite datatypes, or even user-defined object types. Furthermore, you can encapsulate the procedures and functions that work with this data in the same object. This encapsulation of the attributes of an object and the methods that work on it is one of the foundations of object-oriented programming, and for this reason, object types are used for object-oriented programming in Oracle.

One of the simplest forms of object types that you can create is one with data only. For example, the following SQL*Plus statement creates a user-defined datatype that holds address information:

```
SQL> CREATE TYPE address_typ AS OBJECT
  2  (StreetNo      NUMBER(10),
  3   StreetName    VARCHAR2(100),
  4   AptNo         NUMBER(5),
  5   City          VARCHAR2(100),
  6   State         VARCHAR2(100),
  7   ZipCode       NUMBER(9),
  8   Country       VARCHAR2(100))
  9  /

Type created.
```

This type exists in the schema of the user that created it and defines a new datatype that is used in much the same way as VARCHAR2 or NUMBER or any of the other available datatypes in the database. An example that uses this datatype to create a database table with a column of datatype address_typ, then populates a row of that table, follows:

```
SQL> CREATE TABLE people
  2  (ID        NUMBER(5),
  3   FirstName VARCHAR2(100),
```

```
  4   LastName  VARCHAR2(100),
  5   Address   address_typ)
  6  /

Table created.

SQL> INSERT INTO people
  2  VALUES(10,
  3         'John',
  4         'Smith',
  5         address_typ(123, 'Happy Lane', NULL,
  6         'Smalltown', 'Alaska', 12345, 'USA') )
  7  /

1 row created.

SQL> SELECT * FROM people;

    ID FIRSTNAME  LASTNAME   ADDRESS(STREETNO, STREETNAME,
------ ---------- ---------- --------------------------------
    10 John       Smith      ADDRESS_TYP(123, 'Happy Lane',
                             NULL, 'Smalltown', 'Alaska',
                             12345, 'USA')
```

In order to initialize the object type column, you must call the constructor method of the object, which has the same name as the object type. Note that when the object type column is selected, it displays like the constructor method call. You can select, however, just one element from the object type using a "dot notation" in much the same way as with PL/SQL records:

```
SQL> SELECT p.ID, p.FirstName, p.Address.City
  2  FROM   people p
  3  /

    ID FIRSTNAME  ADDRESS.CITY
------ ---------- ------------------------------
    10 John       Smalltown
```

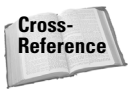
In the address_typ column example, the table alias p is used to prefix the columns selected. Oracle requires the use of table aliases when referencing object type column attributes or methods in order to remove ambiguity. Oracle objects also have a dot notation, which includes the schema and package names.

Reference types

In general, PL/SQL variables are the areas in memory where data is stored, and the datatype of the variable defines what kind of data can be stored there. If instead you want to hold the address of an area in memory, a kind of pointer to some data, then you want to create a variable that is a reference type. It points to an area in memory called a *cursor*.

A cursor is an area in memory where SQL statements are parsed and executed. These cursors are either implicitly declared by the server or explicitly declared by you, the programmer. If you use an explicit cursor for a SELECT statement, then you must follow several steps:

1. **Declare:** Give the area in memory a name and structure.
2. **Open:** Run the SELECT statement and hold onto the result set in that area.
3. **Fetch:** Retrieve one row at a time from the cursor to PL/SQL variables.
4. **Close:** Clear the result set from memory.



The syntax for these steps is discussed in detail in Chapter 11, "Interacting with the Database Using PL/SQL."

To declare a cursor, you simply specify which SELECT statement will populate the area in memory. From this declaration, an area can be reserved that has exactly the right number and datatype of columns. Unfortunately, this means that the cursor can be used only in the block in which it is declared, or in a sub-block of that block.

While a cursor is the actual area in memory, a REF CURSOR is a pointer to an area in memory. This pointer, or reference type, can then be "pointed at" at any area that returns the right kind of values. The following example illustrates the use of a REF CURSOR:

```
DECLARE
   -- First, create the type of the ref cursor

   TYPE course_ref_typ IS REF CURSOR RETURN courses%ROWTYPE;

   -- then create a variable of that type.

   course_ref course_ref_typ;

BEGIN

   OPEN course_ref FOR  SELECT     *
                        FROM       courses
                        WHERE      RetailPrice > 500;
.
.
.
END;
```

In this example, the reference type is declared, and then you can create one or more REF CURSOR variables of that type. These allow much more flexibility than cursors because you can conditionally open them for different SELECT statements, based on some criteria.

Both user-defined and reference types are beyond the scope of the exam but are provided here in order that you may have a more complete picture of the capabilities of Oracle and PL/SQL.

Executing and Testing PL/SQL Blocks

Objective

✦ Execute and test a PL/SQL block

This discussion focuses on the use SQL*Plus to execute and test PL/SQL anonymous blocks. The execution of a block in SQL*Plus is started simply by sending it from the SQL*Plus buffer to the server:

```
SQL> DECLARE
  2    x NUMBER;
  3  BEGIN
  4    x := 10;
  5  END;
  6  /

PL/SQL procedure successfully completed.
```

The server then takes this anonymous block and parses it. It parses the entire block, including any nested sub-blocks, at once. The server checks that the syntax is correct and that the user has the appropriate privileges to carry out any SQL statements contained within the block and, from this, creates a compiled version of the block. That version is then run in the PL/SQL engine.

This process is slightly different for named blocks. For example, if you have a procedure named "book_class" that takes a class ID and a student number as its input parameters, you call this procedure from SQL*Plus as follows:

```
SQL> BEGIN
  2    book_class(51, 1008);
  3  END;
  4  /

PL/SQL procedure successfully completed.
```

Or you can use the EXECUTE command:

```
SQL> EXECUTE book_class(51,1008)

PL/SQL procedure successfully completed.
```

The process is slightly different for stored programs because they have already been compiled, so there is less overhead before the compiled code is sent to the PL/SQL engine.

You can send PL/SQL blocks to the Oracle server in many interactive tools, and each tool can behave differently. Consequently, there is no native functionality for printing data either to a screen or to a printer from within a block. Each tool has its

own methods of enabling you to provide information to the user during block processing. SQL*Plus includes several methods of printing values and providing information to the user. The two most common are:

- ✦ PRINT command
- ✦ DBMS_OUTPUT package

Print command

This method involves printing SQL*Plus variables after a PL/SQL block has run that has populated them with data. Remember that these are host variables that come from the OS, SQL*Plus in this case:

```
SQL> VARIABLE g_result VARCHAR2(100)
SQL> BEGIN
  2   :g_result := 'Hello';
  3  END;
  4  /

PL/SQL procedure successfully completed.

SQL> PRINT g_result

G_RESULT
------------------------------------------------------------
Hello
```

With this method, the bind (or host) SQL*Plus variable must be previously declared with the VARIABLE command. Also note that within the block, the bind variable must be prefixed with a colon (:).

Instead of using the PRINT command after each run of a PL/SQL block, you can change the SQL*Plus environment variable called AUTOPRINT to ON, and for the duration of the session, all SQL*Plus bind variables referenced will be printed automatically.

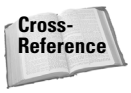
DBMS_OUTPUT package

DBMS_OUTPUT is an Oracle-supplied package that contains procedures and functions for performing file and screen output.

Specifically, the procedure called PUT_LINE enables you to write one line of text to a buffer that can be printed from SQL*Plus. When set ON, the SQL*Plus environment variable SERVEROUTPUT prints to the screen any text in that buffer upon completion of the block of code. The SQL*Plus environment variables, such as

SERVEROUTPUT, retain their setting for the duration of the SQL*Plus session. Therefore, you need to set SERVEROUTPUT ON only once in your session. The following example demonstrates the usage:

```
SQL> SET SERVEROUTPUT ON
SQL> BEGIN
  2   DBMS_OUTPUT.PUT_LINE('Hello');
  3   DBMS_OUTPUT.PUT_LINE('there');
  4  END;
  5  /
Hello
there

PL/SQL procedure successfully completed.
```



For more information on the use of packages, consult Chapter 13, "Introduction to Stored Programs."

Key Point Summary

- ✦ PL/SQL, as a language that incorporates procedural language processing capability with seamless SQL support, offers a number of advantages over precompiled languages and multiple SQL statement script files.
- ✦ PL/SQL is a compiled language that runs in a PL/SQL engine. That engine can reside either on the Oracle Developer client or on the server. When using Oracle's development suite, this means that you can decide where a particular block should run.
- ✦ Blocks of PL/SQL code can either be anonymous, which are compiled at runtime and are not persistent, or named. These named subprograms can be stored in the database as schema objects.
- ✦ Anonymous blocks have three sections: the declare section where all identifiers must be declared, the executable section where the main processing occurs, and the exception section where errors are handled. The declare and exception sections are optional.
- ✦ PL/SQL is free format and case insensitive. Comments and white space can be added for readability. Comments are denoted by two dashes (–) or the multiline switches /* and */.
- ✦ Scalar variables hold data of a particular datatype. PL/SQL supports all of the database datatypes as well as subtypes of these and the BOOLEAN datatype. The datatype must be declared in the declare section but can be taken from another previously defined variable or database column using the %TYPE attribute.

- ✦ Bind variables are placeholders for values from the calling environment, or host. They need not be included in the declare section of the PL/SQL block in which they are referenced. You distinguish bind variables from local scalars by prefixing them with a colon(:).
- ✦ Composite datatypes hold one or more related elements as a logical unit. The available types are records, VARRAYs, nested tables, and index-by tables. Records are composed of elements of different datatypes, while the others are collections of like values.
- ✦ Object-oriented programming is supported in PL/SQL through object types. Reference types are also available to provide pointers to areas of memory.
- ✦ There are two ways of printing to the SQL*Plus screen from within a PL/SQL block: passing values to bind variables or calling the DBMS_OUTPUT.PUT_LINE packaged procedure.

✦ ✦ ✦

STUDY GUIDE

The following section will help you assess your understanding of the benefits of PL/SQL, along with the structure of the PL/SQL block and the types of variables.

Assessment Questions

1. Which of the following SQL statements are supported in PL/SQL? (Select one or more responses.)

   A. DROP TABLE

   B. INSERT

   C. SELECT

   D. GRANT

   E. UPDATE

2. Which of the following types of PL/SQL block cannot be created as a schema object in the database? (Choose the best answer.)

   A. Procedure

   B. Anonymous Block

   C. Package

   D. Function

3. Which of the following is not a valid PL/SQL scalar datatype? (Choose one or more responses.)

   A. DATE

   B. TIME

   C. BOOLEAN

   D. NUMBER

   E. ALPHANUMERIC

4. Evaluate this PL/SQL block:

```
DECLARE
  v_on_hold     BOOLEAN;
  v_total       NUMBER := 3;
  v_class_min   NUMBER := 4;
  v_required    BOOLEAN := TRUE;
```

```
BEGIN
  v_on_hold := (v_total > v_class_min OR v_required);
END;
```

What value does the Boolean variable v_on_hold receive? (Choose the best answer.)

A. NULL

B. TRUE

C. FALSE

D. None of the above

5. Evaluate this PL/SQL block:

```
declare x number:=100; begin x:=x+x/4; exception when
zero_divide then dbms_output.put_line('You cannot divide by
zero'); when value_error then dbms_output.put_line('Wrong
type of value'); end;
```

Although this is a valid block of code, it can be difficult to read and understand what it accomplishes. How might the readability and understanding be improved? (Choose one or more responses.)

A. Use a case convention for keywords and identifiers.

B. Use a separate line for each executable statement.

C. Indent the sections of the block.

D. Add comments to executable lines that explain the program flow.

E. All of the above.

6. In which sections of the anonymous block can the value of a variable be assigned? (Choose one or more responses.)

A. Declare section

B. Executable section

C. Exception section

D. Header section

7. Which of the following declarations creates a PL/SQL record that has elements that stores the contents of one row of the students table? (Choose the best response.)

A. student_rec student.record;

B. student_rec student.ROWTYPE;

C. student_rec student.TABLETYPE;

D. student_rec student%ROWTYPE;

E. student_rec student%RECORD;

8. Evaluate this PL/SQL block:

```
BEGIN
  :g_total := :g_total + 1;
END;
```

Which of the following terms may be used to describe the variable g_total? (Choose the three best responses.)

A. Global variable

B. Host variable

C. Local variable

D. Bind variable

E. Boolean variable

9. Which of the following is a valid assignment of the variable? (Choose the best response.)

A. v_total = 200;

B. v_total := 100

C. v_total := ROUND(v_total,2);

D. v_total := SUM(courses.retailprice);

E. None of the above.

10. Which of the following PL/SQL structures holds no actual data, but instead holds a pointer towards a storage area? (Choose the best answer.)

A. Scalar variable

B. VARRAY

C. Record

D. REF CURSOR

E. Cursor

Scenarios

1. You are the IT manager of a company that has just installed a new Oracle database to keep track of product and order information. The members of your new development team have limited programming experience in Visual Basic, PASCAL, and C. They are going to create a Visual Basic form-based front end for users, but the back end must access the Oracle database. This access will only be for data retrieval and manipulation, but some calculations of tax often will be repeated. You have several choices in how they should do the database access: with multiple SQL calls, precompiled PRO*C procedure calls, or PL/SQL procedure and function calls. Which of these would you choose and why?

2. Working as a developer, you have a need within a PL/SQL block of code to hold the total sales for each month of a particular year. You can store this information in several scalar variables, a VARRAY, a nested table, or in an index-by table. What type of variable(s) would you use to hold these values, and why would you choose that over the other available types?

Lab Exercises

Lab 9-1 Declaring scalar variables

1. Sign on to SQL*Plus as user Student with password oracle.
2. Create an anonymous PL/SQL block with declare and executable sections.
3. Declare a scalar variable named v_1 of datatype NUMBER(7,2) and initialize it as zero.
4. In the executable section, set v_1 to have a value of 100.
5. Print the value of the variable using the DBMS_OUTPUT.PUT_LINE procedure. Remember that you will not see any output unless you have SERVEROUTPUT set ON in your SQL*Plus session.
6. Run your block and confirm that your value of 100 prints on the screen.
7. Edit the block so that you also declare a second variable named v_2 of the exact same datatype, without "hard-coding" the datatype.
8. In the executable section, set v_2 to have a value of one billion (1,000,000,000).
9. Run your block and confirm that the following error occurs:

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

10. Edit the block, change the datatype of v_1 to NUMBER(15,2), and rerun the block. It should complete successfully because v_2 now has this new datatype as well.
11. Save your block in a SQL*Plus script file named lab8-1.sql

Lab 9-2 Enhancing the executable section

1. Edit the PL/SQL block contained in the lab8-1.sql script.
2. Remove the lines of code that set the values of the variables by making them a multi-line comment.

3. Execute the block and confirm that the value printed is now zero, the default for v_1.
4. Edit the block so that it now prints the value of v_2.
5. Execute the block. What is output to the screen and why?

Lab 9-3 Using bind variables

1. Create a numeric SQL*Plus variable named g_sum.
2. Create and execute an anonymous block that sets the value of g_sum to 100. Does this block need a declare section?
3. Create and execute another anonymous block that sets the value of g_sum to be five times what its value was.
4. Print the value of g_sum to the screen.
5. Make a SQL*Plus script file named lab8-3.sql that completes all of the previous steps: creates a SQL*Plus variable, sets the value of that variable in two separate blocks, and then prints the value of it. When do you need to use a colon (:) as a prefix to g_sum? When do you need the SQL*Plus termination character (/)?

Answers to Chapter Questions

Chapter Pre-Test

1. The benefits of using PL/SQL instead of multiple SQL statement calls are because it is a procedural language. As such, it is modular and has constructs like loops, decision structures, variables, and error-handling. It is also preferable to other procedural languages like Pro*C because it has support for SQL, is portable, and allows for object-oriented programming through object types.
2. The two main types of PL/SQL blocks are anonymous and named. The named blocks are usually used to create database objects such as procedures, functions, and packages, while anonymous blocks are not schema objects.
3. There are three sections in the anonymous block: DECLARE begins the declare section, while the executable section starts with the keyword BEGIN, and the word EXCEPTION starts the error-handling exception section.
4. In PL/SQL, variables must be of scalar, composite, user-defined, or reference types, and all must be declared. You may also use bind variables, but they are not declared in PL/SQL but are inherited from the host environment.

5. The base types in PL/SQL are NUMBER, PLS_INTEGER, BINARY_INTEGER, CHAR, VARCHAR2, LONG, RAW, LONG RAW, ROWID, UROWID, DATE, and BOOLEAN. Some of these also have subtypes. In addition, PL/SQL supports LOBs, or large objects.
6. You can use the %TYPE attribute to declare a variable as the same datatype and size as a previously declared variable, or a database column.
7. There are two ways to denote comments in a PL/SQL block: two dashes in a row (--) indicate that the rest of that line is a comment, while multi-line comments start with /* and end with */.
8. PL/SQL allows SELECT, INSERT, UPDATE, DELETE, COMMIT, ROLLBACK, and SAVEPOINT statements embedded directly in the body of a block of code. Also, there is native dynamic SQL, which further allows DDL and DCL commands to be included in the code. You can also use any of the SQL datatypes in PL/SQL.
9. One way to print the value of a PL/SQL variable in SQL*Plus is to pass it to a SQL*Plus bind variable within the block, then PRINT the value of that variable once the block has executed. The other way is to call DBMS_OUTPUT.PUT_LINE, which is an Oracle-supplied packaged procedure. You pass the variable as an input parameter to the procedure, and then it prints the results after the block is executed, provided that the SERVEROUTPUT SQL*Plus environment variable was set ON.
10. Both records and tables are composite PL/SQL datatypes, but records have related elements with differing datatypes and field names, whereas tables have many elements of the same datatype. Table elements have no name but instead have an index value. Records are convenient for holding a database row in memory, while tables are convenient for holding a column or other data that is all of the same type.

Assessment Questions

1. **B, C, E** — The only SQL statements allowed are data retrieval (SELECT), data manipulation (INSERT, UPDATE, DELETE), and transaction processing (COMMIT, ROLLBACK, SAVEPOINT). Data definition (CREATE, DROP, ALTER) statements, along with data control (GRANT, REVOKE) commands, are not supported natively in PL/SQL. Refer to the "Support for SQL" part of the "Uses and Benefits of PL/SQL" section, earlier in this chapter.
2. **B** — Procedures, functions, and packages can be made into schema objects and then be called more than once. By contrast, anonymous blocks run once and then disappear. Refer to the "Types of PL/SQL Blocks" section, earlier in this chapter.
3. **B, E** — TIME and ALPHANUMERIC are not valid datatypes. The DATE type includes a time element, and while alphanumeric characters are allowed in CHAR or VARCHAR2 types, ALPHANUMERIC is not a valid subtype. Refer to the "Variables" section, earlier in this chapter.

4. **B** — The OR operator has the lowest precedence, so the "greater than" operator (v_total > v_class) executes first. This returns a result of FALSE, but the other condition, (v_required) is TRUE. This means that the expression is (FALSE OR TRUE), which evaluates to TRUE. Refer to the logic tables in the "Boolean Datatypes" section, earlier in this chapter.
5. **E** — All of the solutions provided work toward better readability by showing clearly where each line begins and ends, and what action it performs. Refer to the "Comments" section.
6. **A, B, C** — A PL/SQL variable can be assigned a value when it is declared in the declare section or in any executable statement. Executable statements are allowed in the executable and exception sections. Refer to the "Block Structure" section, earlier in this chapter.
7. **D** — The %ROWTYPE attribute can be prefixed with a table name in order to create a record with the same number and datatype of fields as the columns from that table. Refer to the "Composite Datatypes" section, earlier in this chapter.
8. **A, C, D** — The colon (:) before the variable name indicates that it is a bind variable. As a bind variable, it is not declared inside this block, but instead it is a placeholder for a value from the calling environment, or host. That is why it is also known as a *host variable*. Finally, because this bind variable can be referenced in many blocks and is not local to one, it can also be called a *global variable*. Refer to the "Bind Variables" section, earlier in this chapter.
9. **C** — While single-row functions (such as ROUND) are allowed in PL/SQL expressions, group functions (such as SUM) are not. The first answer is incorrect because the assignment operator is :=, not =. The second assignment statement is missing the semicolon (;) at the end. This leaves only answer C as valid. Refer to the "Executable Section" part of the "Block Structure" section, earlier in this chapter.
10. **D** — The REF CURSOR is a reference type, which means that it references an area of memory instead of actually representing that area. Refer to the "Reference Types" section, earlier in this chapter.

Scenarios

1. In this situation, where you need to make many data retrieval and manipulation statements, Pro*C is probably not a good choice. This is certainly the situation if your developers do not have extensive experience with the Pro*C language. PL/SQL allows these types of statements natively so the access can be easily done. As for multiple SQL calls, these lack the modularity of PL/SQL procedures. This means that every time that you need to calculate the tax in a SELECT statement, you have to rewrite the expression. With PL/SQL, you can create a function that does the tax calculation and then call that function many times. Therefore, PL/SQL functions and procedures are the best choice.

2. Because this data is closely related and all of the same datatype (numeric totals), one of the collections would be preferable to 12 separate variables. They can be treated like one logical unit and share one name. Of the collection types, the VARRAY is best in this situation because you have a fixed number of elements (12 months) and you won't ever need to delete one. The ability to do so is one of the main benefits of the table types but is not necessary in this situation.

Lab Exercises

Lab 9-1 Declaring scalar variables

5. Your code should look like this:

```
SET SERVEROUTPUT ON
DECLARE
  v_1 NUMBER(7,2) :=0;
BEGIN
  v_1 := 100;
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_1));
END;
/
```

Although it is not necessary, the TO_CHAR is included here. Implicit datatype conversion takes the numeric variable and changes it to the character string that the PUT_LINE procedure accepts, but explicitly specifying this is slightly better from a performance standpoint.

11. Your final block should look like this:

```
SET SERVEROUTPUT ON
DECLARE
  v_1 NUMBER(15,2) :=0;
  v_2 v_1%TYPE;
BEGIN
  v_1 := 100;
  v_2 := 1000000000;
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_1));
END;
/
```

Lab 9-2 Enhancing the executable section

5. The final code should look like this:

```
SET SERVEROUTPUT ON
DECLARE
  v_1 NUMBER(15,2) :=0;
  v_2 v_1%TYPE;
```

```
BEGIN
/*  v_1 := 100;
  v_2 := 1000000000; */
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_2));
END;
/
```

There will appear to be no output because the value of v_2 was not initialized. In SQL*Plus a NULL value appears as a blank line.

Lab 9-3 Using bind variables

1. The declaration of the SQL*Plus variable is:

```
SQL> VARIABLE g_sum NUMBER
```

2. The block should look like this:

```
BEGIN
  :g_sum := 100;
END;
```

Because no local PL/SQL variables are being used, a declare section is not necessary.

3. The block should look like this:

```
BEGIN
  :g_sum := :g_sum * 5;
END;
```

4. The output should look like this:

```
SQL> PRINT g_sum

   G_SUM
---------
     500
```

5. The entire script should look like this:

```
VARIABLE g_sum NUMBER
BEGIN
  :g_sum := 100;
END;
/
BEGIN
  :g_sum := :g_sum * 5;
END;
/
PRINT g_sum
```

Notice that the bind variable g_sum must begin with the colon (:) only when it appears inside a PL/SQL block. Also, the SQL*Plus termination character (/) comes after each PL/SQL block is completed but not after the SQL*Plus commands VARIABLE and PRINT.

CHAPTER 10

Controlling Program Execution in PL/SQL

EXAM OBJECTIVES

- Writing Control Structures
  - Identify the uses and types of control structures
  - Construct and identify different loop statements
  - Control block flow using nested loops and labels
  - Construct an IF statement
- Describe the rules of nested blocks
- Control transactions in PL/SQL
- Execute and test a PL/SQL block

CHAPTER PRE-TEST

1. How many types of loops are available in PL/SQL, and what are they called?
2. When do you use a FOR loop?
3. How can you tell which loop will be exited with the EXIT statement?
4. How many different conditions can be tested in an IF statement?
5. What are labels used for?
6. Can you nest one block within another? Why would you want to?
7. What statements are used to control transactions within a PL/SQL block?
8. How do you set intermediate points in a transaction so that some, but not all, of the DML statements may be rolled back?
9. What does the ELSIF clause in an IF statement do?
10. How does a WHILE loop differ from a basic loop?

✦ Answers to these questions can be found at the end of the chapter. ✦

Within computer languages, it is necessary to control the order in which lines of code are executed. While some lines must be executed more than once, others need not be executed at all, depending on certain conditions. For this reason, PL/SQL has a variety of loops and decision structures that are similar to those found in many other programming languages.

Loops

Objective

- Construct and identify different loop statements

Loops are used to execute a portion of code more than once, without having to duplicate that code in the program. In programming, you often want to do the same thing for each day of the week, for each order placed with your company, or for some other event that occurs a number of times. Each of the three types of loops available in PL/SQL has a different syntax and usage. The loop types are:

- ✦ Basic loop
- ✦ WHILE loop
- ✦ FOR loop

Basic loop

The basic loop is basic in its syntax. The following is an example of an anonymous block that prints five lines to the SQL*Plus screen. Remember that the SQL*Plus environment variable SERVEROUTPUT must be set on at some point in the SQL*Plus session in order to view the results of the DBMS_OUTPUT.PUT_LINE procedure.

```
SET SERVEROUTPUT ON
DECLARE
  x NUMBER := 1;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('This loop has executed '
                ||TO_CHAR(x)||' time(s)');
    x := x +1;
    EXIT WHEN x > 5;
  END LOOP;
END;
```

Tip

This example can work without the use of the TO_CHAR function. Oracle performs implicit conversion of the numeric variable x to a character datatype that is expected by the concatenation, but it is better for performance to do the conversion explicitly.

In this example, x is a variable used in a counter that keeps track of how many times the loop has executed. After the line has been written the fifth time, the Boolean condition (x > 5) becomes TRUE, and the program moves to the next line following the END LOOP. Because there are no more executable lines at this point, the program ends. The output should look like this:

```
This loop has executed 1 time(s)
This loop has executed 2 time(s)
This loop has executed 3 time(s)
This loop has executed 4 time(s)
This loop has executed 5 time(s)

PL/SQL procedure successfully completed.
```

Within the LOOP and END LOOP statements, you may place as many executable statements as you like, but remember that each one must end with a semicolon (;). The complete syntax for the basic loop follows:

```
LOOP
  statement;
  .
  .
  .
  [EXIT [WHEN condition];]
END LOOP;
```

Note that the EXIT statement is optional, and in fact, it can appear anywhere within the loop. Without an EXIT statement, the loop is endless, although the program may abort at some point when a variable becomes too large. For example, in the earlier example that printed the number of times through the loop, the program ends in error when the default 2,000-byte maximum for the DBMS_OUTPUT buffer is reached.

Also note that the WHEN condition is optional in the exit statement. A loop with an unconditional EXIT statement does not loop at all, so in practice, WHEN is always added. The loop exits only when the condition evaluates to TRUE, not when it is FALSE or NULL.

Refer to the logic tables section of Chapter 9, "Introduction to PL/SQL," for more information on Boolean expressions.

One well-used coding convention is that the LOOP and END LOOP statements should be lined up with the contents of the loop indented.

WHILE loop

The WHILE loop is similar to the basic loop, but it has the exit condition built into the first line. The syntax for the WHILE loop follows:

```
WHILE condition LOOP
  statement;
  .
  .
  .
END LOOP;
```

Notice that no EXIT statement is needed here because the program continues to loop only as long as the WHILE condition is TRUE. You may run into a situation where you also need to include another EXIT statement inside the loop to exit based on some other criteria. Actually, if the WHILE condition is not TRUE when the program first encounters the loop, the statements within the loop are never executed.

Here is a WHILE loop implementation of the exact same program that prints five lines:

```
DECLARE
 v x NUMBER := 1;
BEGIN
  WHILE x <= 5 LOOP
    DBMS_OUTPUT.PUT_LINE('This has executed '
                  ||TO_CHAR(x)||' time(s)');
    x := x +1;
  END LOOP;
END;
```

In this example, the condition has "turned around." Instead of exiting when x is greater than 5, you continue looping until x is no longer less than or equal to 5. Both of these blocks should produce the exact same results.

FOR loop

In most programs where looping is required, you know how many times the loop needs to be repeated. To repeat a portion of code with basic loops and WHILE loops, you need to declare a numeric variable to keep track of the number of times that the loop has executed. You also need to increment that variable, or counter, each time through and include a statement that specifies when to leave the loop. Because the repetition of code a set number of times is such a common practice, there is a shortcut: the FOR loop.

The FOR loop has a built-in counter that need not be declared or incremented by the programmer. Like the WHILE loop, it has the exit conditions specified at the beginning. Here is the FOR loop implementation of the program that prints five lines:

```
BEGIN
  FOR x IN 1..5 LOOP
    DBMS_OUTPUT.PUT_LINE('This has executed '
                ||TO_CHAR(x)||' time(s)');
  END LOOP;
END;
```

The first thing that you should notice when comparing this example to the other loop implementations is that it is much shorter than the other two. Because the FOR loop implicitly declares the counter (also called an *index*), you need not have a DECLARE section. Also missing here is the line of code that adds 1 to the counter each time through the loop. Each time the program loops around to the first line, it automatically increments the counter by 1. It then checks that the counter is still within the range specified.

Tip

FOR loops are available in many programming languages, with the long-standing convention of using the lowercase letter "i" as the counter, or index name. You may use any name you choose, subject to PL/SQL variable-naming guidelines, but code in this book reflects the generally used naming convention.

In the previous example, the lower bound for the index is the numeric literal 1, while the upper bound is the numeric literal 5. In general, these bounds can be any integer expressions. The next example uses an upper bound that is a variable:

```
DECLARE
  v_upper INTEGER := 5;
BEGIN
  FOR i IN 1..v_upper LOOP
    DBMS_OUTPUT.PUT_LINE('This has executed '
                ||TO_CHAR(i)||' time(s)');
  END LOOP;
END;
```

Caution

If the bounds of a FOR loop are set using variables, and a variable value happens to change during an iteration of the loop, the loop does not change its exit condition. The value of each bound is captured when the loop first starts and does not reflect any changes in variable values.

You also have the choice of having the index go in reverse. This means that it can count backwards, say from 10 down to 1. Unlike other programming languages, you may not have the counter step by any value other than 1. The full syntax for the FOR loop follows:

```
FOR counter IN [REVERSE] lower_bound .. upper_bound LOOP
  statement;
  .
  .
  .
END LOOP;
=
```

The FOR loop is generally used when a group of statements must be executed a set number of times, and this number is known when the loop begins. It is possible to leave the loop before the counter reaches the upper bound using the EXIT WHEN statement, as in the next example. This assumes the existence of some sort of Contributions table that holds information on pension contributions for each instructor. The program accepts, from the user, the instructor's ID, monthly contribution, and limit for the year. It then loops through the months, adding each month's contribution (on the first of the month) for a particular instructor to the table until the limit for the year has been exceeded.

```
DECLARE
   v_id Instructors.InstructorID%TYPE := &p_id;
   v_monthly_contrib NUMBER(13,2) := &p_contribution;
   c_yearly_limit CONSTANT NUMBER(13,2) := &p_limit;
   v_total_contrib NUMBER(13,2) := 0;
BEGIN
   FOR i IN 1..12 LOOP
        INSERT INTO Contributions(InstructorID,
             ContDate, Amount)
        VALUES(v_id, TO_DATE(TO_CHAR(i),'MM'),
              v_monthly_contrib);
        v_total_contrib := v_total_contrib + v_monthly_contrib;
        EXIT WHEN v_total_contrib > c_yearly_limit;
   END LOOP;
END;
```

Here is a sample run of the program with a limit that is reached before the year is over:

```
SQL> /
Enter value for p_id: 450
Enter value for p_contribution: 200
Enter value for p_limit: 840

PL/SQL procedure successfully completed.

SQL> SELECT *
  2  FROM   Contributions;
```

```
INSTRUCTORID CONTDATE     AMOUNT
------------ ---------- ---------
         450 01-JAN-01         200
         450 01-FEB-01         200
         450 01-MAR-01         200
         450 01-APR-01         200
         450 01-MAY-01         200
```

In the preceding example, the fifth contribution of $200 puts the instructor over the $800 limit, so it is the last one made. If the yearly limit is never reached, then the FOR loop finishes in its normal way—when the upper bound is reached. However, it may leave early. These types of situations can also be handled with a basic or WHILE loop. You must decide whether it is more advantageous to use the FOR loop to get the implicit counter or whether you prefer the readability of the single exit condition in the following code:

```
DECLARE
   v_id Instructors.InstructorID%TYPE := &p_id;
   v_monthly_contrib NUMBER(13,2) := &p_contribution;
   c_yearly_limit CONSTANT NUMBER(13,2) := &p_limit;
   v_total_contrib NUMBER(13,2) := 0;
   v_counter NUMBER(2):= 1;
BEGIN
   WHILE v_counter <= 12 AND
             v_total_contrib <= c_yearly_limit LOOP
        INSERT INTO Contributions(InstructorID,
              ContDate, Amount)
        VALUES(v_id, TO_DATE(TO_CHAR(v_counter),'MM'),
               v_monthly_contrib);
        v_total_contrib := v_total_contrib + v_monthly_contrib;
        v_counter := v_counter + 1;
   END LOOP;
END;
```

Notice that in the preceding FOR loop example, the value of the index, i, can be used within the FOR loop. It cannot be used outside of the loop, and its value cannot be changed anywhere but by the loop itself. The following is invalid code with the errors that are returned:

```
1  BEGIN
2    FOR i IN 1..5 LOOP
3      DBMS_OUTPUT.PUT_LINE('This has executed '
4                  ||TO_CHAR(i)||' time(s)');
5      i := 4;  -- invalid
6    END LOOP;
7    DBMS_OUTPUT.PUT_LINE('Now the value of i is '
8                  ||TO_CHAR(i));  -- invalid
9* END;
```

```
SQL> /
BEGIN
*
ERROR at line 1:
ORA-06550: line 5, column 5:
PLS-00363: expression 'I' cannot be used as an assignment target
ORA-06550: line 5, column 5:
PL/SQL: Statement ignored
ORA-06550: line 8, column 27:
PLS-00201: identifier 'I' must be declared
ORA-06550: line 7, column 3:
PL/SQL: Statement ignored
```

The first error is in line 5 where the program attempts to change the value of i to the number 4. In each iteration of the FOR loop, the index is treated like a constant, so you cannot change its value. The second error is in line 8 where the index of the loop is used outside of the loop. This fails and returns the same error as when any undeclared variable is used.

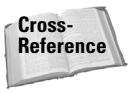
If you need to know what the value of the index was when the loop was exited, then within the loop, you can assign its value to a local variable that has been declared. This variable is accessible outside of the loop. Alternatively, you could just use a basic loop with a local variable as the counter.

Nested loops and labels

- Control block flow using nested loops and labels

In programming, the need often arises to nest one loop inside of another. These are called *nested loops*, and PL/SQL places no restrictions on the number of levels that you nest.

An example of when you might use nested loops is a report that loops through the months to date, totaling the sales per day of the week. The following example creates such a report of student enrollments by day of the week and by month. To print the month and day abbreviations in this example, PL/SQL tables are used. In order to create the month_list and day_list tables, the table type (char4list) must first be declared as a collection of elements that are four characters in length. Also note that these lists are initialized in the DECLARE section using the constructor method that shares the same name as the table type (char4list).



For more information on composite datatypes and retrieving database information in PL/SQL, consult Chapters 9, "Introduction to PL/SQL," and 11, "Interacting with the Database Using PL/SQL."

```
DECLARE
  v_current_month INTEGER := TO_CHAR(SYSDATE, 'MM');
  v_enrollments   INTEGER;
  TYPE char4list IS TABLE OF CHAR(4);
  month_list char4list := char4list('Jan', 'Feb', 'Mar',
          'Apr', 'May', 'June', 'July', 'Aug', 'Sept',
          'Oct', 'Nov', 'Dec');
  day_list char4list := char4list('Sun', 'Mon', 'Tues',
                   'Wed', 'Thur', 'Fri', 'Sat');
BEGIN

  FOR i IN 1..v_current_month LOOP
     /* for months up to, and including the current one */

    DBMS_OUTPUT.PUT_LINE(month_list(i));
        /* print the month name */

    FOR j IN 1..7 LOOP    /* for each day of the week */

      SELECT COUNT(*)
      INTO   v_enrollments
      FROM   ClassEnrollment
      WHERE  TO_CHAR(EnrollmentDate, 'fmYYYYMMD') =
          TO_CHAR(SYSDATE,'YYYY')||TO_CHAR(i)||TO_CHAR(j);

      /* count the number of enrollments in the current
       year, with month and day of the week numbers from the
    respective loop indexes,(i) for months and (j) for days*/

      DBMS_OUTPUT.PUT_LINE(day_list(j)||
' '||v_enrollments);
        /* print the day of the week and the enrollments */
    END LOOP;
  END LOOP;
END;
```

The results of this program, assuming the current month is February and no enrollments were made in the current month, looks something like the following:

```
Jan
Sun  0
Mon  2
Tues 1
Wed  0
Thur 0
Fri  1
Sat  0
Feb
Sun  0
Mon  0
```

```
Tues 0
Wed  0
Thur 0
Fri  0
Sat  0

PL/SQL procedure successfully completed.
```

Notice that the index for the inner loop was chosen to be j. It is legal to reuse the same index, but there is no way from inside the inner loop to reference the outer loop's index value. The way to avoid this confusion is to label each loop in this fashion:

```
<<month_loop>>
FOR i IN 1..v_current_month LOOP
     /* for months up to, and including the current one */

  DBMS_OUTPUT.PUT_LINE(month_list(i));
        /* print the month name */
    <<day_loop>>
  FOR i IN 1..7 LOOP    /* for each day of the week */

      SELECT COUNT(*)
      INTO   v_enrollments
      FROM   ClassEnrollment
      WHERE  TO_CHAR(EnrollmentDate, 'fmYYYYMMD') =
          TO_CHAR(SYSDATE,'YYYY')||
          TO_CHAR(month_loop.i)||TO_CHAR(day_loop.i);

      /* count the number of enrollments in the current
       year, with month and day of the week numbers from the
       respective loop indexes */

      DBMS_OUTPUT.PUT_LINE(day_list(day_loop.i)||
                           ' '||v_enrollments);
       /* print the day of the week and the enrollments */
  END LOOP day_loop;
END LOOP month_loop;
```

Labels can be used in several ways. Here the labels distinguish between the two loops so that prefixes can be added to the index to avoid ambiguity. The index, i, uses a "dot notation" where you name the loop followed by a period and then the loop index. When labeling a loop, you simply enclose the label name in a set of the symbols << and >> on the line immediately before the loop starts.

Optionally, you can include a loop's label name in the END LOOP statement, just before the semicolon. While it is not necessary, this can help you distinguish which END LOOP belongs to which loop when one is nested inside of another.

The GOTO statement

Labels can be added before any executable line of code in a program, and you can branch to those labels using the GOTO statement. The syntax can often look as confusing as this:

```
BEGIN
  <<section1>>
  statement A;
  GOTO section3;
  <<section2>>
  statement B;
  <<section3>>
  statement C;
  IF condition THEN
    GOTO section2;
  END IF;
END;
```

You may find this code difficult to follow. Consider the following code that is equivalent, and then go back and review the previous example; you might have a better chance of understanding the code.

```
BEGIN
  statement A;
  statement C;
  WHILE condition LOOP
    statement B;
    statement C;
  END LOOP;
END;
```

The FOR loop is not the only type of loop that can be nested. Any type of loop can be nested within any other type. In the following example, the EXIT condition exits only one level out of the loop:

```
LOOP
  statements;
  LOOP
    statements;
    EXIT WHEN condition;
  END LOOP;
  statements;
  EXIT WHEN condition;
END LOOP;
```

Sometimes there may be a need to leave the outer loop from inside the inner loop. In order to do so, use labels in the EXIT statement like this:

```
<<outer>>
LOOP
  statements;
  <<inner>>
  LOOP
    statements;
    EXIT outer WHEN condition;
    EXIT WHEN condition;
  END LOOP inner;
  statements;
  EXIT WHEN condition;
END LOOP outer;
```

When nesting one loop inside another, the inner loop must be completely enclosed by the outer. It is illegal to end the outer loop before the inner, and it results in a compile error.

Overuse of the GOTO statement is considered bad structural programming and is frowned upon by most developers. Therefore, it is not covered on the exam and is included here just so that you know about all of the choices available for controlling the flow of a PL/SQL program.

Conditional Processing

✦ Construct an IF statement

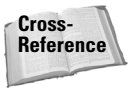
One of the main advantages of using PL/SQL instead of simply creating SQL*Plus script files full of SQL statements is that you can decide, within the program, whether or not a particular line of code should execute. This conditional processing is accomplished using the IF statement.

IF . . . THEN

The simplest IF statement looks like this:

```
IF condition = TRUE THEN
  statement;
  .
  .
  .
END IF;
```

In this structure, the statements execute only when the condition evaluates to TRUE. If it evaluates to FALSE or NULL, the next line executed is the line following the END IF. Note that the condition can be any variable or expression that returns a Boolean (TRUE or FALSE). In fact, the "= TRUE" can be left out entirely if the condition is a complete Boolean expression on its own.



For more information on Boolean conditions, consult Chapter 9, "Introduction to PL/SQL."

The following example uses an expression:

```
IF v_num_confirmed < v_class_max THEN
  INSERT INTO ClassEnrollment (ClassID, StudentNumber,
        Status, EnrollmentDate, Price)
  VALUES (53, 1008, 'Hold', SYSDATE, 1500);
END IF;
```

In this example, if one, or both, of the numeric variables v_num_confirmed and v_class_max are NULL, then the Boolean condition evaluates to NULL, and the insert is not executed. The following code is logically equivalent to the preceding:

```
v_room_left := v_num_confirmed < v_class_max;
IF v_room_left THEN
  INSERT INTO ClassEnrollment (ClassID, StudentNumber,
        Status, EnrollmentDate, Price)
  VALUES (53, 1008, 'Hold', SYSDATE, 1500);
END IF;
```

The only difference between this and the previous example is that the expression is assigned to a variable, v_room_left, of BOOLEAN datatype. This may not flow as well as the previous code as far as readability, but if the condition is going to be checked several times later on in the code, then the PL/SQL engine will have to evaluate the two variables and compare them for each check. Using the second method of storing the result of the comparison in a separate variable cuts down on the amount of work needed for subsequent checks.

In the previous example, the row is inserted into the table if the condition is met, but what if you want it to perform some other operation when the condition is not met? You can use the ELSE clause for this purpose.

ELSE

You may use the ELSE clause to perform an alternative set of statements when an IF condition is not met. The syntax looks like this:

```
IF condition THEN
  statement A;  -- Execute if the condition is TRUE
ELSE
  statement B;  -- Execute if the condition is FALSE or NULL
END IF;
```

No matter what the Boolean condition is, you know that exactly one of either statement A or B will execute and then the program will continue on to the next line after the END IF. Here is the student booking example with an alternative added:

```
v_room_left := v_num_confirmed < v_class_max;
IF v_room_left THEN
  INSERT INTO ClassEnrollment (ClassID, StudentNumber,
        Status, EnrollmentDate, Price)
  VALUES (53, 1008, 'Hold', SYSDATE, 1500);
ELSE
  DBMS_OUTPUT.PUT_LINE(
           'Sorry, there is no room left in that class');
END IF;
```

The next question becomes, "What if you have more than two alternatives for a condition?" For example, assuming you have three locations to hold courses (New York, San Francisco, and Toronto) and the tax on the sale of a course depends on the state or province in which it is held, you can calculate the total cost using nested IF statements like this:

```
IF v_state = 'CA' THEN
  v_cost := v_retailprice * 1.08;
ELSE
  IF v_state = 'NY' THEN
    v_cost := v_retailprice * 1.09;
  ELSE
    /* It must be 'ON' */
    v_cost := v_retailprice * 1.15;
  END IF;
END IF;
```

The preceding example works, but it assumes that only these three choices exist, and it can be a little difficult to follow. Some programming languages have a CASE statement to cope with more than two alternatives in a decision structure, but PL/SQL does not. What it does contain is the ELSIF clause.

ELSIF

You can add as many ELSIF conditions as you need to evaluate different conditions in a decision structure in this manner:

```
IF first_condition THEN
  statement;
ELSIF second_condition THEN
  statement;
ELSIF third_condition THEN
  Statement;
.
.
.
```

```
ELSE
  statement;
END IF;
```

In this structure, if the first condition is met, then the first statement is executed, and control passes to the next line after the END IF; but if it evaluates to FALSE or NULL, then the second condition is tested, and so on. As soon as one condition is found to be TRUE, the others are not tested — it executes the corresponding code and then leaves the decision structure. If all of the conditions fail, then the ELSE statement executes. The ELSE statement is optional, and when it is left out, it is possible that no statements will be executed within the decision structure.

In Oracle, the ELSIF clause of the decision structure is spelled without an "E" after the "S", unlike many other programming languages, which spell it "ELSEIF".

Here is the course tax calculation, performed with the ELSIF instead of nested IF statements:

```
IF v_state = 'CA' THEN
  v_cost := v_retailprice * 1.08;
ELSIF v_state = 'NY' THEN
  v_cost := v_retailprice * 1.09;
ELSE
  v_cost := v_retailprice * 1.15;
END IF;
```

Or if you prefer:

```
IF v_state = 'CA' THEN
  v_cost := v_retailprice * 1.08;
ELSIF v_state = 'NY' THEN
  v_cost := v_retailprice * 1.09;
ELSIF v_state = 'ON' THEN
  v_cost := v_retailprice * 1.15;
END IF;
```

In the first of these two cases, it is assumed that the only choice other than "NY" or "CA" is "ON", so the ELSE can be used. The second example explicitly states each condition, and so it deals with the NULL condition differently. If v_state has a NULL value, the first implementation will have v_cost set to NULL, while the second would not perform any assignment of v_cost.

Many PL/SQL developers run into problems when they overlook the condition where a variable has a NULL value. This is probably because in most other programming languages, BOOLEAN datatypes have only two values — TRUE or FALSE. Always try to keep the NULL condition in mind and, if need be, test for it with the IS NULL or IS NOT NULL operator.

Nested Blocks

Objective

✦ Describe the rules of nested blocks

Wherever you can include an executable statement in a PL/SQL block of code, you are allowed to include a whole block of code. This nested block, or sub-block, can have its own declare, executable, and exception sections. There are several reasons for doing this, but the most common is for error-handling purposes. In the example in Figure 10-1, nested blocks are used in order to clarify which block of statements are to be executed in each of the alternatives of an IF statement. The two inner blocks have been shaded to distinguish them from the outer, or main block.

```
DECLARE                                                    Outer Block
 x NUMBER: = 1;
 v_odd_count NUMBER: = 0;
 v_odd_count NUMBER: = 0;
BEGIN
 WHILE x <= 10 LOOP
 IF MOD(x,2) = 0 THEN
  BEGIN                                                    Inner Block
   v_even_count: =v_even_count + 1;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(x) ||
   ' is an even number.');
  END;
ELSE
  BEGIN                                                    Inner Block
   v_odd_count: =v_odd_count + 1;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(x) ||
   ' is an odd number.');
  END;
END IF;
x: =x +1;
END LOOP;
DBMS_OUTPUT.PUT_LINE(
  ' There are ' || TO_CHAR(v_odd_count) ||
  ' odd numbers and ' || TO_CHAR(v_even_count) ||
  ' even numbers between 1 and 10.');
END;
```

Figure 10-1: An example of nested blocks

When run, the block outputs the following:

```
1 is an odd number.
2 is an even number.
3 is an odd number.
```

```
4 is an even number.
5 is an odd number.
6 is an even number.
7 is an odd number.
8 is an even number.
9 is an odd number.
10 is an even number.
There are 5 odd numbers and 5 even numbers between 1 and 10.

PL/SQL procedure successfully completed.
```

When you trace the program execution, you find that the code within the two inner blocks runs only when the condition is TRUE or FALSE accordingly. This program does not behave any differently when the BEGIN and END lines of sub-blocks are left out. In this case, the sub-blocks are merely present for readability. You can clearly see exactly which set of statements runs when the condition is TRUE and which when it is FALSE.

When you nest one block inside of another, the inner block is referred to as the *nested block* or *sub-block*. The sub-block may have its own declaresection, but this means that there must be rules of scope to decide where a variable can be referenced. The basic rule is that an identifier can be referenced anywhere within the block that it is declared, including any sub-blocks, but not outside of that block. The example in Figure 10-2 illustrates both legal and illegal references. Note that the variables x and y that are declared in the outer block can be used in the inner block, but the inner block variable z cannot be referenced outside of the inner block.

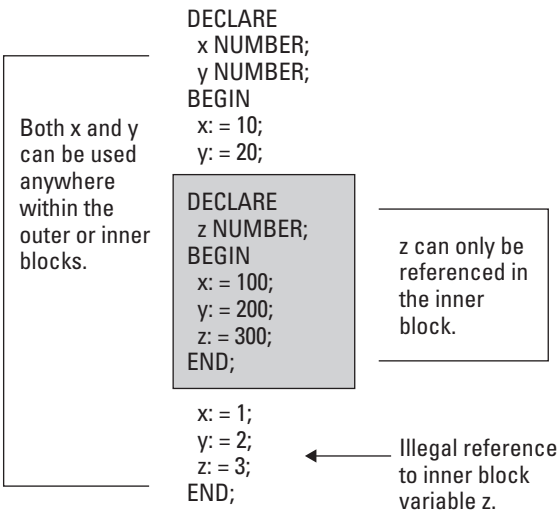


Figure 10-2: Nested blocks and variable scope

It is also possible to declare a variable in a nested block that has the same name as one in the outer block. Then any references to that variable name within the sub-block are to the inner block "copy" of the variable. You may use labels on the blocks and then begin the variable names with the appropriate block labels (outer or inner) in order to distinguish them, much like the nested loop counters seen earlier in this chapter. The best practice is to avoid this duplication of variable names in nested blocks. If unavoidable, remember that any changes made to outer block variables inside the sub-block are actually seen by those variables once the sub-block ends. The result from the example in Figure 10-3 may shed some light on this idea.

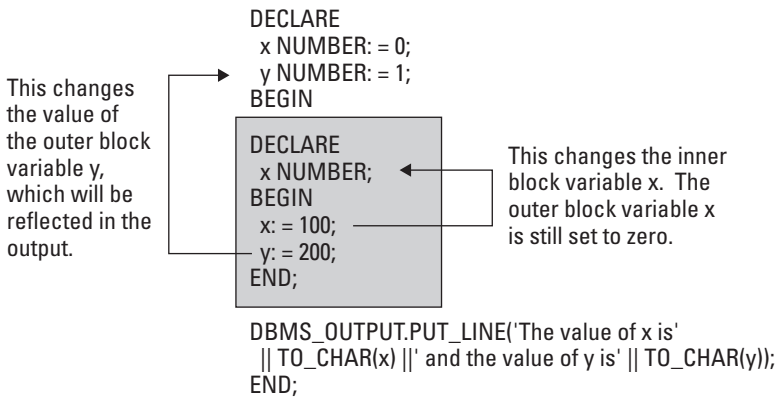


Figure 10-3: Nested blocks and variable assignment

The output line from the example follows:

```
The value of x is 0 and the value of y is 200

PL/SQL procedure successfully completed.
```

As stated earlier, it is possible to assign a value to the outer block "copy" of the variable x, but it involves labeling the blocks. It is best to avoid this situation with ambiguous variable names in sub-blocks.

Transaction Control

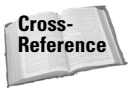
Objective

- ✦ Controlling transactions in PL/SQL

One of the fundamental ideas in the Oracle database is that of a *transaction*. A transaction is a set of one or more SQL statements that together form a consistent change to the database.

One of the classic examples of the use of transactions comes from the world of banking. One logical unit of work in banking is to transfer funds from one account to another. This can be represented by an update to one account to subtract the amount and an update to another to add it. Another table probably records all of the transfers, so a new row must be added to that table as well. This means that the simple transfer has now become two UPDATE statements and one INSERT. The idea of using transactions is that these three statements either should all occur or none should occur. If something goes wrong and the server is unable to perform all three statements, then what has already been done must be "backed out" automatically.

The SQL statement COMMIT is used to make changes permanent, while ROLLBACK is used to "back out," or reverse, any uncommitted changes. Both of these statements, along with the SAVEPOINT statement, are allowed in PL/SQL, within procedures, functions, and anonymous blocks.



For information on the use of the COMMIT, ROLLBACK, and SAVEPOINT statements, consult Chapter 5, "Adding, Updating, and Deleting Data."

A transaction starts when the first SQL statement is issued within a user session and ends when the COMMIT or ROLLBACK statement is issued, either implicitly or explicitly. Implicit commits are done by all DDL and DCL commands and normal termination of a SQL*Plus session, while implicit rollbacks usually occur when an abnormal termination of the session occurs. Otherwise, you must explicitly tell the server to make the changes permanent or to throw away the changes. This is true with statements entered from a command-line environment like SQL*Plus, and it is true of PL/SQL blocks.

Beginner PL/SQL programmers often assume that the statements made inside a block of code are automatically committed when it is executed. This is not necessarily true. The statements made inside the block can still be committed or rolled back after the block runs in SQL*Plus, while a Pro*C program must commit work — otherwise, it is rolled back.

Here is an example that illustrates an uncommitted change being rolled back from outside the block in SQL*Plus:

```
SQL> BEGIN
  2    INSERT INTO Courses(CourseNumber, CourseName,
  3                        RetailPrice)
  4    VALUES(500, 'Java Programming', 2500);
  5  END;
  6  /

PL/SQL procedure successfully completed.

SQL> SELECT CourseNumber, CourseName, RetailPrice
  2  FROM   Courses;
```

```
COURSENUMBER COURSENAME                          RETAILPRICE
------------ ----------------------------------- -----------
         100 Basic SQL                                  2000
         110 Advanced SQL                               2000
         201 Performance Tuning your Database           4000
         200 Database Performance Basics                4000
         210 Database Administration                    4500
         220 Backing up your database                   3000
         300 Basic PL/SQL                               2500
         310 Advanced PL/SQL                            2000
         320 Using your PL/SQLskills                    1750
         500 Java Programming                           2500

10 rows selected.

SQL> ROLLBACK;

Rollback complete.

SQL> SELECT CourseNumber, CourseName, RetailPrice
  2  FROM   Courses;

COURSENUMBER COURSENAME                          RETAILPRICE
------------ ----------------------------------- -----------
         100 Basic SQL                                  2000
         110 Advanced SQL                               2000
         201 Performance Tuning your Database           4000
         200 Database Performance Basics                4000
         210 Database Administration                    4500
         220 Backing up your database                   3000
         300 Basic PL/SQL                               2500
         310 Advanced PL/SQL                            2000
         320 Using your PL/SQLskills                    1750

9 rows selected.
```

When you make changes from within a PL/SQL block of code, you usually want to have them committed or rolled back from within the same block. The following example is a SQL*Plus script file with a PL/SQL block that takes a partial course name and new price for the course as input, then makes a decision to either commit or rollback an update, depending on how many rows are actually updated. It makes use of the implicit cursor attribute %ROWCOUNT, which returns the number of rows affected by the most recent SQL statement. The implicit cursor attributes are covered in Chapter 11.

```
ACCEPT coursename PROMPT "Enter the partial course name: "
ACCEPT newprice NUMBER PROMPT "Enter the new price: "
BEGIN
  UPDATE Courses
  SET    RetailPrice = &newprice
```

```
  WHERE  CourseName LIKE '%&coursename%';
  IF SQL%ROWCOUNT > 1 THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE(
    'More than one course includes '||'&coursename');
  ELSIF SQL%ROWCOUNT = 1 THEN
    COMMIT;
    DBMS_OUTPUT.PUT_LINE(
    'The price has been changed');
  ELSE
    DBMS_OUTPUT.PUT_LINE(
    'No course includes '||'&coursename');
  END IF;
END;
/
SELECT CourseNumber, CourseName, RetailPrice
FROM   Courses
/
```

If this SQL*Plus script is named ChangePrice, then here are a couple of executions of it:

```
SQL> start ChangePrice
Enter the partial course name: Tuning
Enter the new price: 3500
The price has been changed

PL/SQL procedure successfully completed.


COURSENUMBER COURSENAME                          RETAILPRICE
------------ ----------------------------------- -----------
         100 Basic SQL                                  2000
         110 Advanced SQL                               2000
         201 Performance Tuning your Database           3500
         200 Database Performance Basics                4000
         210 Database Administration                    4500
         220 Backing up your database                   3000
         300 Basic PL/SQL                               2500
         310 Advanced PL/SQL                            2000
         320 Using your PL/SQLskills                    1750

9 rows selected.

SQL> start ChangePrice
Enter the partial course name: PL/SQL
Enter the new price: 1500
More than one course includes PL/SQL

PL/SQL procedure successfully completed.
```

```
COURSENUMBER COURSENAME                          RETAILPRICE
------------ ----------------------------------- -----------
         100 Basic SQL                                  2000
         110 Advanced SQL                               2000
         201 Performance Tuning your Database           3500
         200 Database Performance Basics                4000
         210 Database Administration                    4500
         220 Backing up your database                   3000
         300 Basic PL/SQL                               2500
         310 Advanced PL/SQL                            2000
         320 Using your PL/SQLskills                    1750

9 rows selected.

SQL> start ChangePrice
Enter the partial course name: DBA
Enter the new price: 4000
No course includes DBA

PL/SQL procedure successfully completed.

COURSENUMBER COURSENAME                          RETAILPRICE
------------ ----------------------------------- -----------
         100 Basic SQL                                  2000
         110 Advanced SQL                               2000
         201 Performance Tuning your Database           3500
         200 Database Performance Basics                4000
         210 Database Administration                    4500
         220 Backing up your database                   3000
         300 Basic PL/SQL                               2500
         310 Advanced PL/SQL                            2000
         320 Using your PL/SQLskills                    1750

9 rows selected.
```

Note that the first test has only one course name that includes the word "Tuning", so it updates the RetailPrice from $4,000 to $3,500 and commits the change. In the second test, three course names contain "PL/SQL", so their prices appear unchanged. In fact, the prices were updated, then rolled back. In the final example, no course name contains "DBA", so the update affects no rows and is neither committed nor rolled back.

Finally, the following example uses the SAVEPOINT statement to include an intermediate point in the transaction that can be rolled back to, without losing all of the changes before that point.

```
BEGIN
  UPDATE ...
  INSERT ...
  DELETE ...
 SAVEPOINT no_update;
```

```
/* The next line could update more than one row, in which
caseit should be rolled back, but the previous changes should
     not be */

  UPDATE ...
  IF SQL%ROWCOUNT > 1 THEN
    ROLLBACK TO no_update;
  END IF;
  /* now if the update  affected more than one row, it is as if
that
     update was never done, so the first three SQL statements
     can still be either committed or rolled back */
  .
  .
  .
END;
```

Keep in mind that if you reuse the same savepoint name, it replaces (or erases) any earlier savepoints with the same name.

Key Point Summary

In PL/SQL, as in most programming languages, you need to be able to control the flow of statements. Some statements must run more than once, so you put them inside a loop. Other statements need not run at all, so you conditionally execute them in an IF statement.

- ✦ PL/SQL contains three different loop structures: the basic loop, FOR loop, and WHILE loop. The basic loop must contain an EXIT statement; otherwise, it continues to loop indefinitely. This EXIT is often conditional on the value of a variable that is incremented each time through the loop. A shortcut for using EXIT is to use the FOR loop, which has a built-in counter. The WHILE loop also has a built-in conditional termination, and the condition is checked at the beginning of each iteration of the loop.
- ✦ The IF statement in PL/SQL has several forms. The decision structure can be a simple decision to execute or skip a group of statements, based on a Boolean condition, or the structure may choose from several alternatives. The ELSE clause is provided to group the statements that are to be run when the IF condition is not met, while the ELSIF clause can be used to add additional conditions to check.
- ✦ Both loops and decision structures in PL/SQL can be nested within other loops or decision structures. This nesting must not be overlapping: When one structure begins and then another one is nested within the outer, the inner structure must end before the outer. In fact, the END LOOP statement always ends the innermost loop, and an END IF always ends the innermost IF statement.

- When one loop is nested within another and the inner loop contains an EXIT statement, the default functionality is to exit only the innermost loop. When you want to exit the outer loop, you must give the loops labels and specify the outer loop label in the EXIT statement.
- In the same way that one loop can be nested inside another, PL/SQL blocks can be nested. Each block can contain its own declare, executable, and exception sections. The inner block can reference outer block variables, but when it has a variable declared with the same name, the default functionality is to use the inner block variable. This can be overridden by qualifying variable names with the label associated with the outer block.
- The Oracle server notion of transactions still exists in PL/SQL, and one transaction can span across more than one PL/SQL block. Just because a block ends, it does not necessarily mean that the changes in the block have been committed. The SQL transaction control statements COMMIT, ROLLBACK, and SAVEPOINT are all valid statements in PL/SQL.

✦ ✦ ✦

STUDY GUIDE

The following questions can help you assess your understanding of the different loops and conditional processing structures available in PL/SQL. They also test your knowledge of the concepts of nesting and transaction controls.

Assessment Questions

1. Which of the following is not a valid structure in PL/SQL? (Choose the best answer.)

   A. IF ... THEN ... ELSE ... END IF;

   B. LOOP ... END LOOP;

   C. CASE ... WHEN ... THEN ... END CASE;

   D. WHILE ... LOOP ... END LOOP;

   E. FOR i IN lower .. upper LOOP ... END LOOP;

2. Which of the following statements about the FOR loop in PL/SQL are true? (Choose three responses.)

   A. The loop has an index that is implicitly declared.

   B. The statements within the loop must execute at least once.

   C. The index can count by a value other than 1 when the STEP option is added.

   D. The index of the loop is available to be referenced only inside the loop.

   E. The index value cannot be changed within the loop using an assignment statement.

3. Evaluate this PL/SQL block:

```
DELCARE
  v_over_booked BOOLEAN;
BEGIN
  INSERT INTO ClassEnrollment(ClassID, StudentNumber,
         Status, EnrollmentDate, Price)
  VALUES(51, 1001, 'Confirmed', SYSDATE, 4000);
  IF v_over_booked THEN
    ROLLBACK;
  ELSE
    COMMIT;
  END IF;
END;
```

After this block executes, what is true of the change to the database? (Choose the best answer.)

A. The row is not visible in the table because it has not yet been committed.

B. The row is not visible in the table because it was inserted and then the change was rolled back.

C. The change is visible in the table but not yet committed.

D. The change is visible in the table and committed.

4. Consider the following structure:

```
IF x > 1000 THEN
  Statement 1;
ELSIF x > 500 THEN
  Statement 2;
ELSIF x BETWEEN 1 AND 1000 THEN
  Statement 3;
ELSE
  Statement 4;
END IF;
```

If the value of x is 750, then which of the numbered statement(s) will execute? (Choose the best answer.)

A. Statement 4 only.

B. Statement 2 only.

C. Statements 2 and 3.

D. None of the above.

5. Evaluate this PL/SQL block:

```
 1  DECLARE
 2     v_counter INTEGER := 1;
 3  BEGIN
 4     WHILE v_counter <= 10 LOOP
 5       IF MOD(v_counter, 2) = 0
 6        THEN DBMS_OUTPUT.PUT_LINE(
 7      TO_CHAR(v_counter)||' is an even number.');
 8       v_counter = v_counter + 1;
 9  END WHILE;
10     DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_counter));
11  END;
```

This block of code has several syntax errors. Choose all that apply:

A. Line 4 must end with a semicolon (;).

B. The IF statement is not allowed within a WHILE loop.

C. Line 9 should be END LOOP;

D. There must be an END IF before the loop ends.

E. The variable v_counter cannot be used outside of the WHILE LOOP.

6. Given this PL/SQL block:

```
DECLARE
  v_counter INTEGER := 1;
  v_upper INTEGER := 10;
BEGIN
  WHILE v_counter <= v_upper LOOP
     INSERT INTO test(results)
     VALUES(v_counter);
     IF v_counter = 5 THEN
     v_upper := 7;
     END IF;
     v_counter := v_counter + 2;
  END LOOP;
END;
```

How many rows will be inserted into the test table? (Choose the best answer.)

A. 10

B. 7

C. 5

D. 3

E. 4

7. Evaluate this PL/SQL block:

```
BEGIN
  INSERT INTO Courses(CourseNumber, CourseName,
            RetailPrice)
  VALUES(600,'Introduction to Unix', 900);
  SAVEPOINT itu;

  INSERT INTO Courses(CourseNumber, CourseName,
            RetailPrice)
  VALUES(610,'Shell Programming', 1050);
  SAVEPOINT shell;

  INSERT INTO Courses(CourseNumber, CourseName,
            RetailPrice)
  VALUES(620,'System Administration I', 2000);
  SAVEPOINT sysadmin;
```

```
  INSERT INTO Courses(CourseNumber, CourseName,
            RetailPrice)
  VALUES(630,'System Administration II', 2250);

  ROLLBACK TO shell;
  COMMIT;
END;
```

Which of the following new CourseNumbers will be committed? (Choose all that apply.)

A. 600

B. 610

C. 620

D. 630

8. Given this PL/SQL block:

```
DECLARE
  v_lower INTEGER := 1;
  v_upper INTEGER := 100;
BEGIN
  FOR i IN v_lower .. v_upper LOOP
       INSERT INTO test(results)
       VALUES(i);
       IF i = 50 THEN
         v_upper := 70;
       END IF;
  END LOOP;
END;
```

How many rows will be inserted into the test table? (Choose the best answer.)

A. 50

B. 70

C. 0

D. 100

E. 1

9. Evaluate this PL/SQL block:

```
BEGIN
  INSERT INTO Courses(CourseNumber, CourseName,
            RetailPrice)
  VALUES(600,'Introduction to Unix', 900);
  SAVEPOINT itu;

  INSERT INTO Courses(CourseNumber, CourseName,
            RetailPrice)
  VALUES(610,'Shell Programming', 1050);
  SAVEPOINT shell;
  ROLLBACK;
```

```
  INSERT INTO Courses(CourseNumber, CourseName,
            RetailPrice)
  VALUES(620,'System Administration I', 2000);
  SAVEPOINT sysadmin;

  INSERT INTO Courses(CourseNumber, CourseName,
            RetailPrice)
  VALUES(630,'System Administration II', 2250);
  SAVEPOINT sysadmin;
  ROLLBACK TO sysadmin;
  COMMIT;
END;
```

Which of the following new CourseNumbers will be committed? (Choose all that apply.)

A. 600

B. 610

C. 620

D. 630

10. Evaluate this PL/SQL loop:

```
BEGIN
  FOR i IN 1..12 LOOP  -- month loop
    FOR j IN 1..10 LOOP
      EXIT WHEN TO_CHAR(SYSDATE,'MM') = i;
      total_month_ord(i, j);   -- procedure call
    END LOOP;
  END LOOP;
END;
```

Assume that total_month_ord is a procedure that takes two arguments, month number and order number (and order numbers start every month at 1). This block is supposed to call the procedure for the first ten orders of each month up to, and including, the current month. What, if anything, can be changed to make the block accomplish this task? (Choose 1 or more responses.)

A. Change the EXIT WHEN condition from i to i + 1.

B. Move the EXIT WHEN statement after the inner loop ends.

C. Add a label to the outer loop and then exit the outer loop using the label.

D. Both A and C.

E. Change nothing; it already works.

Scenarios

1. You are an application developer for Luxury Cruise Lines, Inc. You have to build a booking application that takes the following information from a travel agent: passenger information, cruise number, cruise date, and cabin preference. When the booking can be made, a deposit is taken from the passenger, the spot is held, and the travel agent is paid a commission. You have a number of PL/SQL procedures that accomplish various pieces of the functionality — for example, CreatePassenger, CreateBooking, TakeDeposit, and PayCommission. How can you set up this application to ensure that you complete all of these steps only when there is room on the particular cruise that the passenger wants?
2. Your company purchased an application that runs with an Oracle database. From time to time, you are called upon to create ad hoc queries in SQL*Plus. Now your manager wants you to start building applications for entering and retrieving data, still with a SQL*Plus interface. Given that SQL has no support for loops and conditional processing, explain why you might want to build at least part of the application in PL/SQL.

Lab Exercise

Lab 10–1 Using loops and conditional processing to print the multiples of an integer

1. Sign on to SQL*Plus as user Student with password oracle.
2. Create a SQL*Plus script file called PrintMultiples. Create three ACCEPT statements, to retrieve integers named x, y, and n. This program is going to print to the screen the first n multiples of the number x that are greater than the value y.
3. Create a PL/SQL block of code with a declare section, which declares a counter called v_mult_count. Initialize v_mult_count to be 1. Also, declare a second integer called v_test_num and initialize it to be y+1. The program tests all numbers greater than y to see if they are divisible by the integer x.
4. In the executable section of the block, make a WHILE loop that loops n times. Use the variable v_mult_count as the counter.
5. Within the loop, if the value of the variable v_test_num is divisible by x, print its value to the screen using DBMS_OUTPUT.PUT_LINE and then increment the variable v_mult_count. Make sure that you have set ON the SQL*Plus SERVEROUTPUT environment variable in the current session. Hint: use the MOD function to determine divisibility.

6. Be sure to increment the variable v_test_num before ending the loop.
7. Save your script file and then execute it using the START command in SQL*Plus. Try a variety of values, including zero and negative values for the parameters.

Answers to Chapter Questions

Chapter Pre-Test

1. There are three types of loops in PL/SQL — the basic loop that starts LOOP ... END LOOP, the FOR loop, and the WHILE loop.
2. A FOR loop is used when you have a set number of iterations to perform. Because it has a built-in index and boundary checking, it is much simpler to use in those situations.
3. The default functionality of the EXIT statement is to exit only the innermost loop in which it is contained. However, if a label is given in the EXIT statement, it exits all loops up to and including the level of the loop that has that label.
4. Oracle places no limit on the number of different conditions that can be tested with the IF statement. Each condition must be a Boolean and can be a complex expression involving AND, OR, and NOT. There can also be as many alternative conditions as you want by including ELSIF clauses.
5. Labels are used to identify an executable line of code, loop, or even a PL/SQL block. When they are used to identify a line of code, you can unconditionally jump to that line using the GOTO statement. When used to identify a loop, they can be included in the EXIT condition to specify which loop to exit. Finally, when used to identify a block, they can be used to qualify variable names that are reused in a sub-block. Labels can also be used just for readability — name the loop, and then label the corresponding END LOOP.
6. You can nest one block inside another wherever an executable line of code is allowed. This is done for several reasons, including readability and error handling. The readability is enhanced because you can group several lines of code together with a BEGIN and END around them to show that they execute together. In error handling, a sub-block can have its own exception section that tailors error handling to the lines of code only within the sub-block.
7. The COMMIT, ROLLBACK, and SAVEPOINT statements, along with the SET TRANSACTION statement, are allowed within a PL/SQL block and are used to control transactions.

8. After a transaction is started by an SQL statement, you can include the executable line:

   ```
   SAVEPOINT name;
   ```

   Where name is the name of the intermediate point. Then other Data Manipulation Language (DML) statements can occur and if the following statement is issued:

   ```
   ROLLBACK TO name;
   ```

   then only the DML statements that have occurred after the savepoint was issued are rolled back.

9. The ELSIF clause checks an alternative condition to the one in the IF condition THEN ... END IF; If the preceding condition is not TRUE, it goes to the first ELSIF and tests that condition, which has its own set of statements to run if TRUE. There can be as many ELSIF clauses as you need to test all of the possible conditions, but at most, one of them actually runs its corresponding statements. The first condition that returns TRUE runs its statements, and then control skips to the next line after the END IF.

10. The WHILE loop has a built-in exit condition, while the basic loop does not. The latter executes the statements only inside when the condition evaluates to TRUE. Therefore, the basic loop may not even enter the loop if the condition is not initially met. The basic loop always executes at least once, but the first statement inside the loop could be an EXIT statement that checks a condition. When this is the case, the two loops are logically equivalent but with a different syntax.

Assessment Questions

1. **C** — There is no case structure in PL/SQL. If you want to test multiple cases, use an IF statement with one or more ELSIF clauses. Refer to the "Conditional Processing" section, earlier in this chapter.

2. **A, D, E** — The FOR loop implicitly declares an index that increments by 1 or –1 each iteration. You cannot make it step by another value, and you cannot change its value. When the loop ends, the index is no longer available. The statements within the loop do not execute even once when the lower bound is initially higher than the upper bound. Refer to the "FOR Loop" section, earlier in this chapter.

3. **D** — Because the Boolean variable is not given a value, it has a NULL value. After the row is inserted, the condition of the IF statement is not be met, so the ELSE statement (COMMIT) is executed. Then the change is committed and is visible to everyone.

4. **B** — With a value of 750, the first condition is not met, so it continues on to the second, which is met. Therefore, statement 2 runs, and then control goes to the next line of code after the END IF.

5. **C, D** — IF statements can be nested inside loops, but they must end before the loop does. The loop ends with the END LOOP; statement, but no semicolon is used at the beginning of the loop. Only the FOR loop implicitly declares an index that can be referenced only inside the loop — in this example, the scope of v_counter is the entire block in which it is declared. Refer to the "WHILE Loop" and "Conditional Processing" sections, earlier in this chapter.
6. **E** — In the WHILE loop, the exit condition is checked at the beginning of every iteration. During each iteration, the counter is incremented by 2, so instead of running ten times, the loop runs five times, except that the upper bound changes on the third iteration (when v_counter = 5) to 7. Hence, the values inserted are 1, 3, 5, and 7. Refer to the "Loops" section, earlier in this chapter.
7. **A, B** — The ROLLBACK TO statement allows a transaction to remove some, but not all, changes made by sending it back to a marker. In this case, it rolls back to the point just after the first two INSERT statements and then commits these two. Refer to the "Transaction Control" section, earlier in this chapter.
8. **D** — The FOR loop takes the value of the bounds when it first encounters the loop and does not reflect any changes in them. Therefore, this loop runs from 1 to 100 because the line that tries to assign a value to the upper bound does not affect the loop. Refer to the "FOR Loop" section, earlier in this chapter.
9. **C, D** — With the first ROLLBACK, the first two rows inserted are taken out. Then the second ROLLBACK rolls back only to the last savepoint named "sysadmin". In fact, this rolls back no inserts because the same savepoint name is reused in the line immediately before. Refer to the "Transaction Control" section, earlier in this chapter.
10. **B, D** — The way the block currently is written, the inner loop gets exited when it is the current month, so all months except the current one have orders processed. By moving the EXIT WHEN to after the inner loop ends, the outer loop exits when the current month's orders have been processed. An alternative to this block is to leave the EXIT WHEN in the loop but exit before it processes the month following the current one's orders. Hence the i + 1. However, this change alone does not solve the problem, because you need the program to exit the outer loop at that point. Thus, the label is necessary on the outer loop. Refer to the section "Nested Loops and Labels," earlier in this chapter.

Scenarios

1. Because this is an "all or nothing" kind of situation, it should be done in one transaction. Remember that one transaction can span across more than one PL/SQL block, so it can be started in the main block, which calls all of the procedures mentioned. That way, you can test the success of each of the steps, and when something goes wrong (for example, no room is left on the cruise), then a ROLLBACK occurs, and the passenger information is never entered, and so on.

2. Without using PL/SQL, all looping and decision making must be made by the user of the application. For example, if a decision is based on the results of a query, then the SQL*Plus application displays the results of the SELECT statement on the screen, and the user has to initiate the next move, based on his or her decision. In a PL/SQL program, the results of the query are held in variables, and the decision logic can be included in IF statements. Also, looping enables you to perform a set of tasks multiple times, instead of forcing the user to run a script a number of times.

Lab Exercise

Lab 10–1 Using loops and conditional processing to print the multiples of an integer

The entire script file should look something like this:

```
SET SERVEROUTPUT ON
ACCEPT x NUMBER PROMPT "Enter any positive integer: "
ACCEPT y NUMBER PROMPT "Show multiples greater than: "
ACCEPT n NUMBER PROMPT "Number of multiples to print: "
DECLARE
  v_mult_count INTEGER := 1;
  v_test_num INTEGER := &y + 1;
BEGIN
  WHILE v_mult_count <= &n LOOP
    IF MOD(v_test_num, &x) = 0 THEN
      DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_test_num)
           ||' is a multiple of '||TO_CHAR(&x));
      v_mult_count := v_mult_count + 1;
    END IF;
    v_test_num := v_test_num + 1;
  END LOOP;
END;
/
```

Remember that in order for printing in SQL*Plus, the environment variable SERVEROUTPUT must be set ON. Here is a sample run:

```
SQL> START PrintMultiples
Enter any positive integer: 7
Show multiples greater than: 40
Enter the number of multiples to print: 5
42 is a multiple of 7
49 is a multiple of 7
56 is a multiple of 7
63 is a multiple of 7
70 is a multiple of 7

PL/SQL procedure successfully completed.
```

CHAPTER 11

Interacting with the Database Using PL/SQL

EXAM OBJECTIVES

- Interacting with the Oracle Server
  - Write a successful SELECT statement in PL/SQL
  - Write DML statements in PL/SQL
  - Determine the outcome of SQL DML statements
- Working with Composite Datatypes
  - Create user-defined PL/SQL records
  - Create a PL/SQL table
  - Create a PL/SQL table of records
  - Describe the difference between records, tables, and tables of records
- Writing Explicit Cursors
  - Distinguish between an implicit and an explicit cursor
  - Use a PL/SQL record variable
  - Write a cursor FOR loop
- Advanced Explicit Cursor Concepts
  - Write a cursor that uses parameters
  - Determine when a FOR UPDATE clause in a cursor is required
  - Determine when to use the WHERE CURRENT OF clause
  - Write a cursor that uses a subquery

CHAPTER PRE-TEST

1. What clause must be added to the SELECT statement in PL/SQL to specify where to store the values returned?
2. Which cursor attributes enable you to check whether an UPDATE statement updated any records?
3. What do you use to select multiple records from the database?
4. What is the effect of adding the FOR UPDATE clause to a cursor declaration?
5. Which cursor attributes can you use to determine if all rows have been fetched from an explicit cursor?
6. What is a PL/SQL table?
7. What is a cursor parameter?
8. What are the advantages to using a cursor FOR loop?
9. In what section of the PL/SQL program do you write the SELECT statement that populates an explicit cursor?
10. Do you get an error if a SELECT statement that is not part of an explicit cursor returns no rows?

✦ Answers to these questions can be found at the end of the chapter. ✦

PL/SQL is a programming language designed to interact with the Oracle database. This chapter explains how to access information stored in the database from within a PL/SQL program. This chapter introduces the SELECT INTO statement, which enables you to select a single row of data from a database table. It also introduces explicit cursors, which you can use to select multiple rows of data from the database. Explicit cursors have additional features such as the cursor FOR loop, the FOR UPDATE clause, the WHERE CURRENT OF clause, and cursor parameters. This chapter also covers using the INSERT, UPDATE, and DELETE statements to modify the data stored in database tables.

When accessing the database in a program, you often want to retrieve multiple rows or multiple columns from the database. To simplify storage of multiple values, this chapter also explains how to use the composite datatypes records, PL/SQL tables, PL/SQL tables of records, and VARRAYs.

SQL Statements

The PL/SQL language is used extensively to access the data contained in the database. SELECT, INSERT, UPDATE, and DELETE statements can be written directly into PL/SQL programs. PL/SQL requires a slightly different syntax from SQL*Plus for some SQL commands and may return errors from SQL statements that would run successfully in SQL*Plus. It is important to understand these differences so you can successfully use SQL statements in your PL/SQL programs.

You can issue SELECT statements and Data Manipulation Language (DML) statements such as INSERT, UPDATE, and DELETE from PL/SQL code. You can also issue SAVEPOINT, COMMIT, and ROLLBACK commands from PL/SQL code. You cannot issue Data Definition Language (DDL) statements such as CREATE and DROP or Data Control Language (DCL) statements such as GRANT and REVOKE directly from PL/SQL code without using either the EXECUTE IMMEDIATE statement or the DBMS_SQL package.

SELECT

SELECT statements can be issued as implicit or explicit cursors. This section covers the SELECT statement as an implicit cursor. Later in this chapter, the section "Explicit Cursors" covers using the SELECT statement as an explicit cursor. When you use a SELECT statement as an implicit cursor, Oracle expects exactly one row to be returned by the SELECT statment. If the SELECT statement returns more than one row, or no rows, Oracle raises an exception. When programming with implicit cursors, you should use the primary key whenever possible in the WHERE clause to reduce the chance of returning multiple rows and raising an exception. If you wish to return multiple rows from a SELECT statement, then you should use an explicit cursor.

You can write simple or complex SELECT statements in your PL/SQL code. You must add an INTO clause to your SELECT statement to accept the values returned by the SELECT. The INTO is placed between the SELECT and the FROM clause. The INTO clause must list variables to hold each of the values returned by the SELECT statement. The variables listed in the INTO clause can be PL/SQL variables declared in the DECLARE section or bind variables.

```
ACCEPT p_student NUMBER PROMPT "Enter Student Number: "
DECLARE
   v_lastName VARCHAR2(30);
   v_firstName VARCHAR2(30);
BEGIN
   SELECT FirstName, LastName
     INTO v_firstName, v_lastName
     FROM students
   WHERE studentNumber = &p_student;

   DBMS_OUTPUT.PUT_LINE('Student Name is '||
             v_firstname||' '||v_lastname);
END;
/
```

This program produces the following output:

```
Enter Student Number: 1000
Student Name is John Smith
```

Because the variables being declared are used to store values derived from database columns, you may want to declare the variables using %TYPE. By using %TYPE, you reduce the chance of having to make changes to your code when changes are made to the structure of the database.

You can write SELECT statements using table joins, single-row functions, group functions, and subqueries in PL/SQL programs as long as you add the INTO clause with variables that can hold the values returned by SELECT.

There are some restrictions on what you can do with SELECT statements in PL/SQL code.

The values returned by the SELECT statement are being written to variables that can hold only one value; therefore, Oracle returns the exception, TOO_MANY_ROWS, if your SELECT statement returns more than one row. When you want to fetch more than one row from the database table, you must use explicit cursors, which are covered later in this chapter.

ORDER BY clauses may not be used in SELECT statements within PL/SQL because you are not allowed to return multiple rows from a SELECT statement. SELECT statements must return exactly one row; Oracle returns the exception

NO_DATA_FOUND when a SELECT statement returns no rows. You learn how to handle the NO_DATA_FOUND and TOO_MANY_ROWS exceptions in Chapter 12, "Handling Errors and Exceptions in PL/SQL."

```
DECLARE
   v_lastName students.lastname%TYPE;
   v_firstName students.firstname%TYPE;
BEGIN
   SELECT FirstName, LastName
     INTO v_firstName, v_lastName
     FROM students
   WHERE studentNumber = 99999;

   DBMS_OUTPUT.PUT_LINE('Student Name is '||
             v_firstname||' '||v_lastname);
END;
/
```

This program returns the error message:

```
ORA-01403: no data found
```

Oracle returns an error when a SELECT statement returns more than one row.

```
DECLARE
   v_lastName students.lastname%TYPE;
   v_firstName students.firstname%TYPE;
BEGIN
   SELECT FirstName, LastName
     INTO v_firstName, v_lastName
     FROM students;

   DBMS_OUTPUT.PUT_LINE('Student Name is '||
             v_firstname||' '||v_lastname);
END;
/
```

This program returns the error message:

```
ORA-01422: exact fetch returns more than requested number of
rows
```

When executing SQL statements within PL/SQL code, it is vital that your variable names not be the same as the database column names. Within a WHERE clause, Oracle cannot distinguish between variable names and column names. If a variable used in a SQL statement has the same name as a database column, Oracle assumes that you are referring to the database column of the same name. This can cause exceptions and unexpected results.

```
DECLARE
   lastname students.lastname%TYPE;
   firstname students.firstname%TYPE;
   studentnumber students.studentnumber%TYPE := 1000;
BEGIN
   SELECT firstname, lastname
     INTO firstname, v_lastname
     FROM students
   WHERE studentnumber = studentnumber;

   DBMS_OUTPUT.PUT_LINE('Student Name is '||
               firstname||' '||lastname);
END;
/
```

This program returns the error message:

```
ORA-01422: exact fetch returns more than requested number of
rows
```

INSERT

The INSERT statement can be used in a PL/SQL program to add rows to a table. The syntax of the INSERT statement is the same in PL/SQL as it is in SQL*Plus. Any rows inserted using the INSERT statement in a PL/SQL program are not permanent until a COMMIT statement is issued. You must include COMMIT and ROLLBACK statements directly in your PL/SQL code. Oracle does not perform an automatic commit or rollback when a PL/SQL program finishes execution, so it is a good programming habit to include COMMIT and ROLLBACK statements in your code. You should COMMIT or ROLLBACK at the end of each transaction. If you omit the COMMIT and ROLLBACK statements, the status of your transaction depends on the actions taken after the program is completed. If you issue a DCL command, DDL command, or COMMIT statement after the program completes, the transaction is saved. If you issue a ROLLBACK, the transaction is rolled back.

```
ACCEPT p_coursenumber NUMBER PROMPT "Enter course number: "
ACCEPT p_name CHAR PROMPT "Enter course name: "
ACCEPT p_price NUMBER PROMPT "Enter price: "
ACCEPT p_description CHAR PROMPT "Enter description: "

BEGIN
   INSERT INTO courses (coursenumber, coursename,
   replacescourse, retailprice, description)
   VALUES (&p_coursenumber, '&p_name',
   null , &p_price, '&p_description');

COMMIT;
END;
/
```

This program produces the following output:

```
Enter course number: 12345
Enter course name: mycourse
Enter price: 500
Enter description: mydescription

SQL> SELECT *
  2  FROM courses
  3  WHERE coursenumber = 12345;

COURSENUMBER COURSENAME           REPLACESCOURSE RETAILPRICE
------------ -------------------- -------------- -----------
DESCRIPTION
------------------------------------------------------------
       12345 mycourse                                    500
mydescription
```

You do not have to specify the list of columns in an INSERT statement if you provide values for all columns in the table. When you list the columns in your INSERT statement, you reduce code maintenance. By listing the columns, you may not have to change your code when a column is added to the table or the order of the columns on the table changes.

You can use variables such as SYSDATE and USER in your INSERT statements.

```
BEGIN
   INSERT INTO classenrollment (classid, studentNumber,
   status, enrollmentdate, price, grade, comments)
   VALUES (53, 1002, 'Hold', SYSDATE, 1500, null, null);
END;
/

SQL> SELECT *
  2  FROM classenrollment
  3  WHERE classid = 53
  4  AND studentnumber = 1002;

CLASSID    STUDENTNUMBER STATUS     ENROLLMENTDATE
---------- ------------- ---------- -----------------------
PRICE      GRAD COMMENTS
---------- ---- ----------------------------------------
        53          1002 Hold       12-FEB-01
1500
```

Sequences can also be accessed directly from an INSERT statement in a PL/SQL program. The following example uses a sequence called classid_seq, which would have been created by the DBA to populate the classid column. More information on sequences can be found in Chapter 7, “Creating and Managing Oracle Database Objects.”

```
BEGIN
   INSERT INTO scheduledclasses (classid, coursenumber,
   locationid, classroomnumber, instructorid, startdate,
   daysduration, status, comments)
   VALUES (classid_seq.NEXTVAL, 300,
   300, 1, 100, '01-JAN-2001',
   4, 'Hold',null);
END;
/

SQL> SELECT *
  2  FROM scheduledclasses
  3  WHERE coursenumber=300
  4  AND locationid=300;

CLASSID COURSENUMBER LOCATIONID CLASSROOMNUMBER INSTRUCTORID
---------- ------------ ---------- --------------- ---------
STARTDATE                     DAYSDURATION STATUS
----------------------------- ------------ ----------
COMMENTS
----------------------------------------------------------------
        54          300        300               1          100
01-JAN-01                                  4 Hold
```

UPDATE

The UPDATE statement can be used within PL/SQL programs to update a row or a set of rows with a new value. The syntax for the UPDATE statement is the same in PL/SQL programs as in SQL*Plus.

```
ACCEPT p_coursenumber NUMBER PROMPT "Enter course number: "
ACCEPT p_newprice NUMBER PROMPT "Enter new course price: "
BEGIN
   UPDATE courses
   SET retailprice = &p_newprice
   WHERE coursenumber = &p_coursenumber;
END;
/
```

This program produces the following output:

```
Enter course number: 12345
Enter new course price: 1000

SQL> SELECT *
  2  FROM courses
  3  WHERE coursenumber = 12345;
```

```
COURSENUMBER COURSENAME           REPLACESCOURSE RETAILPRICE
------------ -------------------- -------------- -----------
DESCRIPTION
------------------------------------------------------------
       12345 mycourse                                   1000
mydescription
```

DELETE

The DELETE statement can be used within PL/SQL programs to delete a row or a set of rows. The syntax for the DELETE statement is the same in PL/SQL programs as in SQL*Plus.

```
ACCEPT p_coursenumber NUMBER PROMPT "Enter course number: "
BEGIN
   DELETE FROM courses
   WHERE coursenumber = &p_coursenumber;
END;
/
```

This program produces the following output:

```
Enter course number: 12345

SQL> SELECT *
  2  FROM courses
  3  WHERE coursenumber = 12345;

no rows selected
```

Implicit cursors

Whenever a SQL statement is executed in a PL/SQL program, a memory area is created to parse the SQL statement; this area is called a *cursor*. This is referred to as an *implicit cursor* because it is controlled by Oracle. Explicit cursors are covered in the section "Explicit Cursors." The implicit cursor used for SQL statements is created, populated, and deleted by Oracle. Cursors have a set of attributes that return information about the SQL statement. These attributes can be accessed in PL/SQL code. The four implicit cursor attributes are:

- ✦ SQL%FOUND returns TRUE if SQL statement found one or more records.
- ✦ SQL%NOTFOUND returns TRUE if SQL statement found no records.
- ✦ SQL%ROWCOUNT returns number of rows affected by SQL statement.
- ✦ SQL%ISOPEN always returns FALSE, indicating that the implicit cursor has been closed.

When you execute a DELETE or UPDATE statement in SQL*Plus, SQL*Plus returns a message telling you how many rows were updated or deleted. In PL/SQL, you do not receive a message, so you must examine the implicit cursor attributes after the statement is executed to find out if any rows were updated or deleted.

```
ACCEPT p_coursenumber NUMBER PROMPT "Enter course number: "
ACCEPT p_newprice NUMBER PROMPT "Enter new course price: "
BEGIN
   UPDATE courses
      SET retailprice = &p_newprice
   WHERE coursenumber = &p_coursenumber;

IF SQL%NOTFOUND THEN
     DBMS_OUTPUT.PUT_LINE('No rows updated. '
              ||&p_coursenumber||' does not exist');
   END IF;
END;
/
```

This program produces the following output:

```
Enter course number: 12345
Enter new course price: 2000

No rows updated. 12345 does not exist

ACCEPT p_classid NUMBER PROMPT "Enter class id: "
BEGIN
   DELETE FROM ClassEnrollment
    WHERE classid = &p_classid;

   DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT||' rows deleted.');
END;
/
```

This program produces the following output:

```
Enter class id: 12345
0 rows deleted.
```

The implicit cursor attributes should not be used to determine the number of rows returned by a SELECT statement. In the following example, the programmer has coded an IF statement to check the status of SQL%NOTFOUND after executing the SELECT statement. This program will not work as intended when no rows are selected.

```
ACCEPT p_coursenumber NUMBER PROMPT "Enter course number: "
ACCEPT p_newprice NUMBER PROMPT "Enter new course price: "
```

```
BEGIN
   SELECT retailprice
     INTO v_retailprice
     FROM courses
   WHERE coursenumber = &p_coursenumber;

   IF SQL%NOTFOUND THEN
        DBMS_OUTPUT.PUT_LINE('No rows selected. '
              ||&p_coursenumber||' does not exist');
END IF;
END;
/
```

When the SELECT statement returns no rows, this program returns the error message:

```
ORA-01403: no data found
```

When the SELECT statement returns no rows, an exception is raised that causes the program to immediately go to the exception handling section or, if there is no exception handling, causes the program to exit with an unhandled error. The IF statement is never executed. If you want to determine if a SELECT statement returns any rows, you should either use explicit cursors and explicit cursor attributes, or use implicit cursors with exception handling, which is covered in Chapter 12, "Handling Errors and Exceptions in PL/SQL."

Explicit Cursors

If you want to write a program that selects multiple rows from a database table, you have to use explicit cursors because the SELECT INTO statement generates an error when it selects more than one row. The set of rows returned by a multiple row query is called an *active set*. When you use an explicit cursor, you are creating a pointer to keep track of the next row in the active set to be fetched. There are several steps involved in cursor processing. You declare the cursor. You use the OPEN command to create theactive set, and populate it with values.You use the FETCH command to fetch the values one row at a time. When you have processed all the rows, you use the CLOSE command to close the cursor.

Declaring explicit cursors

To create the cursor, you declare it in the declaration section of the program. In the cursor declaration, you write the SELECT statement that will return the desired record or records. You do not list the INTO clause in the cursor declaration.

The syntax of the cursor declaration is the following:

```
CURSOR cursorname IS select statement;
```

The SELECT statement used in a cursor declaration may return multiple rows, so you can add an ORDER BY clause to the SELECT statement if you want to return the rows in a particular order.

The following example declares a cursor to fetch a list of students enrolled in a particular class:

```
ACCEPT p_classid NUMBER PROMPT "Enter Class id: "
DECLARE
   /* Declaration of cursor to fetch a list of students
   enrolled in a particular class and their status */

   CURSOR student_enrolled IS
   SELECT studentnumber, status
     FROM classenrollment
    WHERE classid = &p_classid
    ORDER BY studentnumber;
BEGIN
...
END;
/
```

Opening explicit cursors

Once the cursor is declared, you populate it using the OPEN command. The OPEN command goes in the executable portion of the program. The OPEN command allocates memory, executes the SELECT statement, populates the active set with the row or rows selected, and sets the pointer to point before the first row.

The syntax of the OPEN command is the following:

```
OPEN cursorname;
```

The following example opens the student_enrolled cursor created in our previous example:

```
ACCEPT p_classid NUMBER PROMPT "Enter Class id: "
DECLARE
   CURSOR student_enrolled IS
   SELECT studentnumber, status
     FROM classenrollment
    WHERE classid = &p_classid
    ORDER BY studentnumber;
```

```
BEGIN
   /* Use the OPEN command to execute the SELECT statement
   and populate the cursor with the values returned */

   OPEN student_enrolled;
   ...
END;
/
```

Closing an explicit cursor

After you have fetched all the rows needed from the cursor, you close it using the CLOSE command. The CLOSE command disables the cursor and undefines the active set. If you try to FETCH from a cursor after it has been closed, the exception INVALID_CURSOR is raised.

The syntax for the CLOSE command is the following:

```
CLOSE cursorname;
```

The following example closes the cursor from our previous example:

```
ACCEPT p_classid NUMBER PROMPT "Enter Class id: "
DECLARE
   CURSOR student_enrolled IS
   SELECT studentnumber, status
     FROM classenrollment
    WHERE classid = &p_classid
    ORDER BY studentnumber;

v_studentnumber classenrollment.studentnumber%TYPE;
v_status        classenrollment.status%TYPE;
BEGIN

   OPEN student_enrolled;

/* Code will be added here to fetch and process the
   rows in the cursor */

   /* After processing all the rows in the cursor, we close
   the cursor to free up the memory */

   CLOSE student_enrolled;
END;
/
```

Fetching from explicit cursors

In order to access the values in the active set, you must use the FETCH command. The first FETCH command fetches the first row contained in the active set. The second FETCH command returns the second row in the active set, the third FETCH returns the third row, and so on. After each FETCH, the pointer in the active set advances to the next row. When you use the FETCH command, you must use the INTO clause to specify which variables should be populated with the values returned from the cursor. Your INTO clause should have one variable for each column selected in the cursor declaration. The variables must be listed in the same order as the columns in the cursor declaration.

The syntax for the FETCH command is the following:

```
FETCH cursorname
INTO variable1, variable2, ... ;
```

The following example fetches a row from the cursor student_enrolled, created in our previous example:

```
ACCEPT p_classid NUMBER PROMPT "Enter Class id: "
DECLARE
   CURSOR student_enrolled IS
   SELECT studentnumber, status
     FROM classenrollment
    WHERE classid = &p_classid
    ORDER BY studentnumber;

   /* Declare variables to hold values returned by the
   cursor */

   v_studentnumber classenrollment.studentnumber%TYPE;
   v_status        classenrollment.status%TYPE;
BEGIN

   OPEN student_enrolled;

   /* Fetch first row from the cursor into the variables
   declared in the DECLARE section */

   FETCH student_enrolled
   INTO v_studentnumber, v_status;
   ...
END;
/
```

You usually want to fetch all the rows in the cursor, so the FETCH statement is usually placed inside a loop that executes over and over until the last row is fetched from the cursor. Chapter 10, "Controlling Program Execution in PL/SQL," provides a detailed description of loops. In order to detect when the last row in the cursor is fetched, you must use the explicit cursor attributes. There are four explicit cursor attributes:

- ✦ cursorname%FOUND returns TRUE if the last FETCH returned a row.
- ✦ cursorname%NOTFOUND returns TRUE if the last FETCH did not return a row.
- ✦ cursorname%ROWCOUNT returns the number of rows fetched so far.
- ✦ cursorname%ISOPEN returns TRUE if the cursor is still open.

The %ISOPEN attribute can be used to check if a cursor is open before executing a fetch, or to check if a cursor is still open and needs to be closed before exiting a program. The %FOUND, %NOTFOUND, and %ROWCOUNT cursor attributes can be used to determine when a loop should be exited.

```
ACCEPT p_classid NUMBER PROMPT "Enter Class id: "
DECLARE
   CURSOR student_enrolled IS
   SELECT studentnumber, status
     FROM classenrollment
    WHERE classid = &p_classid
    ORDER BY studentnumber;

   v_studentnumber classenrollment.studentnumber%TYPE;
   v_status        classenrollment.status%TYPE;
BEGIN

OPEN student_enrolled;

/* Use a basic loop to fetch rows from the cursor and
print each row on the screen until all rows are
fetched */

LOOP
        FETCH student_enrolled
          INTO v_studentnumber, v_status;

     /* Use the %NOTFOUND cursor attribute to determine
     when to exit the basic loop */

     EXIT WHEN student_enrolled%NOTFOUND;

     DBMS_OUTPUT.PUT_LINE(v_studentnumber||' '||v_status);
   END LOOP;
     CLOSE student_enrolled;
END;
/
```

%ROWTYPE

In Chapter 9, "Introduction to PL/SQL," you learned how to use %ROWTYPE to create a record based on the structure of a database table. The %ROWTYPE can also be used to create a record based on the structure of a cursor. The syntax for declaring this type of record is:

```
recordname    cursorname%ROWTYPE;
```

Here is an example using %ROWTYPE to declare a record that will hold the values returned by the cursor:

```
ACCEPT p_classid NUMBER PROMPT "Enter Class id: "
DECLARE
   CURSOR student_enrolled IS
   SELECT studentnumber, status
     FROM classenrollment
    WHERE classid = &p_classid
    ORDER BY studentnumber;

   /* The record declaration is now done using %ROWTYPE */
   student_record student_enrolled%ROWTYPE;

BEGIN

   OPEN student_enrolled;

   LOOP
        FETCH student_enrolled
          INTO student_record;

        EXIT WHEN student_enrolled%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE(student_record.studentnumber||
                             ' '||student_record.status);
   END LOOP;

   CLOSE student_enrolled;
END;
/
```

This program produces the following output:

```
Enter Class id: 50
1001 Confirmed
1002 Confirmed
1005 Confirmed
```

The cursor FOR loop

Whenever you use an explicit cursor, you always execute the same commands. You declare the cursor, open the cursor, fetch the rows, and close the cursor. To simplify your code, you can use the cursor FOR loop.

The cursor FOR loop has the following syntax:

```
FOR recordname IN cursorname LOOP
...
END LOOP;
```

The cursor FOR loop performs a number of steps automatically, simplifying your code. The following steps are executed by the cursor FOR loop:

1. Declares the record named in loop header.
2. Opens the cursor when the loop starts.
3. Fetches a row from cursor into the record each time through the loop.
4. Exits the loop after fetching the last record from the cursor.
5. Closes the cursor when exiting the loop.

Here is our previous example, which fetched all the students enrolled in a particular course, rewritten using the cursor FOR loop:

```
ACCEPT p_classid NUMBER PROMPT "Enter Class id: "
DECLARE
   CURSOR student_enrolled IS
   SELECT studentnumber, status
     FROM classenrollment
    WHERE classid = &p_classid
    ORDER BY studentnumber;

   /* No variable or record declarations required
   record will be declare implicitly by cursor FOR loop */
BEGIN

/* No OPEN or FETCH cursor statements required, cursor is
opened by loop and a row is automatically fetched into the
record each time through the loop */

   FOR student_record IN student_enrolled LOOP

     /* No EXIT statement required, loop automatically exits
     when there are no more rows to fetch */

     DBMS_OUTPUT.PUT_LINE(student_record.studentnumber||
                          ' '||student_record.status);
```

```
   END LOOP;

   /* No CLOSE statement required, loop automatically closes
   the cursor when exiting */
END;
/
```

This program produces the following output:

```
Enter Class id: 50
1001 Confirmed
1002 Confirmed
1005 Confirmed
```

This is a much simpler program than the example where we used a traditional loop and had to explicitly manipulate the cursor. When you use the cursor FOR loop, you can still use the EXIT command to exit the loop under other conditions. Using the EXIT command to exit the cursor FOR loop still closes the cursor automatically when exiting the loop.

The code can be further simplified by specifying the SELECT statement for the cursor within the cursor FOR loop.

```
FOR recordname IN (select_statement) LOOP
...
END LOOP;
```

If you specify the SELECT statement in the cursor FOR loop, the previous example looks like the following:

```
ACCEPT p_classid NUMBER PROMPT "Enter Class id: "
BEGIN
   /* No declare section required, because SELECT statement
   is specified in Cursor For Loop */

   FOR student_record IN      (SELECT studentnumber, status
                                 FROM classenrollment
                               WHERE classid = &p_classid
                               ORDER BY studentnumber)
   LOOP

     DBMS_OUTPUT.PUT_LINE(student_record.studentnumber||
                          ' '||student_record.status);
   END LOOP;

END;
/
```

This program produces the following output:

```
Enter Class id: 50
1001 Confirmed
1002 Confirmed
1005 Confirmed
```

The drawback to specifying the SELECT statement in the cursor FOR loop is that you no longer have access to the cursor attributes %FOUND, %NOTFOUND, %ROWCOUNT, and %ISOPEN because your cursor does not have a name. Another drawback is that you cannot use the same cursor again without rewriting the query.

Cursor variables

Although cursor variables are not included in the "Introduction to Oracle SQL and PL/SQL exam," they are included here to provide a more complete understanding of cursors. Cursor variables are pointers to the area in memory where the cursor data is stored. Cursor variables enable you to create dynamic cursors. When you create a static cursor, you specify the SELECT statement associated with the cursor when you declare the cursor. When you create a cursor variable, you are creating a dynamic cursor because you specify the SELECT statement when you open the cursor. The same cursor variable can be reopened with a different query as often as needed. Working with cursor variables is similar to working with cursors. You still have to go through four steps: declaring, opening, fetching, and closing the cursor.

The first step when working with cursor variables is to declare the cursor variable. Pointers to a cursor are of datatype REF CURSOR. To declare a cursor variable, you must first declare a REF CURSOR type and then a variable based on the REF CURSOR type.

The syntax for declaring a cursor variable is the following:

```
TYPE ref_cursor_type IS REF CURSOR
[RETURN record_structure];

cursor_variable    ref_cursor_type;
```

For example, in a program you could declare a cursor variable that will be used to fetch records from the courses table.

```
DECLARE
   TYPE courses_ref_cursor_type IS REF CURSOR
   RETURN courses%ROWTYPE;

   courses_cursor       courses_ref_cursor_type;
BEGIN
...
END;
/
```

After you have declared your cursor variable, you need to open the cursor. This is done in the executable section of the program. When you open a cursor variable, you must specify the SELECT statement that will be executed. The same cursor variable can be opened over and over with different SELECT statements. If you open the same cursor variable twice with different SELECT statements, the data from the first SELECT statement is lost and replaced with the data from the second SELECT statement.

The syntax for opening a cursor variable is the following:

```
OPEN cursor_variable FOR select_statement;
```

For example, you can open a cursor variable with a SELECT statement from the courses table.

```
DECLARE
   TYPE courses_ref_cursor_type IS REF CURSOR
   RETURN courses%ROWTYPE;

   courses_ref_cursor         courses_ref_cursor_type;
BEGIN
   /* Open cursor variable, specifying the SELECT statement
   to be executed. */
   OPEN courses_ref_cursor FOR
   SELECT * FROM courses;
   ...
   CLOSE courses_ref_cursor;
END;
/
```

Once the cursor has been opened, you can fetch from the cursor. The syntax for fetching from a cursor variable is the same as the syntax for fetching from an explicit cursor.

```
FETCH cursor_variable INTO record | variables
```

Because you usually want to fetch all the records returned by the cursor, you use a loop to execute the fetch over and over until all records have been fetched. Use the cursor attribute %NOTFOUND to determine when the last record has been fetched from the cursor.

After all the records from the cursor have been fetched, you should close the cursor using the CLOSE command. The syntax for the CLOSE command is the following:

CLOSE cursor_variable;

You cannot use cursor FOR loops with cursor variables.

The following example fetches the records from a cursor variable that returns course information and prints the course name on the screen:

```
DECLARE
   TYPE courses_ref_cursor_type IS REF CURSOR
   RETURN courses%ROWTYPE;

   courses_ref_cursor         courses_ref_cursor_type;

   /* Declare a record to hold the information returned by
   the cursor */
   course_record              courses%ROWTYPE;
BEGIN
   OPEN courses_ref_cursor FOR
   SELECT * FROM courses;

   LOOP
        /* Fetch the first record from the cursor */
        FETCH courses_ref_cursor
        INTO   course_record;

        /* Exit when last record retrieved from cursor */
        EXIT WHEN courses_ref_cursor%NOTFOUND;

        DBMS_OUTPUT.PUT_LINE('Course name is: '||
                             course_record.coursename);
   END LOOP;

   CLOSE courses_ref_cursor;
END;
/
```

Cursors with subqueries

Cursors can be built based on SELECT statements that contain subqueries. The subquery can be located in the WHERE clause, the HAVING clause, or the FROM clause of the SELECT statement. The subquery must be enclosed in parentheses.

The following example uses a subquery in the FROM clause of a cursor within a cursor FOR loop:

```
BEGIN
   /* The FROM Clause of the SELECT statement contains a
      subquery that select the total number of students
      enrolled in each class */

   FOR class_record IN
        (SELECT sc.classid, sc.instructorid, cl.nbr_students
         FROM scheduledclasses sc,
               (SELECT COUNT(*) nbr_students, classid
                FROM ClassEnrollment
```

```
                GROUP BY classid) cl
          WHERE sc.classid = cl.classid)
   LOOP

     DBMS_OUTPUT.PUT_LINE(class_record.classid||
        ' taught by: '||class_record.instructorid||
        ' total students: '||class_record.nbr_students);
   END LOOP;
END;
/
```

This program produces the following output:

```
50 taught by: 100 total students: 3
51 taught by: 200 total students: 3
53 taught by: 110 total students: 1
```

Cursor parameters

When you declare a cursor, you specify a WHERE clause that determines which rows are selected by the cursor. The WHERE clause is specified when the program is written. Using cursor parameters, you can specify values used in the WHERE clause at run time.

The syntax for declaring cursor parameters is the following:

```
CURSOR cursorname (p_parameter1 datatype,
 p_parameter2 datatype, ...)
IS SELECT columns
FROM table
WHERE column1 = p_parameter1
AND column2 = p_parameter2;
```

The syntax for opening cursors with parameters is the following:

Using the OPEN command

```
OPEN cursorname(parametervalue1, parametervalue2, ...);
```

Using cursor FOR loops

```
FOR recordname IN cursorname(parametervalue1,
parametervalue2,...) LOOP
```

In the following example, cursor parameters are used to specify the classid for which students are listed at runtime:

```
ACCEPT p_locationid NUMBER PROMPT "Enter Location id: "
DECLARE
   /* Declare a cursor to get a list of all classes at a
   particular location */
```

```
    CURSOR classlist IS
    SELECT classid
    FROM scheduledclasses
    WHERE locationId = &p_locationid;

    /* Declare a cursor to get all students in a particular
    class, classid will be specified at runtime in a
    parameter */

    CURSOR student_enrolled (p_classid NUMBER) IS
    SELECT studentnumber, status
    FROM classenrollment
    WHERE classid = p_classid;

BEGIN
    /* Fetch each classid in the specified location */
    FOR location_record IN classlist LOOP

         /* Fetch list of students for each classid, by
         passing location_record.classid to the
         student_enrolled cursor */

         FOR student_record IN
          student_enrolled(location_record.classid) LOOP

              DBMS_OUTPUT.PUT_LINE('Class: '||
                              location_record.classid||
                              ' Student: '||
                              student_record.studentnumber||
                              ' '||student_record.status);
          END LOOP;
    END LOOP;
END;
/
```

This program produces the following output:

```
Enter Location id: 300
Class: 51 Student: 1003 Cancelled
Class: 51 Student: 1004 Confirmed
Class: 51 Student: 1008 Confirmed
Class: 53 Student: 1003 Hold
```

FOR UPDATE and WHERE CURRENT OF

Explicit cursors are often used in batch jobs that select and update a number of rows in one or more tables. When a cursor is opened, all the rows returned by the SELECT statement are placed in the cursor. If the contents of the accessed table change while the program fetches rows from the cursor, the program does not pick up those changes. To prevent changes from being made to the records being processed by the cursor, you can use the FOR UPDATE option. The FOR UPDATE option

locks the records in the database when the cursor is opened. The records remain locked until a COMMIT or ROLLBACK is performed. This is very useful in programs that select a set of records and later update those records. The FOR UPDATE option is specified when the cursor is declared.

The syntax of the FOR UPDATE clause is the following:

```
CURSOR cursorname IS
SELECT columns FROM table WHERE condition FOR UPDATE;
```

If a SELECT statement fetches data from more than one table, records in all the tables accessed will be locked when you specify FOR UPDATE. If you are only updating one or some of the tables, you can instruct Oracle to lock only the rows in those tables by specifying the names of columns in the updated tables after the FOR UPDATE.

In the following example, all the rows in both the classenrollment table and the students table would be locked.

```
SELECT ce.studentnumber, ce.status, s.lastname
  FROM classenrollment ce, students s
 WHERE s.studentnumber=ce.studentnumber
   FOR UPDATE;
```

If you are updating the classenrollment table, but not the students table, specify the name of any column on the classenrollment table after the FOR UPDATE clause. Specifying a column on the classenrollment table instructs Oracle to lock only the records selected from the classenrollment table.

```
SELECT ce.studentnumber, ce.status, s.lastname
  FROM classenrollment ce, students s
 WHERE s.studentnumber=ce.studentnumber
   FOR UPDATE OF ce.status;
```

When you specify FOR UPDATE in the cursor declaration, all the rows selected by the cursor are locked when the cursor is opened. If one of the records being selected for update is already locked by another user, your program waits until that record lock is released. This can cause problems if a record is locked for an extended period of time. If you prefer not to have the program wait until the record is unlocked, you can specify the NOWAIT option when you declare the cursor. When you specify NOWAIT, an exception is raised when you open the cursor, and one of the records being selected for update is locked.

The syntax of the NOWAIT clause is the following:

```
CURSOR cursorname IS
SELECT columns FROM table WHERE condition FOR UPDATE NOWAIT;

CURSOR cursorname IS
SELECT columns FROM table WHERE condition FOR UPDATE OF
columnname NOWAIT;
```

The FOR UPDATE cursor not only locks records, it also allows you to use the WHERE CURRENT OF clause in UPDATE or DELETE statements.

The syntax of the WHERE CURRENT OF option is the following:

```
UPDATE tablename
SET columnname = newvalue
WHERE CURRENT OF cursorname;
```

The following example checks all the student enrollments and updates their status based on whether their course is confirmed. This program selects all student records and then updates their status, so you use the FOR UPDATE option to lock the records when the cursor is opened. If any records selected by the cursor are already locked, you raise an exception instead of waiting for the locks to be released with the NOWAIT clause. In the UPDATE statement, you use the WHERE CURRENT OF option to identify the record to be updated.

```
DECLARE

   /* Declare cursor to select all enrollments
   using FOR UPDATE option */

   CURSOR enrollments
   IS SELECT studentnumber, classid, status
   FROM classenrollment FOR UPDATE of status NOWAIT;

   v_status scheduledclasses.status%TYPE;
BEGIN

   FOR enrollment_record IN enrollments LOOP

        /* For each enrollment check the status of the class
        for that enrollment */

        SELECT status
          INTO v_status
          FROM scheduledclasses
         WHERE classid = enrollment_record.classid;

        /* If the class is cancelled, update the enrollment
        record to indicate the class is cancelled, use the
        WHERE CURRENT OF option to specify which record
        should be updated */

        IF v_status = 'Cancelled' THEN
              UPDATE classenrollment
                 SET status = 'Course cancelled'
               WHERE CURRENT OF enrollments;
        END IF;
   END LOOP;
END;
/
```

Composite and Collection Datatypes

As discussed in Chapter 9, "Introduction to PL/SQL," certain datatypes can hold more than one value at a time. These datatypes are referred to as *composite* or *collection datatypes*. If you create a variable with one of the collection datatypes, your variable can hold more than one value. Record datatypes can hold a single value for many different fields. Table and VARRAY datatypes can hold multiple values of a field or a set of fields.

PL/SQL records

A PL/SQL record is a variable made up of one or more fields, each of which can hold a different value. A record is used to hold information about an object. For example, you can create a PL/SQL record to hold information about a particular course. The record can contain a field for coursenumber, a field for coursename, and a field for retailprice. Records enable you to hold related information in one record variable, instead of creating separate scalar variables for each value you want to store. PL/SQL records are often used to hold the values returned by a SELECT statement.

The following example creates a record to hold a student's firstname, lastname, city, and email address. The example specifies the datatype and size of each field using the %TYPE clause.

```
DECLARE
   TYPE student_rec_struct IS RECORD
        (firstname    students.firstname%TYPE,
         lastname       students.lastname%TYPE,
         city           students.city%TYPE,
         email          students.email%TYPE);

   /* Create a record called student_record based on the
   record structure called student_rec_struct */

   student_record  student_rec_struct;
BEGIN
   ...
END;
/
```

A record can also be created using the %ROWTYPE clause. This clause creates a record based on the structure of a table. The fields within the records have the same names and datatypes as columns in the database table. The %ROWTYPE clause is very useful when you are selecting many columns from a database table, because you may not have to modify the code if the database table changes.

The syntax for creating a record using %ROWTYPE is the following:

```
    record_name     table_name%ROWTYPE;
```

The following example fetches all the information in the students table for a particular student number and prints the student's first and last name on the screen:

```
ACCEPT p_student NUMBER PROMPT "Enter student number: "
DECLARE
   student_record  students%ROWTYPE
BEGIN
   SELECT *
     INTO student_record
     FROM students
    WHERE studentnumber = &p_student;

   DBMS_OUTPUT.PUT_LINE(student_record.firstname||' '||
   student_record.lastname);
END;
/
```

This program produces the following output:

```
Enter student number: 1000
John Smith
```

More information on records and %ROWTYPE can be found in Chapter 9, "Introduction to PL/SQL."

An entire record can be populated at once using a SELECT statement. When you use a SELECT statement to populate the record, you must list the columns in the SELECT statement in the same order as the fields are listed in the record structure.

The syntax to populate a record with a SELECT statement is the following:

```
SELECT column1, column2, column3
  INTO recordname
  FROM table
 WHERE condition;
```

Instead of populating the record with a SELECT statement, individual fields within the record can be populated one by one. To reference a particular field within the record, you must specify the record name, a period (.), and then the field name. This syntax is used to populate individual fields within a record or to retrieve values from individual fields within a record.

The syntax to populate a field within a record is the following:

```
recordname.fieldname := value;
```

In the following example, a student record is populated using a SELECT statement, and the student's first and last names are printed on the screen:

```
ACCEPT p_student NUMBER PROMPT "Enter student number: "
DECLARE
   TYPE student_rec_struct IS RECORD
        (firstname    students.firstname%TYPE,
         lastname       students.lastname%TYPE,
         city           students.city%TYPE,
         email          students.email%TYPE);

   student_record  student_rec_struct;
BEGIN

   /* Specify the name of the record to be populated in the
   INTO clause of the SELECT statement */

   SELECT firstname, lastname, city, email
     INTO student_record
     FROM students
    WHERE studentnumber = &p_studentnumber;

   /* The student's name is printed on the screen */

   DBMS_OUTPUT.PUT_LINE(student_record.firstname||' '||
                  student_record.lastname);
END;
/
```

This program produces the following output:

```
Enter student number: 1000
John Smith
```

Index-by tables

An index-by table (formerly known as a *PL/SQL table*) is a composite variable that can hold multiple values of the same datatype and size. An index-by table must be composed of a key of datatype BINARY_INTEGER and a variable. The variable may be a scalar variable or a composite variable, such as a record. (Tables of records are covered in the section "Tables of Records.") An index-by table is the PL/SQL version of an array. It is used to hold a set of values. For example, an index-by table can be used to hold the names of all the courses offered.

An index-by table is created in two steps. First, you specify the structure of the table and give that structure a name. When specifying the structure, you must specify the datatypes and sizes of the values you want to store in the table. You do not need to specify the number of rows in the index-by table. Index-by tables increase in size dynamically as you add rows.

The syntax for declaring the structure of an index-by table is the following:

```
TYPE structurename IS TABLE OF datatype(size)
INDEX BY BINARY_INTEGER;
```

The datatype and size of the field can be listed explicitly or may be specified using the %TYPE clause.

The following two examples create an index-by table structure for an index-by table to hold course names. The first example specifies the datatype and size of the field explicitly; the second example specifies the datatype and size of the field using the %TYPE clause.

```
DECLARE
   /* Declare a table structure  and specify the datatype
   and size explicitly */
   TYPE course_table_struct IS TABLE OF VARCHAR2(200)
        INDEX BY BINARY_INTEGER;
..
BEGIN
..
END;
/

DECLARE
   /* Declare a table structure and use %TYPE to make the
   field the same datatype and size as the column in the
   database */
   TYPE course_table_struct IS TABLE OF
        courses.coursename%TYPE
        INDEX BY BINARY_INTEGER;
..
BEGIN
..
END;
/
```

After the table structure is declared, you create one or more index-by tables based on the structure. The syntax for creating the table is the following

```
table_name    structure_name;
```

The following example creates an index-by table to hold all the course names:

```
DECLARE
   TYPE course_table_struct IS TABLE OF
        courses.coursename%TYPE
        INDEX BY BINARY_INTEGER;
```

```
   /* Create the index-by table based on the
    course_table_struct table structure declared above */

   course_table course_table_struct;
BEGIN
..
END;
/
```

Once the index-by table has been created, you can populate rows in the index-by table with values. Each row in the index-by table contains a value and a key integer that enables you to identify the row. Whenever you populate or retrieve a value from an index-by table, you must specify which row in the index-by table you want to access. For example, you can have a row holding the course name "Database Administration" and the integer 5. When you want to retrieve the course name "Database Administration", you specify row 5. The row number is specified in parentheses after the index-by table name. The row number can be any valid integer, including negative values. The first row populated in the array does not have to be row 1.

```
tablename(rownumber);
```

The following example populates the first row of an index-by table with the course name "Database Administration" and prints that course name on the screen:

```
DECLARE
   TYPE course_table_struct IS TABLE OF
        courses.coursename%TYPE
        INDEX BY BINARY_INTEGER;

   course_table    course_table_struct;
BEGIN
   /* Populate first row of the index-by table with 'Database
administration' */
   course_table(1)    := 'Database Administration';

   /* Print first row of the index-by table on the screen */
   DBMS_OUTPUT.PUT_LINE(course_table(1));
END;
/
```

This program produces the following output:

```
Database Administration
```

Explicit cursors are often used to populate arrays. Each value returned by the cursor is put into a different row of the index-by table.

```
DECLARE
   TYPE course_table_struct IS TABLE OF
        courses.coursename%TYPE
        INDEX BY BINARY_INTEGER;

   course_table    course_table_struct;

   CURSOR courses_cursor IS
        SELECT coursename
          FROM courses
         ORDER BY coursenumber;

   /* Declare a counter to specify the row of the index-by
   table being accessed */
   v_row      NUMBER := 1;
BEGIN

   OPEN courses_cursor;

   LOOP
     /* Fetch each row from the cursor into a different
        row within the index-by table course_table */
        FETCH courses_cursor
         INTO course_table(v_row);

        EXIT WHEN courses_cursor%NOTFOUND;

        /* Print each row of the index-by table course_table
        on the screen */
        DBMS_OUTPUT.PUT_LINE(course_table(v_row));

        /* Increment the counter, so a different row is
        populated the next time through the loop */
        v_row := v_row + 1;
   END LOOP;

   CLOSE courses_cursor;
END;
/
```

This program produces the following output:

```
Basic SQL
Advanced SQL
Database Performance Basics
Performance Tuning your Database
Database Administration
Backing up your database
Basic PL/SQL
Advanced PL/SQL
Using your PL/SQL skills
```

Tables of records

When you create an index-by table, you can create a table that holds multiple values of one field, or multiple values of more than one field. An index-by table that holds multiple values of more than one field is called a *table of records*. To create a table of records, you must execute three steps:

1. Declare a record structure.
2. Declare a table structure that can hold the record structure.
3. Declare a table based on the table structure.

The following example declares a table of student records that can hold the first name, last name, city, and email address of one or more students:

```
DECLARE

   /* Declare record structure to hold first name, last name
   city and email address of a student */
   TYPE student_rec_struct IS RECORD
        (firstname    students.firstname%TYPE,
         lastname       students.lastname%TYPE,
         city           students.city%TYPE,
         email          students.email%TYPE);

   /* Declare table structure that can hold the record
   structure */
   TYPE student_table_struct IS TABLE OF student_rec_struct
   INDEX BY BINARY_INTEGER;

   /* Create PL/SQL table based on table structure */
   student_table   student_table_struct;
BEGIN
   ...
END;
/
```

When you want to reference or populate a value in a table of records, you must specify which element of the record and which row in the table you want to access. The syntax for referencing an element in a table of records is the following:

```
table_name(row).fieldname
```

The following example populates and prints the first row of an index-by table that holds student records:

```
DECLARE
   TYPE student_rec_struct IS RECORD
        (firstname      students.firstname%TYPE,
```

```
          lastname       students.lastname%TYPE,
          city           students.city%TYPE,
          email          students.email%TYPE);

   TYPE student_table_struct IS TABLE OF student_rec_struct
   INDEX BY BINARY_INTEGER;

   student_table   student_table_struct;
BEGIN

   /* Populate the first row of the PL/SQL table
   student_table */
   student_table(1).firstname := 'John';
   student_table(1).lastname  := 'Doe';
   student_table(1).city      := 'New York';
   student_table(1).email     := 'john.doe@ecom.com';

   /* Print the values in the first row of the index-by table
   student_table */
   DBMS_OUTPUT.PUT_LINE('Name is: '||
        student_table(1).firstname||' '||
        student_table(1).lastname||
        ' in city '||student_table(1).city||
        ' email '||student_table(1).email);
END;
/
```

This program produces the following output:

```
Name is: John Doe in city New York email john.doe@ecom.com
```

The %ROWTYPE clause may be used when creating a table of records to create an index-by table that can hold records that match the structure of a database table. If you use %ROWTYPE, you do not need to declare the record structure; instead, you specify the %ROWTYPE clause in the index-by table structure declaration. The following example declares a table of records to hold all the information stored in the students table:

```
DECLARE
   TYPE student_table_struct IS TABLE OF students%ROWTYPE
   INDEX BY BINARY_INTEGER;
   student_table   student_table_struct;
BEGIN
   ...
END;
/
```

Nested tables

In Oracle 8, a new form of PL/SQL table was added called the *nested table* (formerly known as the *V8 PL/SQL table*). To create a nested table, omit the INDEX BY BINARY_INTEGER clause from the table structure declaration. Nested tables

operate slightly differently from the index-by table. When you use a nested table, you must initialize it by specifying an initial value for the first row or rows. When you want to populate a row in a nested table, you must create the row first and then populate it.

The following is the syntax for declaring a nested table:

```
TYPE structurename IS TABLE OF datatype(size);

tablename     structurename;
```

The syntax for specifying an initial value for the first row or rows uses a constructor method. Constructor methods are used to populate objects; they are created by the database. The constructor method for tables has the same name as the table structure. To use the constructor method, you specify the name of the constructor method and then the value(s) to put in the row(s)in parentheses. The initialization can be done when the table is declared or in the BEGIN section.

The syntax for initializing one row in a nested table in the DECLARE section is the following:

```
tablename     structurename := structurename(initialvalue);
```

The syntax for initializing one row in a nested table in the BEGIN section is the following:

```
BEGIN
tablename := structurename(initialvalue);
```

The syntax for initializing multiple rows in a nested table in the DECLARE section is the following:

```
tablename     structurename := structurename(initialvalue1,
initialvalue2, initialvalue3, ...);
```

The syntax for initializing multiple rows in a nested table in the BEGIN section is the following:

```
BEGIN
tablename := structurename(initialvalue1, initialvalue2,
initialvalue3, ...);
```

When you want to populate a row in a nested table, you must use the table method EXTEND to add one or more rows to the PL/SQL table before they are populated. The first row added will always be row one, the second row added will always be row two, and so on. EXTEND is one of the collection methods that can be used to examine the contents of and manipulate collections. Additional collection methods are covered in “Collection Methods,” later in this chapter.

```
DECLARE
   TYPE course_table_struct IS TABLE OF
        courses.coursename%TYPE;

   /* Specify initial value for new rows using
   constructor method */
   course_table    course_table_struct
                        := course_table_struct(null);
BEGIN

   /* Create a new row in the course_table
   using the EXTEND method */
   course_table.EXTEND;

   /* Populate the new row of the nested table with
   'Database administration' */
   course_table(1) := 'Database Administration';

   /* Print first row of the nested table on the screen */
   DBMS_OUTPUT.PUT_LINE(course_table(1));

END;
/
```

This program produces the following output:

```
Database Administration
```

VARRAYs

VARRAYs are not included in the "Introduction to Oracle SQL and PL/SQL exam" but are included here so that a complete list of the composite datatypes is provided. The VARRAY type is a datatype that holds a fixed number of values of a particular datatype. Unlike PL/SQL tables whose size can change dynamically, the size of a VARRAY must be specified when it is declared, and its size remains the same at all times. When you use a VARRAY, you must give an initial value for the first row or rows. When you want to populate a row in a VARRAY, you must create the row first and then populate it.

Creating a VARRAY is done in two steps. First, you specify the structure of the VARRAY and give that structure a name. When specifying the structure, you must specify the datatypes and sizes of the values you want to store in the VARRAY and the number of rows in the VARRAY.

The syntax for declaring the structure of a VARRAY is the following:

```
TYPE varray_structure IS VARRAY(size) OF datatype(size);
```

The %TYPE clause may be used to specify a datatype and size. A record structure can also be specified instead of a datatype and size, if you want to create a VARRAY of records.

After the VARRAY structure is declared, you create one or more VARRAYs based on the structure. The syntax for creating the VARRAY is the following:

```
varray_name   varray_structure;
```

The syntax for specifying an initial value for the first row or rows uses a constructor method. The constructor method has the same name as the VARRAY structure. After specifying the constructor method, you specify the initial value for new rows in parentheses. The initialization can be done when the VARRAY is declared or in the BEGIN section.

The syntax for initializing one row in a VARRAY table in the DECLARE section is the following:

```
varray_name   varray_structure :=
              varray_structure(initialvalue);
```

The syntax for initializing one row in a V8 PL/SQL table in the BEGIN section is the following:

```
BEGIN
varray_name := varray_structure(initialvalue);
```

The syntax for initializing multiple rows in a V8 PL/SQL table is the following:

```
varray_name   varray_structure :=
              varray_structure(initialvalue1, initialvalue2,
              initialvalue3, ...);
```

Or

```
BEGIN
varray_name   := varray_structure(initialvalue1,
              initialvalue2, initialvalue3, ...);
```

When you want to populate a row in a VARRAY table, you must use the EXTEND method to add one or more rows to the VARRAY before they are populated. The first row added always is row one, the second row added always is row two, and so on. The EXTEND method and other methods that can be used with VARRAYs are explained later in this chapter in the section “Collection Methods.”

The following example creates a VARRAY that holds 12 location names. This VARRAY can be used to hold the busiest sales location of each month for a particular year.

```
DECLARE

   /* Declare structure of VARRAY to hold the 12 location
   names */
   TYPE location_struct IS VARRAY(12) OF VARCHAR2(50);
```

```
   /* Declare VARRAY based on structure and initialize
   first row to a NULL value*/
   location_varray      location_struct :=
                        location_struct(null);
BEGIN
   ...
END;
/
```

When you want to populate or reference a value stored in a VARRAY, you must specify which row in the VARRAY you want to access. The row you want to access is specified in parentheses after the VARRAY name.

The following example populates and prints the first row of the VARRAY with the highest profit location in January 2001:

```
DECLARE

   TYPE location_struct IS VARRAY(12) OF VARCHAR2(50);

   location_varray      location_struct :=
                        location_struct(null);

   v_highest_profit     VARCHAR2(50);
BEGIN
   ...
   /* create and Populate first row of VARRAY with contents of
   the variable v_highest_profit */
   location_varray.EXTEND;
   location_varray(1) := v_highest_profit;

   /* Print the first value in the VARRAY on the screen */
   DBMS_OUTPUT.PUT_LINE(location_varray(1));
END;
/
```

Collection methods

A series of collection methods can be used to determine the size, and the rows populated, in any collection datatype: index-by tables, VARRAYs, and nested tables. These methods enable you to find out information about the collection and also enable you to manipulate the collection. The following is a list of the collection methods and their purposes:

- ✦ EXISTS(row) returns TRUE if the row specified exists.
- ✦ COUNT returns the number of rows.
- ✦ FIRST returns the row number of the first populated row.
- ✦ LAST returns the row number of the last populated row.

- ✦ PRIOR(row) returns the row number of the last row populated before the row specified.
- ✦ NEXT(row) returns the row number of the next row populated after the row specified.
- ✦ DELETE removes all rows.
- ✦ DELETE(row) removes the specified row.
- ✦ DELETE(start_row,end_row) removes all rows between and including the start_row and end_row.
- ✦ TRIM removes the last row.
- ✦ TRIM(n) removes the last n rows.
- ✦ EXTEND adds one row.
- ✦ EXTEND(n) adds n rows.
- ✦ EXTEND(n,m) adds n copies of row m.

The TRIM and EXTEND methods apply only to nested tables and VARRAYs.

The syntax for collection methods is the following:

```
collection_name.method_name(arguments)
```

The following example uses the COUNT method to display the number of rows contained in an index-by table:

```
DECLARE
   TYPE course_table_struct IS TABLE OF
   courses.coursename%TYPE
   INDEX BY BINARY_INTEGER;

   course_table    course_table_struct;

   CURSOR courses_cursor IS
        SELECT coursename
          FROM courses
         ORDER BY coursenumber;

   v_row      NUMBER := 1;
BEGIN

   OPEN courses_cursor;

   LOOP
        FETCH courses_cursor
         INTO course_table(v_row);

        EXIT WHEN courses_cursor%NOTFOUND;
```

```
        DBMS_OUTPUT.PUT_LINE(course_table(v_row));

        v_row := v_row + 1;
    END LOOP;

    CLOSE courses_cursor;

    /* Use the table method COUNT to find out how many
    rows are in the index-by table course_table */
    DBMS_OUTPUT.PUT_LINE('Total rows: '||course_table.COUNT);
END;
/
```

This program produces the following output:

```
Basic SQL
Advanced SQL
Database Performance Basics
Performance Tuning your Database
Database Administration
Backing up your database
Basic PL/SQL
Advanced PL/SQL
Using your PL/SQL skills
Total rows: 9
```

The following example populates and prints the first row in a nested table with a course name. The EXTEND method is used to create the new row in the nested table.

```
DECLARE
    TYPE course_table_struct IS TABLE OF
        courses.coursename%TYPE;

    /* Specify initial value for new rows using
    constructor method */
    course_table    course_table_struct
                        := course_table_struct(null);
BEGIN

    /* Create a new row in the course_table
    using the EXTEND method */
    course_table.EXTEND;

    /* Populate the new row of the nested table with
    'Database administration' */
    course_table(1)    := 'Database Administration';

    /* Print first row of the nested table on the screen */
    DBMS_OUTPUT.PUT_LINE(course_table(1));

END;
/
```

This program produces the following output:

```
Database Administration
```

Key Point Summary

SQL statements can be embedded within PL/SQL programs enabling you to write SELECT statements to access the data in the database and Data Manipulation Language (DML) statements to modify the data in the database. In this chapter, you learned how to write SQL statements in your PL/SQL programs.

- ✦ SELECT statements require an INTO clause in PL/SQL code.
- ✦ SELECT statements return an error if they do not select any rows or if they select more than one row.
- ✦ INSERT, UPDATE, and DELETE statements can be executed from within PL/SQL programs.
- ✦ Data Definition Language (DDL) and Data Control Language (DCL) commands cannot be executed directly from PL/SQL programs.
- ✦ Implicit cursor attributes enable you to find out how many rows were affected by DML commands.
- ✦ Explicit cursors can be used to select more than one row from the database.
- ✦ The FOR UPDATE clause can be used to lock records selected by a cursor.
- ✦ The WHERE CURRENT OF clause enables you to identify which row selected by a cursor should be updated in the database.
- ✦ Cursor parameters can be used to affect which rows are selected by an explicit cursor at runtime.
- ✦ Subqueries can be used in the SELECT statements of cursors.

A number of composite datatypes can hold more than one value. PL/SQL tables, VARRAYs, and records are all datatypes that can hold more than one value. This chapter discusses the differences between these datatypes and how each one is used.

- ✦ The PL/SQL record can hold many fields of information about an object.
- ✦ The PL/SQL table can hold multiple values of a field.
- ✦ The PL/SQL table of records can hold multiple records.
- ✦ The VARRAY can hold a fixed number of values of a field.

✦ ✦ ✦

STUDY GUIDE

Now that you have learned about interacting with the database and composite datatypes, you should test your understanding by reviewing the assessment questions and performing the exercises that follow.

Assessment Questions

1. The following SELECT statement is executed in a PL/SQL program as an implicit cursor. Which of the following clauses should be added to the SELECT statement in order for the program to execute successfully?

```
SELECT coursenumber, coursename
  FROM courses
 WHERE coursenumber = 100;
```

   A. No clause

   B. ORDER BY coursename

   C. INTO v_coursenumber

   D. POPULATE v_coursenumber, v_coursename

   E. INTO v_coursenumber, v_coursename

2. A SELECT statement executed as an implicit cursor in a PL/SQL program returns no rows. What happens when you execute the program?

   A. An exception is raised because the SELECT statement returns no rows.

   B. The program executes successfully.

   C. The program cannot be executed because you get a compile error.

   D. An exception is raised because the SELECT statement returns more than one row.

   E. You cannot execute a SELECT statement from within a PL/SQL program.

3. A DELETE statement in a PL/SQL program does not delete any rows. What happens when you execute the program?

   A. An exception is raised because the DELETE statement deleted no rows.

   B. The program executes successfully.

   C. The program cannot be executed because you get a compile error.

   D. An exception is raised because the DELETE statement deletes more than one row.

   E. You cannot execute a DELETE statement from within a PL/SQL program.

4. You write a PL/SQL program to select the names of all the courses in the courses table. Which of the following cursor declarations should you use?

A. coursenames IS CURSOR SELECT coursename FROM courses;

B. coursenames IS CURSOR SELECT coursename FROM courses ORDER BY coursenumber;

C. CURSOR coursenames IS SELECT coursename INTO v_coursename FROM courses;

D. CURSOR coursenames IS SELECT coursename INTO v_coursename FROM courses ORDER BY coursenumber;

E. CURSOR coursenames IS SELECT coursename FROM courses;

5. You declare a cursor to select all the records in the classenrollment table. Later in the program, you update the records, and you want them locked when the cursor is opened. Which of the following clauses should you specify when you declare the cursor?

A. FOR UPDATE

B. LOCK RECORDS

C. WHERE CURRENT OF

D. INTO

E. LOCK CURSOR

6. You have a loop that fetches all the rows returned by a cursor. Which of the following cursor attributes can you use to determine when to exit the loop because all the rows have been fetched

A. %FOUND

B. %NOTFOUND

C. %ISOPEN

D. Either A or B

E. Either A, B or C

7. You declare the following cursor in a PL/SQL program. Which of the following commands successfully opens the cursor?

```
CURSOR student_cursor (p_studentid NUMBER)
IS SELECT firstname, lastname
FROM students
WHERE studentid = p_studentid;
```

A. OPEN student_cursor;

B. OPEN student_cursor(1000);

C. OPEN student_cursor('John Doe');

D. OPEN student_cursor 1000;

E. OPEN student_cursor 'John Doe';

8. You create a PL/SQL table called student_names. Which of the following commands populates the first row in the table with the name "James Decker"?

A. student_names = 'James Decker';

B. student_names := 'James Decker';

C. 'James Decker' := student_names(1);

D. student_names.name := 'James Decker';

E. student_names(1) := 'James Decker';

9. You create a record called student_record that contains the elements firstname, middleinitial, and lastname. Which of the following commands populates the lastname of the record with the name "Decker"?

A. student_record.lastname := 'Decker';

B. student_record(3) := 'Decker';

C. student_record(1) := 'Decker';

D. student_record := 'Decker';

E. lastname := 'Decker';

10. Which of the following steps is performed automatically by the cursor FOR loop. (Choose all that apply.)

A. Opens cursor.

B. Closes cursor.

C. Fetches values from cursor into a record.

D. Declares a record to hold values returned by cursor.

E. Returns an error if cursor returns no rows.

11. The SELECT statement for an explicit cursor does not return any rows. What happens when you execute the program?

A. An exception is raised because the SELECT statement returns no rows.

B. The program executes successfully.

C. The program cannot be executed because you get a compile error.

D. An exception is raised because the SELECT statement returns more than one row.

E. You cannot execute a SELECT statement from within a PL/SQL program.

12. Which of the following cursor attributes tell you the number of rows that have been fetched from an explicit cursor:

    A. %FOUND

    B. %NOTFOUND

    C. %ISOPEN

    D. %ROWCOUNT

    E. %ROWNUMBER

Scenarios

1. The company sales teams are frequently asked when a particular course is running. You need to write a program that accepts a course ID and returns the date or dates and locations where the course is running.

    A. Write the SELECT statement that is used for the cursor.

    B. Declare a cursor called coursedates_cursor using your SELECT statement.

    C. Declare a record that holds the values returned by the cursor using %ROWTYPE.

    D. Write the command that opens the cursor.

    E. Write the command to fetch a record from the cursor into the record.

    F. Write the command to close the cursor.

    G. Use a Basic loop to fetch all the records and exit when there are no more records. Use the %NOTFOUND cursor attribute to determine when to exit the loop.

    H. Write a command using DBMS_OUTPUT.PUT_LINE to print each course date and location on the screen.

    I. Change your basic loop to a cursor FOR loop.

    J. Which commands can now be removed from your original code.

2. The company needs a program that updates the status of class enrollments when a class is confirmed or canceled. The program should accept a class ID and a status. The program should update all the records in the class enrollment table with the specified class ID to the new status. The program should return a message specifying how many records were updated.

    A. Should you use an INSERT, UPDATE, or DELETE in this program?

    B. Write the DML statement that updates the records in the class enrollment table.

C. What implicit cursor attribute can you use to find out how many rows were updated?

D. Write a program to perform the update. Use the ACCEPT command to accept the class ID and the status.

3. The sales staff are often asked questions about a particular course. The staff members need a program that prints the course name, price, and description of a particular course when they specify a course number.

A. Declare a record structure called course_record_type that can hold a course name, price, and description.

B. Declare a record called course_record based on the record structure course_record_type.

C. Write a SELECT statement that populates the record course_record with the values in the columns coursename, retailprice, and description for a course ID stored in a local PL/SQL variable called v_courseid.

D. Using DBMS_OUTPUT.PUT_LINE, write a command that prints the course name on the screen.

Lab Exercises

Lab 11-1 Cursor FOR loop and index-by table

A. Write a program that fetches all the course names, stores them in an index-by table, and prints the course names on the screen. First, declare an index-by table structure called coursename_table_type that can hold course names.

B. Declare an index-by table based on the table structure coursename_table.

C. Declare a cursor called courses_cursor that fetches all the course names from the courses table.

D. Declare an integer variable called counter. This variable will be used to specify which row of the PL/SQL table is being populated. Initialize this variable to one.

E. Using a cursor FOR loop, write code that fetches each row from the cursor and populates a row in the index-by table. Increment the variable counter each time through the loop, so that a different row in the PL/SQL table is populated each time through the loop.

F. Using DBMS_OUTPUT.PUT_LINE in the cursor FOR loop, print each course name on the screen.

G. Run your program; what is your output?

Lab 11-2 INSERT, UPDATE, DELETE

A. Write a PL/SQL program that inserts a row into the courses table with the following values: coursenumber=900, coursename='My Course', replacescourse=null, retailprice=500, description='How to insert rows'.

B. Write a SELECT statement to display the row you have just inserted.

C. Write a PL/SQL program that updates the row you just inserted with a new course description of "How to update rows".

D. Write a SELECT statement to display the row you have just updated.

E. Write a PL/SQL program that deletes the row you have just inserted.

F. Write a SELECT statement to ensure the row was deleted.

Lab 11-3 SELECT statements and records

A. Write a PL/SQL program that selects the first name, last name, and middle initial of student 1000 and displays the student's full name on the screen. Fetch each database column selected into a separate variable.

B. Modify the PL/SQL program to use a record to hold the values returned by the SELECT statement.

C. Modify the PL/SQL program to SELECT all the columns in the student table for student number 1000 and use a record created with the %ROWTYPE to hold the values returned.

D. Modify the SELECT statement to return the information for student number 9999, which does not exist. What error message is returned?

E. Remove the WHERE clause from your SELECT statement. What error message is returned?

Answers to Chapter Questions

Chapter Pre-Test

1. The INTO clause must be added to specify which variables hold the values returned by the SELECT statement in PL/SQL.

2. SQL%ROWCOUNT returns the number of records updated; SQL%FOUND and SQL%NOTFOUND indicate if any records were updated.

3. You must use explicit cursors to select multiple records from the database.

4. Adding the FOR UPDATE clause to a cursor declaration locks the records when the cursor is opened and selects the ROWID of records selected enabling you to use the WHERE CURRENT OF clause.

5. The %FOUND and %NOTFOUND attributes tell you whether all the rows have been fetched from an explicit cursor.
6. A PL/SQL table is a composite datatype that can hold multiple values of a particular datatype.
7. A cursor parameter is a value passed to the cursor at runtime that affects the rows selected by the cursor.
8. The cursor FOR loop opens the cursor, declares a record to hold the values returned by the cursor, fetches the value from the cursor into the record, loops until no more records are to be fetched, and closes the cursor without any additional coding. With the Basic loop, you must add code to perform that logic.
9. You write the SELECT statement to populate a cursor in the DECLARE section.
10. Yes, you get an error if a SELECT statement returns no rows.

Assessment Questions

1. **E** — You must add an INTO clause to specify a variable to hold each column that is selected.
2. **A** — If a SELECT statement returns no rows, an exception is raised at runtime. You may prefer to use an explicit cursor, if you do not want an exception raised.
3. **B** — The program executes successfully. Oracle does not raise an exception when a DELETE statement in a PL/SQL program does not delete any rows.
4. **E** — The syntax for declaring a cursor is CURSOR cursorname IS select_statement. The INTO clause is specified when you FETCH from the cursor, not when the cursor is declared.
5. **A** — The FOR UPDATE clause locks the records when the cursor is opened.
6. **D** — The %NOTFOUND or %FOUND attributes can be used to determine when the last row has been fetched from a cursor. %ISOPEN tells you whether the cursor is still open.
7. **B** — The syntax for opening a cursor with a parameter is OPEN cursorname(parametervalue). The parameter value must be the same datatype specified for the parameter in the cursor declaration.
8. **E** — The syntax to populate a row in a table is tablename(rownumber) := value;.
9. **A** — The syntax to populate a field in a record is recordname.fieldname := value;.
10. **A + B + C + D** — These steps are all performed by the cursor FOR loop. A cursor FOR loop does not return an error if the cursor returns no rows.

11. **B** — The program executes successfully if a SELECT statement executed by an explicit cursor returns no rows.
12. **D** — The %ROWCOUNT attribute returns the number of rows fetched from an explicit cursor.

Scenarios

1. The company sales team is frequently asked when a particular course is running. You need to write a program that accepts a course ID and returns the date or dates and locations where the course is running.

   A. Write the SELECT statement that will be used for the cursor.

   ```
   SELECT startdate, locationid
     FROM scheduledclasses
    WHERE coursenumber = &p_coursenumber;
   ```

   B. Declare a cursor called coursedates_cursor using your SELECT statement.

   ```
   CURSOR coursedates_cursor IS
   SELECT startdate, locationid
     FROM scheduledclasses
    WHERE coursenumber = &p_coursenumber;
   ```

   C. Declare a record that holds the values returned by the cursor using %ROWTYPE.

   ```
   coursedates_record coursedates_cursor%ROWTYPE;
   ```

   D. Write the command that opens the cursor.

   ```
   OPEN coursedates_cursor;
   ```

   E. Write the command to fetch a record from the cursor into the record.

   ```
   FETCH coursedates_cursor
    INTO coursedates_record;
   ```

   F. Write the command to close the cursor.

   ```
   CLOSE coursedates_cursor;
   ```

   G. Use a Basic loop to fetch all the records and exit when there are no more records. Use the %NOTFOUND cursor attribute to determine when to exit the loop.

   ```
   LOOP
      FETCH coursedates_cursor
       INTO coursedates_record;

      EXIT WHEN coursedates_cursor%NOTFOUND;
   END LOOP;
   ```

H. Write a command using DBMS_OUTPUT.PUT_LINE to print each course date and location on the screen. You may need to set SERVEROUTPUT ON in your SQL*Plus session to see the output.

```
DBMS_OUTPUT.PUT_LINE('Course date:'
||coursedates_record.startdate||' location: '
||coursedates_record.locationid);
```

I. Change your Basic loop to a cursor FOR loop.

```
FOR coursedates_record IN coursedates_cursor LOOP

  DBMS_OUTPUT.PUT_LINE('Course date:'
  ||coursedates_record.startdate||' location: '
  ||coursedates_record.locationid);

END LOOP;
```

J. The OPEN cursor, CLOSE cursor, FETCH INTO, EXIT WHEN, and record declaration can all be removed from the original code when you use the cursor FOR loop, because the cursor FOR loop executes all these commands implicitly.

2. You need to write a program that updates the status of class enrollments. The program should accept a class ID and a status. The program should update all the records in the class enrollment table with that particular class ID to the specified status. The program should return a message specifying the number of records that were updated.

A. Should you use an INSERT, UPDATE, or DELETE statement in this program?

An UPDATE statement should be used.

B. Write the DML statement that updates the records in the class enrollment table. Assume the status and class ID are stored in local PLSQL variables called v_status and v_classid.

```
UPDATE ClassEnrollment
   SET status = v_status
 WHERE classid = v_classid;
```

C. What implicit cursor attribute can you use to find out the number of rows that were updated?

SQL%ROWCOUNT

D. Write a program to perform the update. Use the ACCEPT command to accept the class ID and the status.

```
ACCEPT p_classid NUMBER -
PROMPT "Enter the class id to be updated: "
ACCEPT p_status CHAR -
PROMPT "Enter the new class status: "
```

```
DECLARE
   v_status   classenrollment.status%TYPE
              := '&p_status';
   v_classid  classenrollment.classid%TYPE
              := &p_classid;
BEGIN
   UPDATE ClassEnrollment
     SET status = v_status
   WHERE classid = v_classid;

DBMS_OUTPUT.PUT_LINE(SQL%ROWCOUNT||
  ' class enrollment records were updated.');
END;
/
```

3. The sales staff are often asked questions about a particular course. The staff members need a program that prints the course name, price, and description of a particular course when they specify a course number.

A. Declare a record structure called course_record_type that can hold a course name, price, and description.

```
TYPE course_record_type IS RECORD
(coursename     VARCHAR2(200),
 retailprice    INT,
 description    VARCHAR2);
```

B. Declare a record called course_record based on the record structure course_record_type.

```
course_record       course_record_type;
```

C. Write a SELECT statement that populates the record course_record with the values in the columns coursename, retailprice, and description for a course ID stored in a local PL/SQL variable called v_courseid.

```
SELECT coursename, retailprice, description
  INTO course_record
  FROM courses
 WHERE courseid = v_courseid;
```

D. Using DBMS_OUTPUT.PUT_LINE, write a command that prints the course name on the screen.

```
DBMS_OUTPUT.PUT_LINE('Course Name is :'
                     course_record.coursename);
```

Lab Exercises

Lab 11-1 Cursor FOR loop and index-by table

```
DECLARE
   /* Create index-by table structure called
   coursename_table_type */
   TYPE coursename_table_type IS TABLE OF
```

```
courses.coursename%TYPE
   INDEX BY BINARY_INTEGER;

   /* Create index-by table called coursename table */
   coursename_table      coursename_table_type;

   /* Declare cursor to fetch all coursenames from courses
   table */
   CURSOR courses_cursor IS
   SELECT coursename
     FROM courses;

   /* Declare a counter to keep track of which row in the
   index-by table is being populated */
   counter         PLS_INTEGER := 1;

BEGIN
   /* Fetch all rows from the cursor with cursor FOR loop */
   FOR course_record IN courses_cursor LOOP
        /* Populate coursename_table with row fetched from
        cursor */
        coursename_table(counter) :=
                   course_record.coursename;

        /* Print course name on the screen */
        DBMS_OUTPUT.PUT_LINE(coursename_table(counter));

        /* Increment counter so next time a new row will be
        populated in the coursename table */
        counter := counter+1;

   END LOOP;
END;
/
```

This program produces the following output:

```
Basic SQL
Advanced SQL
Performance Tuning your Database
Database Performance Basics
Database Administration
Backing up your database
Basic PL/SQL
Advanced PL/SQL
Using your PL/SQL skills
```

Lab 11-2 INSERT, UPDATE, DELETE

A.

```
BEGIN
     INSERT INTO courses (coursenumber, coursename,
     replacescourse, retailprice, description)
     VALUES (900, 'My Course',null, 500,
     'How to insert rows');
END;
/
```

B.

```
SELECT *
FROM courses
WHERE coursenumber=900;

COURSENUMBER COURSENAME
------------ ----------------------------------------
REPLACESCOURSE RETAILPRICE DESCRIPTION
-------------- ----------- ----------------------------
900          My Course
                         500 How to insert rows
```

C.

```
BEGIN
     UPDATE courses
     SET description='How to update rows'
     WHERE coursenumber=900;
END;
/
```

D.

```
SELECT *
FROM courses
WHERE coursenumber=900;

COURSENUMBER COURSENAME
------------ ----------------------------------------
REPLACESCOURSE RETAILPRICE DESCRIPTION
-------------- ----------- ----------------------------
900          My Course
                         500 How to update rows
```

E.

```
BEGIN
     DELETE FROM courses
     WHERE coursenumber=900;
END;
/
```

F.

```
SELECT *
FROM courses
WHERE coursenumber=900;

no rows selected
```

Lab 11-3 SELECT statements and records

A.

```
DECLARE
     v_firstname     students.firstname%TYPE;
     v_lastname      students.lastname%TYPE;
     v_middleinitial students.middleinitial%TYPE;
BEGIN
  SELECT firstname, lastname, middleinitial
    INTO v_firstname, v_lastname, v_middleinitial
    FROM students
   WHERE studentnumber=1000;

   DBMS_OUTPUT.PUT_LINE(v_firstname||' '||
v_middleinitial||' '||v_lastname);
 END;
```

B.

```
DECLARE
TYPE student_record_type IS RECORD
     (firstname            students.firstname%TYPE,
      lastname             students.lastname%TYPE,
      middleinitial        students.middleinitial%TYPE);

      student_record student_record_type;
BEGIN
  SELECT firstname, lastname, middleinitial
    INTO student_record
    FROM students
   WHERE studentnumber=1000;

   DBMS_OUTPUT.PUT_LINE(student_record.firstname||' '||
     student_record.middleinitial||
     ' '||student_record.lastname);
END;
```

C.

```
DECLARE
      student_record students%ROWTYPE;
BEGIN
  SELECT *
    INTO student_record
    FROM students
```

```
    WHERE studentnumber=1000;

    DBMS_OUTPUT.PUT_LINE(student_record.firstname||' '||
      student_record.middleinitial||
      ' '||student_record.lastname);
END;
```

D. ORA-01403: no data found

E. ORA-01422: exact fetch returns more than requested number of rows

CHAPTER 12

Handling Errors and Exceptions in PL/SQL

EXAM OBJECTIVES

- Handling Exceptions
  - Define PL/SQL exceptions
  - Recognize unhandled exceptions
  - List and use different types of PL/SQL exception handlers
  - Trap unanticipated errors
  - Describe the effect of exception propagation in nested blocks
  - Customize PL/SQL exception messages
- Use coding conventions
- Execute and test a PL/SQL block

CHAPTER PRE-TEST

1. What are compile errors, and how do you get information about them?
2. What are the three types of exceptions in PL/SQL?
3. How are exceptions raised?
4. What happens in a SQL*Plus environment when an exception is raised in a block with no exception section?
5. What does an exception handler do?
6. What does the WHEN OTHERS clause of the exception section do?
7. How can you capture the code and message of the current error in an exception handler?
8. How do you associate a descriptive name with an Oracle non-predefined error number?
9. How can you pass a user-defined error message to the calling environment, just like regular Oracle errors?
10. How can you tailor error-handling statements to a particular piece of PL/SQL code?

✦ Answers to these questions can be found at the end of the chapter. ✦

In the course of creating programs, errors are bound to occur. Sometimes your program won't compile, while sometimes it compiles and runs but it ends prematurely because of a runtime error. Sometimes your program appears to function correctly, but it won't do what it is supposed to do because of errors in the program logic. In this chapter, you will investigate how to handle the various errors that can occur in PL/SQL.

Types of Errors

The various errors that occur in a PL/SQL program can be classified as three types:

- ✦ Compile errors
- ✦ Program logic errors
- ✦ Runtime errors or exceptions

Compile errors

At compile time, the PL/SQL compiler examines the code and determines whether or not all of the references made are legal. For example, if a database table is used in the code, the compiler must determine if the object exists in the database and if the user has the proper privileges on it. It must also check that all of the identifiers that are used have been declared, and that variable assignments are legal. If the compilation is unsuccessful, you must examine the compilation errors in order to fix them and try again.

PL/SQL anonymous blocks are compiled just before running, and if any compile errors exist, the anonymous blocks will not execute. In SQL*Plus, these errors are output to the screen as follows:

```
SQL> DECLARE
  2    x NUMBER;
  3  BEGIN
  4    x := 10
  5  END;
  6  /
END;
*
ERROR at line 5:
ORA-06550: line 5, column 1:
PLS-00103: Encountered the symbol "END" when expecting one of
the following:
* & = - + ; < / > in mod not rem an exponent (**)
<> or != or ~= >= <= <> and or like between is null is not ||
is dangling
The symbol ";" was substituted for "END" to continue.
```

This output tells you that there is an error at line 5, column 1, and it shows that line with an asterisk under the part that the compiler believes contains an error. The line that the compiler reports as having an error often is not really the line at fault. In this example, the line before contains an error: it does not end with a semicolon (;). The actual error message displayed does not clearly state the problem but indicates that it reached the word END when it was expecting something else in order to make the line before legal.

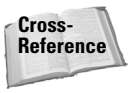
SQL*Plus handles the compilation of stored program units differently than anonymous blocks. Program units such as procedures and functions are compiled when they are first created. The following function calculates the tax on an amount and returns it, but again, a compilation error will be found due to a missing semicolon:

```
  1  CREATE FUNCTION tax(p_amount IN  NUMBER)
  2  RETURN NUMBER
  3  IS
  4    v_tax NUMBER;
  5  BEGIN
  6    v_tax := p_amount * 0.07
  7    RETURN v_tax;
  8* END;
SQL> /

Warning: Function created with compilation errors.
```

Note that SQL*Plus does not show the compilation error messages but creates the object and warns you of the errors. In order to discover what errors exist for stored program units that you own, you can query the USER_ERRORS data dictionary view. Even more convenient than this is to use the SQL*Plus command SHOW ERRORS, which shows only the errors for program units that you created in this session:

```
SQL> SHOW ERRORS
Errors for FUNCTION TAX:

LINE/COL ERROR
-------- -----------------------------------------------------------------
7/3      PLS-00103: Encountered the symbol "RETURN" when expecting one of
         the following:
         * & = - + ; < / > in mod not rem an exponent (**)
         <> or != or ~= >= <= <> and or like between is null is not ||
         is dangling

9/0      PLS-00103: Encountered the symbol "end-of-file" when expecting
         one of the following:
         begin function package pragma procedure subtype type use
         <an identifier> <a double-quoted delimited-identifier> cursor
         form current
```

Optionally, you can pass to the SHOW ERRORS command the name of a program unit, and it will show any compilation errors for it, even if you did not create it in your current session.



For more information on creating and using stored program units, refer to Chapter 13, "Introduction to Stored Programs."

Program logic errors

Errors in program logic, known as *bugs*, are often the most difficult to find because the program seems to run correctly. Such errors become known usually after the program has been tested several times. Searching for these errors is the process known as *debugging*. The process of debugging PL/SQL depends heavily on the environment that you are using for developing your code.

For instance, in some PL/SQL development environments, like Oracle's Procedure Builder tool, you can halt the execution of a program at a specified line in order to examine the values of variables at that point. You can then step through the code one line at a time to watch how the program behaves. In an environment like SQL*Plus, however, you do not have the ability to stop execution and step through line-by-line. The only real option you have for debugging in SQL*Plus is to write the values of variables to the screen at various points during the program using the DBMS_OUTPUT.PUT_LINE procedure. The following is an example of a program that completes successfully but does not do what it is supposed to. It should prompt the user for the name and price of a new course, and then insert that course into the table, using a course number determined by adding 10 to the highest course number.

```
DECLARE
  v_new_number Courses.CourseNumber%TYPE := 0;
  v_new_name   Courses.CourseName%TYPE := '&Name';
  v_new_price   Courses.RetailPrice%TYPE := &Price;
BEGIN
  SELECT MAX(CourseNumber) + 10
  INTO   v_new_price
  FROM   Courses;
  INSERT INTO Courses(CourseNumber, CourseName,
                      RetailPrice)
  VALUES(v_new_number, v_new_name, v_new_price);
END;
```

Here is a sample execution of the code, with a search for the new course:

```
SQL> /
Enter value for name: Intro to Shell Programming
Enter value for price: 1500
```

```
PL/SQL procedure successfully completed.

SQL> SELECT CourseNumber, CourseName, RetailPrice
  2  FROM   Courses
  3  WHERE  CourseName = 'Intro to Shell Programming'
  4  AND    RetailPrice = 1500;

no rows selected
```

In order to determine what went wrong, the last attempt is rolled back and output lines are added to the code to show the three variables at different stages of the program:

```
DECLARE
  v_new_number Courses.CourseNumber%TYPE := 0;
  v_new_name   Courses.CourseName%TYPE := '&Name';
  v_new_price   Courses.RetailPrice%TYPE := &Price;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before the SELECT');
  DBMS_OUTPUT.PUT_LINE('Number is: '||TO_CHAR(v_new_number));
  DBMS_OUTPUT.PUT_LINE('Name is: '||v_new_name);
  DBMS_OUTPUT.PUT_LINE('Price is: '||TO_CHAR(v_new_price));

  SELECT MAX(CourseNumber) + 10
  INTO   v_new_price
  FROM   Courses;

  DBMS_OUTPUT.PUT_LINE('After the SELECT');
  DBMS_OUTPUT.PUT_LINE('Number is: '||TO_CHAR(v_new_number));
  DBMS_OUTPUT.PUT_LINE('Name is: '||v_new_name);
  DBMS_OUTPUT.PUT_LINE('Price is: '||TO_CHAR(v_new_price));

  INSERT INTO Courses(CourseNumber, CourseName,
                      RetailPrice)
  VALUES(v_new_number, v_new_name, v_new_price);
END;

SQL> /
Enter value for name: Intro to Shell Programming
Enter value for price: 1500
Before the SELECT
Number is: 0
Name is: Intro to Shell Programming
Price is: 1500
After the SELECT
Number is: 0
Name is: Intro to Shell Programming
Price is: 330

PL/SQL procedure successfully completed.
```

Upon investigating the output, you should find that the SELECT statement appears to be populating the wrong variable — v_new_price has changed, not v_new_number. This is the error in program logic.

Runtime errors or exceptions

Runtime errors in PL/SQL are called *exceptions*. An exception is a PL/SQL identifier, meaning it must be declared in much the same way as a variable must be (although some exceptions are declared for you by Oracle), and the exception follows the same rules of scope as for variables: it can be referenced only within the block in which it is declared. These exceptions are then "raised" at runtime and alter the normal execution of your code. The rest of this chapter is devoted to the use of exceptions.

Exception Handling

Objective

- List and use different types of PL/SQL exception handlers

When an error condition is encountered during the execution of a PL/SQL block of code, an exception is raised, and the regular flow of the program is altered. Control of the program immediately jumps to the end of the current block and searches for an error handler in the exception section, which starts with the EXCEPTION keyword. This is done so that the program does not continue to execute lines of code, possibly causing even more errors to occur.

Some programming languages do not have a specific error-handling section, so they must test after each line whether or not an error has occurred. In PL/SQL, this is all done in the exception section, so your block will look like this:

```
DECLARE
  declaration statements;
BEGIN
  executable statements;
EXCEPTION
  error handlers;
END;
```

The error condition can be encountered by the Oracle server, which raises an exception, or you can raise exceptions in the code yourself, using the RAISE statement. Either way, control of the program jumps to the exception section. Because this section is optional, the error may propagate to the calling environment. This is what is called an *unhandled exception*. The following example illustrates an unhandled exception:

```
DECLARE
  v_num1 NUMBER := &first_number;
  v_num2 NUMBER := &second_number;
  v_result NUMBER;
BEGIN
  v_result := v_num1/v_num2;
  DBMS_OUTPUT.PUT_LINE('The quotient is '||TO_CHAR(v_result));
END;
```

This block prompts the user for two numbers and then prints the result when the first is divided by the second. Here are two executions of this code. The first one works correctly and gives the proper output, while the second raises an unhandled exception when the program attempts to divide by zero.

```
SQL> /
Enter value for first_number: 10
Enter value for second_number: 5
The quotient is 2

PL/SQL procedure successfully completed.

SQL> /
Enter value for first_number: 15
Enter value for second_number: 0
DECLARE
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 6
```

Notice that when this exception is propagated to the calling environment (in this case, SQL*Plus), it simply outputs the error code and message to the screen. Other calling environments deal with unhandled exceptions differently. In SQL*Plus, the first line of the error output states that there is an error at line 1, and it shows the asterisk under the word DECLARE. There is nothing wrong with line 1, but this output is telling you that an unhandled exception is in the block that begins on line 1. The real statement in error, along with the error code and message, are reported in the second and third lines of output.

In order to handle raised exceptions, you must first understand the three types of exceptions in PL/SQL:

- ✦ Predefined
- ✦ Non-predefined
- ✦ User-defined

Predefined exceptions

Because exceptions are identifiers like variables, constants, and cursors, they also need to be declared before they can be used. However, a number of common exceptions have been declared for you in the STANDARD package. This means that you may use them without first declaring them in the declaration section. One example of a predefined exception is ZERO_DIVIDE. In the last example, this exception was raised by the server when an attempt was made to divide a number by zero. In that example, the exception was not handled and, therefore, propagated to the calling environment. In this next example, an exception handler is included in the exception section:

```
DECLARE
  v_num1 NUMBER := &first_number;
  v_num2 NUMBER := &second_number;
  v_result NUMBER;
BEGIN
  v_result := v_num1/v_num2;
  DBMS_OUTPUT.PUT_LINE('The quotient is '||TO_CHAR(v_result));
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('You cannot divide by zero');
END;
```

Here are the results when the same values are given to the program:

```
SQL> /
Enter value for first_number: 10
Enter value for second_number: 5
The quotient is 2

PL/SQL procedure successfully completed.

SQL> /
Enter value for first_number: 15
Enter value for second_number: 0
You cannot divide by zero

PL/SQL procedure successfully completed.
```

Note that if there are no errors, the program runs until it hits the exception section, which is skipped. However, as soon as the server encounters the division by zero, the program jumps down to the exception section, skipping the first output statement. Once the error-handling statement runs, the program leaves the block. In fact, there is no way to return to the executable section from an error handler, not even using the GOTO statement. In addition, because the error is handled, control goes back to the calling environment (SQL*Plus) without an error condition, and the feedback tells you that the block successfully completed.

In the previous example, there is only one error handler, and it runs only one statement. In general, you may include more than one handler, and each may run any number of statements, although only one handler will run before the block ends. You can specify that more than one exception use the same handler, but each exception can appear in at most only one handler. Then the exception section will generally look like this:

```
EXCEPTION
  WHEN exception_name [OR exception_name ...] THEN
    statement;
    ...
  [WHEN exception_name [OR exception_name ...] THEN
    statement;
    ...]
  [WHEN OTHERS THEN
   statement;
   ...]
END;
```

While it is possible to define more than one handler in a block, doing so makes it difficult to determine what Oracle will do and can cause unpredictable execution of code. In reality, you should never define more than one exception handler in a block, and doing so is never done.

Notice that you may include the optional WHEN OTHERS clause as the last clause. This is a "catch-all" clause, much like the ELSE clause of an IF statement. If an exception is raised in the executable section of the block, control jumps to the exception section, and the program looks for the first handler associated with that error. If it does not find a specific handler for that error, then the statements in the WHEN OTHERS clause run. If there is no WHEN OTHERS clause, then the unhandled exception propagates to the calling environment.

The list of predefined exceptions and their meanings is in Table 12-1.

Table 12-1
Predefined Exceptions

Exception Name	*Oracle Error Number*	*Description*
ACCESS_INTO_NULL	ORA-06530	When objects have not been initialized, you cannot assign values to their attributes, and this exception will be raised.
COLLECTION_IS_NULL	ORA-06531	Similar to the previous exception, you cannot call methods on an uninitialized nested table or VARRAY (except EXISTS).

Exception Name	*Oracle Error Number*	*Description*
CURSOR_ALREADY_OPEN	ORA-06511	If a cursor is open and you try to open it again, this exception will be raised.
DUP_VAL_ON_INDEX	ORA-00001	You cannot insert duplicate values on a unique index, which is created by adding a PRIMARY KEY or UNIQUE constraint.
INVALID_CURSOR	ORA-01001	If you try to fetch or close a cursor that has not been declared and opened, this exception is raised.
INVALID_NUMBER	ORA-01722	You cannot convert a character string to a number if it is not numeric. This is raised by SQL statements, not PL/SQL executable statements.
LOGIN_DENIED	ORA-01017	When an attempt is made to log in to the Oracle sever with an invalid username and/or password, this exception is raised.
NO_DATA_FOUND	ORA-01403	When a SELECT...INTO statement is embedded in PL/SQL, it must fetch one row. If none are returned, this exception is raised.
NOT_LOGGED_ON	ORA-01012	This exception is raised when you try a database operation without being logged in.
PROGRAM_ERROR	ORA-06501	This exception is raised when an error internal to PL/SQL is encountered. This exception is very rare.
ROWTYPE_MISMATCH	ORA-06504	This exception is raised when passing cursor variables between a host and a stored procedure or function with different return types.
SELF_IS_NULL	ORA-30625	In objects with member methods, this exception is raised when the object is NULL and a method is called.
STORAGE_ERROR	ORA-06500	This exception is an out-of-memory error.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	For nested tables and VARRAYs, there is an upper bound on allowable index values. If you try to use a number past that bound, this exception is raised.

Continued

Table 12-1 *(continued)*

Exception Name	*Oracle Error Number*	*Description*
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Similar to preceding exception, but the index value is not even a legal value—for example, if negative numbers were used.
SYS_INVALID_ROWID	ORA-01410	This exception is raised when attempting to convert a character string that does not have the right format to a ROWID.
TIMEOUT_ON_RESOURCE	ORA-00051	Some Oracle resources have a time-out. When it expires, this exception is raised.
TOO_MANY_ROWS	ORA-01422	The SELECT...INTO in a PL/SQL statement cannot return more than one row, and this exception is raised.
VALUE_ERROR	ORA-06502	This exception is raised when an assignment statement attempts to put a value that is too large into a variable. It is also raised when a PL/SQL statement tries to convert a non-numeric string to a number.
ZERO_DIVIDE	ORA-01476	If the program attempts to perform a division by zero, this exception is raised.

Although there are 20 predefined exceptions, the ones most likely to appear on the exam are ZERO_DIVIDE, NO_DATA_FOUND, and TOO_MANY_ROWS.

Non-predefined exceptions

The majority of exceptions raised by the Oracle server are not predefined. That is, they have an error code but do not have a name associated with them. Therefore, in order to trap them in an error handler, you must declare them. The syntax for declaring an exception is similar to that for a variable:

```
DECLARE
  v_num NUMBER;
  e_child_found EXCEPTION;
```

Just as the numeric variable v_num is defined by stating its name followed by its datatype, the exception e_child_found is defined by stating its name and that it is an EXCEPTION. As the name of this exception implies, it has something to do with

the error that occurs when an attempt is made to delete a parent record in a foreign key relationship. However, this exception will not automatically be associated with that error because of the name that you have chosen for it. You must explicitly associate that exception name with the Oracle error number. Then when Oracle raises that error number, you will be able to trap the exception using the name you have associated with it.

The syntax for associating an exception name to an Oracle error number involves a compiler directive, or pragma. A pragma is followed by the compiler but is ignored at runtime. The following examples show the use of the pragma called EXCEPTION_INIT, which initializes (or associates) an exception with an error number. In the first case, the referential integrity constraint error ORA-02292 is unhandled. In the second case, it is associated with the exception named v_child_found:

```
DECLARE
  v_num Courses.CourseNumber%TYPE := &number;
BEGIN
  DELETE FROM Courses
  WHERE  CourseNumber = v_num;
  DBMS_OUTPUT.PUT_LINE('Course has been deleted');
END;
```

Here is the output from an execution where the course number to be deleted is in use in the ScheduledClasses table:

```
SQL> /
Enter value for number: 100
DECLARE
*
ERROR at line 1:
ORA-02292: integrity constraint
(STUDENT.FK_SCHEDCLASS_COURSENUM) violated - child record found
ORA-06512: at line 4
```

Now the error will be trapped by declaring an exception and associating it with the error number –2292:

```
DECLARE
  v_num Courses.CourseNumber%TYPE := &number;
  e_child_found EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_child_found, -2292);
BEGIN
  DELETE FROM Courses
  WHERE  CourseNumber = v_num;
  DBMS_OUTPUT.PUT_LINE('Course has been deleted');
EXCEPTION
  WHEN e_child_found THEN
    DBMS_OUTPUT.PUT_LINE('Course has already been scheduled');
    DBMS_OUTPUT.PUT_LINE('It cannot be deleted');
END;
```

The output is now as follows:

```
SQL> /
Enter value for number: 100
Course has already been scheduled
It cannot be deleted

PL/SQL procedure successfully completed.
```

In order to decide which errors you are going to associate with exceptions, you must fully test your program for all possible situations. From these tests, you should be able to capture the error codes of the form ORA-XXXXX and then decide what should be done in each case. Remember that for some of these codes, the names are predefined, but for the rest of them, you have to create and associate exception names. The exception name is associated to just the numeric part of the code, which is of the form -XXXXX. This is a negative integer, so leading zeros need not be included. Thus, the ORA-02292 error can be trapped with the number -02292 or -2292.

User-defined exceptions

The last type of exception is one that is not automatically raised by the Oracle server. It represents some sort of error according to your application's business rules. It must be raised within the program using the RAISE statement. For example, the ClassEnrollment table has a column called Status. One business rule may be that an enrollment may not be deleted if it has a status of "Confirmed". This is not a problem for the server, because no referential integrity (foreign key) constraints are being violated, but it should be treated as an error in the application. Here is the example:

```
DECLARE
  v_class   ClassEnrollment.ClassID%TYPE := &class;
  v_student ClassEnrollment.StudentNumber%TYPE := &student;
  v_status  ClassEnrollment.Status%TYPE;
  e_confirmed EXCEPTION;
BEGIN
  SELECT Status
  INTO   v_status
  FROM   ClassEnrollment
  WHERE  ClassID = v_class
  AND    StudentNumber = v_student;

  IF v_status = 'Confirmed' THEN
    RAISE e_confirmed;
  END IF;

  DELETE FROM ClassEnrollment
  WHERE  ClassID = v_class
  AND    StudentNumber = v_student;
```

```
  DBMS_OUTPUT.PUT_LINE('This enrollment has been deleted');
EXCEPTION

 WHEN e_confirmed THEN
  DBMS_OUTPUT.PUT_LINE('This enrollment is already confirmed');
  DBMS_OUTPUT.PUT_LINE('It cannot be deleted');
END;
```

The output for two executions, one with status "Hold" and one with status "Confirmed" is as follows:

```
SQL> /
Enter value for class: 53
Enter value for student: 1003
This enrollment has been deleted

PL/SQL procedure successfully completed.

SQL> /
Enter value for class: 51
Enter value for student: 1008
This enrollment is already confirmed
It cannot be deleted

PL/SQL procedure successfully completed.
```

Note that the user-defined exception is declared in the same way as the unpredefined exception, but that it is not associated with an actual Oracle error number. Therefore, it is never raised without the RAISE statement. The RAISE statement can also be used to raise Oracle errors, whether predefined or not. Here is an example that raises the NO_DATA_FOUND error if an UPDATE statement fails to alter any rows. Generally, the NO_DATA_FOUND error is raised automatically by the server when a SELECT statement retrieves no rows, but you may want to use the same handler for both:

```
BEGIN
  SELECT ...
  UPDATE ...
  IF SQL%NOTFOUND THEN
    RAISE NO_DATA_FOUND;
  END IF;
  ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
   ...
END;
```

This example then does the same error handling whether it is the SELECT or the UPDATE statement that affects no rows.

WHEN OTHERS Clause

Objective

✦ Trap unanticipated errors

Even with rigorous testing, you cannot anticipate every error that can arise in the use of your program. You need to decide which ones you will build specific handlers for, and for the rest, you have two choices: build a generic handler for all other errors using the WHEN OTHERS clause, or let the error propagate to the calling environment.

The WHEN OTHERS clause is sometimes necessary to provide error-handling statements for any error situation. Some examples of things that are commonly done include rolling back transactions and closing all open cursors before leaving the block. Note that if it is to be used, the WHEN OTHERS clause must be the last of the error handlers in the exception section. The problem with the WHEN OTHERS clause is that sometimes poor programmers will fall into the trap of doing the following:

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    ...
  WHEN TOO_MANY_ROWS THEN
    ...
  WHEN OTHERS THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Some error occurred');
END;
```

If you output such a message, you must be prepared to receive support calls like the following:

"Something went wrong with my application."

"What does the message say?"

"Some error occurred."

Unless you are going to provide further information in the message, or possibly in some error table, you will not have any clue what went wrong in the application. You can use the functions SQLCODE and SQLERRM to return the error number and text within your exception handler. You can then either print these out to the screen or write them to an error table. This example illustrates the latter:

```
DECLARE
  v_code NUMBER;
  v_message VARCHAR2(255);
  v_num1  NUMBER := &first_number;
  v_num2  NUMBER := &second_number;
  v_result NUMBER;
```

```
BEGIN
  v_result := v_num1/v_num2;
  DBMS_OUTPUT.PUT_LINE(v_result);
EXCEPTION
  WHEN OTHERS THEN
    v_code := SQLCODE;
    v_message := SUBSTR(SQLERRM,1,255);
    DBMS_OUTPUT.PUT_LINE('Some error occurred');
    INSERT INTO Errors(UserName, Code, Message)
    VALUES(USER, v_code, v_message);
END;
```

When executed, this block prompts for two numbers, and if the second is zero, then it fails with the ZERO_DIVIDE exception. Here is what the output looks like, as well as the new row in the Errors table:

```
SQL> /
Enter value for first_number: 15
Enter value for second_number: 0
Some error occurred

PL/SQL procedure successfully completed.

SQL> SELECT *
  2  FROM   Errors;

USERNAME                   CODE MESSAGE
-------------------- ---------- ------------------------------
STUDENT                   -1476 ORA-01476: divisor is equal to
                                zero
```

Most of the Oracle errors result in a code that is simply the –XXXX portion of the error number of the form ORA-XXXXX. There are some exceptions to this rule, including the NO_DATA_FOUND error, which returns +100. All user-defined exceptions return a SQLCODE of +1 with the SQLERRM "User-Defined Exception". If there is no error, then SQLCODE returns zero, and the message is "ORA-0000: normal, successful completion".

Note that you cannot insert the values SQLCODE and SQLERRM directly into a SQL command. You must first assign their values to local variables, and with SQLERRM, it is a good idea to use the substring function to avoid any value errors because it is a LONG datatype.

Error Propagation

Objective

- ✦ Describe the effect of exception propagation in nested blocks

In SQL*Plus, unhandled exceptions are simply printed to the screen, but what if the calling environment for the PL/SQL block is not SQL*Plus? If the block is nested

within another PL/SQL block, the results are somewhat different. For example, in the following block, an error is raised in the sub-block but is not handled:

```
DECLARE
 x NUMBER;
BEGIN
 x := 10;
 BEGIN
  x := 20;
  RAISE INVALID_NUMBER;
 END;
 DBMS_OUTPUT.PUT_LINE(x);
END;
```

When the program looks for an exception handler, it first looks at the inner block; then control passes out to the outer block, where it looks for a handler for INVALID_NUMBER. If it finds no handler there, only then does it pass to the calling environment:

```
SQL> /
DECLARE
*
ERROR at line 1:
ORA-01722: invalid number
ORA-06512: at line 7
```

Now, if there were an exception handler in the outer block, it would trap the error:

```
DECLARE
 x NUMBER;
BEGIN
 x := 10;
 BEGIN
  x := 20;
  RAISE INVALID_NUMBER;
 EXCEPTION
  WHEN INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('error handled');
 END;
 DBMS_OUTPUT.PUT_LINE(x);
END;
```

With the error trapped in the inner block, control then passes to the next executable line after that block ends. Therefore, the value of x is printed:

```
SQL> /
error handled
20

PL/SQL procedure successfully completed.
```

This flow of control is one of the main reasons why blocks are nested within one another. When all statements are placed in just one block and an exception is raised, the exception handler is processed, and then the program must end. When certain error-prone statements are placed in a nested block, only the inner block needs to end, so the program can continue.

The other main reason that nested blocks are used is so that error handling can be tailored to a particular section of code. Consider the following code:

```
BEGIN
  SELECT ...
  SELECT ...
  SELECT ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    ...
END;
```

In this example, no matter which of the three SELECT statements raises the NO_DATA_FOUND exception, the one exception handler will run its statements. What if you want different statements to run depending on which one fails? This next example will show how each one can differ:

```
BEGIN

  BEGIN
    SELECT ...
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      ...
  END;

  BEGIN
    SELECT ...
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      ...
  END;

  SELECT ...
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    ...
END;
```

This provides three different exception handlers. This block will work slightly differently than the earlier one if the error is in the first or second sub-blocks. This is because once they handle the error, they will continue with the flow of the main block. If you were to re-raise the exception from one of the handlers, it would also

propagate the error to the outer block, which can sometimes be a useful technique. In the example in Figure 12-1, nested block exception handlers take care of the message output to the screen, then re-raise the current exception to propagate it to the outer block, which logs the error.

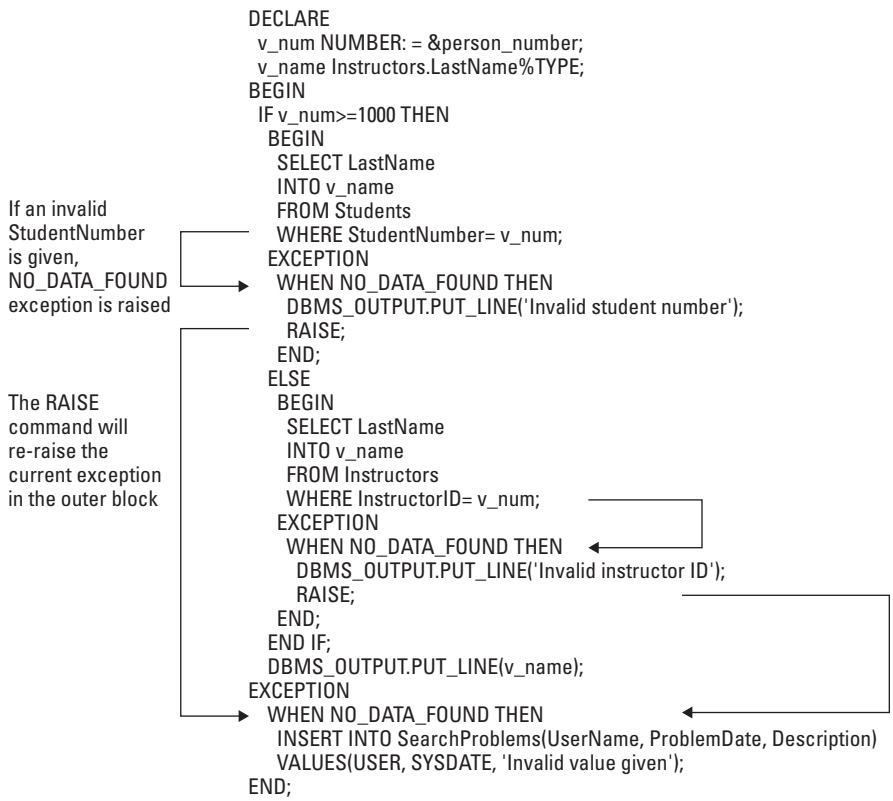


Figure 12-1: An example of error propagation

Note that the syntax to re-raise the current exception from a handler is simply the RAISE statement with no exception name given. Following are two successful and two unsuccessful executions of the code, along with the entries in the SearchProblems table. This block takes advantage of the fact that student numbers start at 1,000 and go up, while all of the instructor IDs are less than 1,000:

```
SQL> /
Enter value for person_number: 1001
Jones
```

```
PL/SQL procedure successfully completed.

SQL> /
Enter value for person_number: 300
Harrison

PL/SQL procedure successfully completed.

SQL> /
Enter value for person_number: 1302
Invalid student number

PL/SQL procedure successfully completed.

SQL> /
Enter value for person_number: 122
Invalid instructor ID

PL/SQL procedure successfully completed.

SQL> SELECT *
  2  FROM   SearchProblems;

USERNAME             PROBLEMDATE DESCRIPTION
-------------------- ----------- --------------------

STUDENT              20-FEB-01   Invalid value given

STUDENT              20-FEB-01   Invalid value given
```

From an exception handler, you may also RAISE a different exception by using the syntax:

```
RAISE exception_name;
```

This error immediately propagates to the exception section of the outer block, regardless of whether or not a handler for this exception appears later in the same exception section. Similarly, exceptions raised in the declare section of a block cannot be handled within the exception section of the same block.

Because different calling environments deal with unhandled exceptions in different ways, you may want to avoid going back to the calling environment with an unhandled exception. The alternative, if you want the calling application to receive some sort of error message, is to use the RAISE_APPLICATION_ERROR procedure from the DBMS_STANDARDS package. It enables you to send a user-defined message back to the calling environment in a way that is consistent with Oracle errors. That is, if the calling environment can trap exceptions, then it will be able to trap this user-defined error. The following three examples are similar in nature but illustrate

the different ways in which to propagate an error to the calling environment. The first simply raises an unhandled exception, while the second handles that exception but also propagates it to the calling environment. The third makes use of RAISE_APPLICATION_ERROR to propagate a user-defined message to the calling environment:

```
SQL> DECLARE
  2    x NUMBER;
  3  BEGIN
  4    x := 10/0;
  5* END;
SQL> /
DECLARE
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 4

SQL> DECLARE
  2    x NUMBER;
  3  BEGIN
  4    x := 10/0;
  5  EXCEPTION
  6    WHEN ZERO_DIVIDE THEN
  7      DBMS_OUTPUT.PUT_LINE('Cannot divide by zero');
  8      RAISE;
  9  END;
 10  /
Cannot divide by zero
DECLARE
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at line 8

  1  DECLARE
  2    x NUMBER;
  3  BEGIN
  4    x := 10/0;
  5  EXCEPTION
  6   WHEN ZERO_DIVIDE THEN
  7    RAISE_APPLICATION_ERROR(-20001,'Cannot divide by zero');
  8* END;
  9  /
DECLARE
*
ERROR at line 1:
ORA-20001: Cannot divide by zero
ORA-06512: at line 7
```

You will notice that, in the final example, the error message output in the calling environment was user defined. It was also given an error number –200001. In the RAISE_APPLICATION_ERROR procedure, the range of user-defined error numbers is –20000 to –20999. This means that you have a thousand error numbers that you can use.

Some companies use the range of 1,000 user-defined error numbers to set corporate standards for how applications pass control back to their calling programs. Keeping a library of what each number means can simplify the error handling in the calling programs.

The RAISE_APPLICATION_ERROR procedure also has an optional third argument. In addition to passing the user-defined error number and message, you can pass a BOOLEAN (TRUE or FALSE) to specify whether the user-defined message should be placed on the existing error stack. The default is FALSE, so it replaces the stack. Here is the last example with the error placed on the stack:

```
SQL> DECLARE
  2    x NUMBER;
  3  BEGIN
  4    x := 10/0;
  5  EXCEPTION
  6    WHEN ZERO_DIVIDE THEN
  7      RAISE_APPLICATION_ERROR(-20001,
  8          'Cannot divide by zero', TRUE);
  9* END;
SQL> /
DECLARE
*
ERROR at line 1:
ORA-20001: Cannot divide by zero
ORA-06512: at line 7
ORA-01476: divisor is equal to zero
```

Here the user-defined error is shown, in addition to the regular ZERO_DIVIDE error message, because the ZERO_DIVIDE error was raised within the program. In general, the RAISE_APPLICATION_ERROR procedure can be called from executable sections as well as exception sections, although no previous errors in the stack are displayed in those cases.

Coding Conventions

✦ Use coding conventions

Many different coding conventions are used in exception handling. One convention, discussed previously, is to maintain a library of user-defined error messages and numbers for use in RAISE_APPLICATION_ERROR calls. Then when calling a

subprogram developed by someone else, you can easily decide how to handle the errors that are returned by that subprogram, because you know what the possible outcomes are.

Another popular coding convention is to create a generic error-handling routine. Instead of coding in every single block what to do when, say, a ZERO_DIVIDE exception is raised, why not build WHEN OTHERS clauses that simply pass the SQLCODE and SQLERRM as parameters to a stored procedure, which will decide what to do if it is a ZERO_DIVIDE, or some other exception?

The use of an error table that logs the username, date of the error, name of the offending application, along with possibly the line number that caused the error, and the SQLCODE and SQLERRM, is another way in which some developers deal with errors. This can serve to shield the users from confusing error messages, while the support team can investigate the error table to see what really happened.

As for the naming conventions for non-predefined or user-defined exceptions, many people use e_name, which differentiates them from local variables that may begin with another prefix, such as v_.

Key Point Summary

In PL/SQL, a number of different errors may arise at different points in the programming process. Some of these are errors in logic, while some are syntax errors, and others are just unavoidable. The unavoidable errors that occur at runtime can be handled through exceptions.

- ✦ PL/SQL is a compiled language. That means that it must be compiled before code can execute, but if syntax errors exist in the code, then it will not compile. If a block cannot compile, you can find a listing of those compile errors and fix them.
- ✦ Once a PL/SQL block of code is compiled and executed, it may appear to run without error, but the program logic may contain errors that make it produce the wrong results. These are called *bugs*, and the process of searching for them is called *debugging*. There is little support for debugging in SQL*Plus, but other tools may be better.
- ✦ Even when a program is compiled and bug-free, runtime errors may still arise. Your program can run out of memory or invalid data can be provided from another application, the database, or from the user, and so on. These errors generally raise an exception. You can specify what to do in the event that these errors come up by coding handlers in the exception section.
- ✦ Some exceptions have a predefined name and are raised by the server. In order to trap these errors, you simply use that name in the handler in the exception section.

- ✦ The vast majority of errors raised by the server do not have a name, so they are known as non-predefined exceptions. To trap these errors, you must declare an exception and use the pragma (compiler directive) EXCEPTION_INIT to associate the name you have declared with the Oracle error number that the exception produces.
- ✦ Sometimes you may want to raise an exception, even though the server has not encountered an error. For this purpose, you can declare an exception and explicitly raise it by calling the RAISE statement. You then trap it just like any Oracle error.
- ✦ If errors are not trapped in the exception section of the block in which they are raised, they propagate to the calling environment. If the calling environment is another block (so the error occurred in a sub-block), control passes to the exception section of the outer block, looking for a handler for the error. However, if the error is handled in the sub-block, control passes to the next line after the sub-block, and the outer block carries on without error.
- ✦ If an exception is propagated to a calling environment such as SQL*Plus, then the error code and message will simply be output to the screen.
- ✦ You can propagate your own user-defined error messages to the calling environment from anywhere within a block by calling the RAISE_APPLICATION_ERROR procedure.

✦ ✦ ✦

STUDY GUIDE

The following questions will help you assess your understanding of the different types of errors and exceptions in PL/SQL. They will also test your knowledge of the concept of error propagation.

Assessment Questions

1. Which of the following statements are true of error handling in PL/SQL? (Choose the best answer.)

A. Error handling is done in-line; you must test for an error after each executable line that could produce an error.

B. Predefined exceptions do not need to be declared in each block that they are used.

C. Non-predefined exceptions are associated with error numbers using the ASSOC_EXCEPTION pragma.

D. User-defined exceptions are raised automatically by the Oracle server.

E. User-defined exceptions are the only type of exceptions that can be raised using the RAISE statement.

2. Consider the following exception section:

```
EXCEPTION
  WHEN NO_DATA_FOUND
    DBMS_OUTPUT.PUT_LINE('Invalid value');
  WHEN OTHERS THEN
    ROLLBACK;
    RAISE;
  WHEN TOO_MANY_ROWS
    DBMS_OUTPUT.PUT_LINE('More than one match');
END;
```

Which of the following are errors in this exception section? (Choose two answers.)

A. The RAISE statement must include an exception name.

B. Each handler must include the word THEN before the statements that are to be run.

C. The WHEN OTHERS clause must be the last handler.

D. The ROLLBACK statement is not allowed in the exception section.

3. Evaluate this PL/SQL block:

```
DECLARE
  v_num Courses.CourseNumber%TYPE;
BEGIN
  SELECT CourseNumber
  INTO   v_num
  FROM   Courses
  WHERE  CourseName LIKE '%&p_name%';
EXCEPTION
  WHEN OTHERS THEN
    INSERT INTO Errors
    VALUES(USER, SYSDATE, SQLCODE, SQLERRM);
END;
```

What is the result when this block is executed? (Choose the best answer.)

A. If the partial course name provided does not match one in the database, a new row is added to the Errors table for the NO_DATA_FOUND exception.

B. If the partial course name matches more than one course in the database, then a new row is added to the Errors table for the TOO_MANY_ROWS exception.

C. The block won't execute because of a compile error — SQLCODE and SQLERRM cannot be used in the INSERT statement.

D. A and B.

E. None of the above.

4. Consider the following PL/SQL block:

```
BEGIN
  BEGIN
    INSERT ...
    SELECT ...
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Missing or invalid data');
  END;
  COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    ROLLBACK;
END;
```

What is the result if the SELECT statement returns no rows? (Choose two answers.)

A. The DBMS_OUTPUT.PUT_LINE statement is executed.

B. The inserted row is rolled back due to the ROLLBACK statement in the outer block exception section.

C. The inserted row is rolled back because the block ends without committing the change.

D. The inserted row is committed.

E. The DBMS_OUTPUT.PUT_LINE statement is not executed.

5. Consider the following PL/SQL block:

```
BEGIN
  BEGIN
    INSERT ...
    SELECT ...
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE('Missing or invalid data');
      RAISE;
  END;
  COMMIT;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    ROLLBACK;
END;
```

What is the result if the SELECT statement returns no rows? (Choose the best answer.)

A. The DBMS_OUTPUT.PUT_LINE statement is executed.

B. The inserted row is rolled back due to the ROLLBACK statement in the outer block exception section.

C. The inserted row is rolled back because the block ends without committing the change.

D. The inserted row is committed.

E. A and B.

6. Given this PL/SQL block:

```
DECLARE
  v_error_code NUMBER;
BEGIN
  DECLARE
    e_no_rows EXCEPTION;
  BEGIN
    INSERT ...
    UPDATE ...
    IF SQL%NOTFOUND THEN
      RAISE e_no_rows;
    END IF;
    DBMS_OUTPUT.PUT_LINE...
```

```
  EXCEPTION
    WHEN e_no_rows THEN
      ROLLBACK;
      v_error_code := SQLCODE;
      INSERT INTO Errors
      VALUES(USER, SYSDATE, v_error_code);
      RAISE ZERO_DIVIDE;
    WHEN OTHERS THEN
      v_error_code := SQLCODE;
      INSERT INTO Errors
      VALUES(USER, SYSDATE, v_error_code);
  END;
  SELECT ...
  RAISE_APPLICATION_ERROR(-20001, 'Error occurred');
EXCEPTION
  WHEN OTHERS THEN
    v_error_code := SQLCODE;
    INSERT INTO Errors
    VALUES(USER, SYSDATE, v_error_code);
END;
```

If the UPDATE statement in the second executable line of the outer block fails to affect any rows, what value(s) of error number(s) will be inserted into the Errors table? (Choose the best answer.)

A. –1476

B. +1

C. –20001

D. A and B.

E. A, B, and C.

7. Which of the following exception declarations is legal? (Choose the best answer.)

A. DECLARE
e_foreign_key ERROR;
PRAGMA EXCEPTION_ASSOC(e_foreign_key, -2292);

B. DECLARE
e_foreign_key EXCEPTION;
PRAGMA INIT_EXCEPTION(-2292, e_foreign_key);

C. DECLARE
e_foreign_key ERROR;
EXCEPTION_INIT(e_foreign_key, -2292);

D. DECLARE
e_foreign_key EXCEPTION;
PRAGMA EXCEPTION_INIT(e_foreign_key, -2292);

E. None of the above.

8. From the exception section of a block, once you have handled an error, how can you ensure that the block will not end and return to the calling environment without an error condition? (Choose the best answer.)

A. Re-raise the same exception using the RAISE statement.

B. Raise a different error using the RAISE statement.

C. Raise a user-defined error by calling the RAISE_APPLICATION_ERROR procedure.

D. All of the above.

E. None of the above.

9. Consider the following exception section:

```
EXCEPTION
  WHEN NO_DATA_FOUND OR TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE(
       'Problem with the query, retrying');
    BEGIN
      SELECT ...
    EXCEPTION
      WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE('Nothing found');
    END;
  WHEN TOO_MANY_ROWS THEN
    DBMS_OUTPUT.PUT_LINE('Be more specific');
END;
```

What, if anything, is illegal with this code? (Choose the best answer.)

A. You cannot make one handler for both NO_DATA_FOUND or TOO_MANY_ROWS.

B. You cannot have two handlers in the same block for the TOO_MANY_ROWS exception.

C. You cannot nest a sub-block with its own exception section in an exception handler.

D. Nothing is wrong; it will run without problems.

10. Evaluate this PL/SQL block:

```
BEGIN
  DECLARE
     e_my_err EXCEPTION;
  BEGIN
    RAISE e_my_err;
  EXCEPTION
    WHEN e_my_err THEN
      RAISE;
  END;
EXCEPTION
  WHEN e_my_err THEN
    DBMS_OUTPUT.PUT_LINE('Error handled');
END;
```

What, if anything, is illegal within this block? (Choose the best answer.)

A. The RAISE statement cannot be the first statement in a block.

B. You cannot use the RAISE statement to re-raise a user-defined exception.

C. The exception called e_my_err is out of scope in the outer block exception section.

D. Nothing is wrong.

Scenarios

1. You are in charge of a PL/SQL application that has grown over time. Initially, you developed it and were the only support person for a small number of well-behaved users. Unexpected errors came up from time to time, and you simply printed these errors to a console line at the bottom of the user's screen. The user noted the error and got in touch with you immediately to find out what to do. As the application grows in complexity and number of users, this situation is becoming increasingly more difficult to manage. Some users close the application without noting the error message, and some do not even report that anything has happened. What might you do to shield your users from this error burden, while still enabling a support team to discover what errors are taking place in the system?
2. You are the head of an IT department that has several small teams working on different PL/SQL applications. These teams work independently of one another, but the applications that make up the entire working system must interface with one another at many points. Over time, these teams have developed different standards for coding, but this is generally not a problem, with the one notable exception of error handling. When a PL/SQL block from one application makes a call to another application, the developers are never sure what to expect back in an error situation. How might you standardize this process?

Lab Exercises

Lab 12–1 Trap a predefined exception

1. Sign on to SQL*Plus as user "student" with password "oracle".
2. Create a SQL*Plus script file called FindCourse. Create an ACCEPT statement to retrieve a number named p_num. **This program is going to print to the screen the name of the course with that CourseNumber.**
3. Create a PL/SQL block of code with a declare section, which declares a variable called v_name to hold the CourseName from the Courses table.

4. Use the SELECT ... INTO statement to retrieve the course name into v_name.
5. Print the value of v_name to the screen using DBMS_OUTPUT.PUT_LINE. Hint: Make sure that the SQL*Plus environment variable SERVEROUTPUT is set ON.
6. Add an exception handler for the case when an invalid course number is given, and print a suitable message.
7. Save your script file and then execute it using the START command in SQL*Plus. Try a variety of values, including valid and invalid course numbers.

Lab 12–2 Trap a non-predefined exception

1. Sign on to SQL*Plus as user "student" with password "oracle".
2. Create a SQL*Plus script file called DeleteCourse. Create an ACCEPT statement to retrieve a number named p_num. **This program is going to delete the course with that CourseNumber, if it is able to. If the course has already been scheduled, it will return a message to that effect.**
3. Create a PL/SQL block of code with a declare section. Declare an exception e_scheduled.
4. Associate the exception e_scheduled with the Oracle Error ORA-02292, which is the error that comes up when you attempt to delete a parent record with existing child records in a foreign key relationship.
5. Attempt to delete the course. If successful, print a message to say so using the DBMS_OUTPUT.PUT_LINE procedure.
6. Add an exception handler for the case when the course has been scheduled and the delete fails. Print a suitable message.
7. Save your script file and then execute it using the START command in SQL*Plus. Try a variety of values, including scheduled and unscheduled courses. Make sure to ROLLBACK your changes from the command line after each test, so that your database will still have data for future tests.

Lab 12–3 Trap a user-defined exception

1. Sign on to SQL*Plus as user "student" with password "oracle".
2. Create a SQL*Plus script file called UpdateCourse. Create two ACCEPT statements: one to retrieve a number named p_num, the other to retrieve a character string v_name. **This program is going to update the course specified by that CourseNumber, giving it a new name.**
3. Create a PL/SQL block of code with a declare section. Declare an exception e_not_found.
4. Attempt to update the course. Test for failure using SQL%NOTFOUND. If your program does not find the row to update, then raise your error. If successful, print a message to say so using the DBMS_OUTPUT.PUT_LINE procedure.

5. Add an exception handler for the case when the course number is invalid and have it print a suitable message.
6. Save your script file and then execute it using the START command in SQL*Plus. Try a variety of values for the course number, including valid and invalid numbers. Make sure to ROLLBACK your changes from the command line after each test, so that your database will still have data for future tests.

Answers to Chapter Questions

Chapter Pre-Test

1. When the compiler prepares a block of PL/SQL code, it has to check that the syntax of all the statements is correct, that all references to database objects are legal, and that all identifiers have been declared. Failure of any of these is a compile error, and with anonymous blocks, these errors are automatically output to the screen when you attempt to execute the block.
2. The three types of exceptions in PL/SQL are predefined, non-predefined, and user-defined exceptions.
3. Predefined and non-predefined exceptions are raised automatically by the Oracle server when it encounters an error condition. User-defined exceptions must be raised explicitly with the RAISE statement. The RAISE statement can also be used to raise predefined and non-predefined Oracle errors as well.
4. If a block has no exception section, then all exceptions raised within that block are propagated to the calling environment.
5. An exception handler traps a particular error and contains statements that are run when that error is raised within that block. It stops the error from propagating to the calling environment.
6. The WHEN OTHERS clause is a generic error handler that traps any exceptions from the block that do not have specific exception handlers. It ensures that the block will not return to the calling environment with any unhandled exceptions.
7. The functions SQLCODE and SQLERRM return the number and message text of the current exception. They can be placed into local variables and then possibly inserted into an errors table or output to the user.
8. The pragma (which is a compiler directive) called EXCEPTION_INIT takes two arguments — a user-declared exception name and an Oracle error number — and associates the name with the number.
9. Passing a user-defined error number between –20000 and –20999, as well as the text for an error message to the RAISE_APPLICATION_ERROR procedure, causes a PL/SQL block to return to the calling environment with an exception. That exception has the specified number and error message.

10. If you have a group of PL/SQL statements that need their own error handling that is separate from that done in the exception section from the block, then you may choose to nest those statements in a sub-block with its own exception section. You have to decide whether the program should continue to function after the inner block deals with error situations, or whether the error should be propagated to the exception section of the outer block for further error handling.

Assessment Questions

1. **B** — Predefined exceptions are just that — predefined. You do not need to define them again within each block. Error handling is not done in-line, but instead the program automatically jumps to the exception section when there is an error. Non-predefined exceptions are associated with a number using the EXCEPTION_INIT pragma. User-defined exceptions can be raised only using the RAISE statement, although that statement can be used to raise any error. Refer to the section "Types of Errors," earlier in this chapter.
2. **B, C** — The RAISE statement without an exception name simply re-raises the current exception, so this is not an error. The THEN statement is missing from the handlers, and WHEN OTHERS must be the last handler. Transaction controls such as ROLLBACK are enabled in any executable lines of code, including those in the exception section. Refer to the "Exception Handling" section.
3. **C** — The functions SQLCODE and SQLERRM are not allowed in SQL statements, only in PL/SQL executable statements. Therefore, block will not even compile. Refer to the "WHEN OTHERS Clause" section.
4. **A, D** — With the SELECT failing in the inner block, only the inner block's exception handler will run, so the DBMS_OUPUT.PUT_LINE statement will run. Then control passes out to the next line after the inner block, so the insert is committed and the program ends. Refer to the "Exception Handling" section.
5. **E** — The SELECT statement fails, causing the inner block exception handler for NO_DATA_FOUND to execute the DBMS_OUTPUT.PUT_LINE statement. It then executes a RAISE statement, which raises another NO_DATA_FOUND exception. This one is handled by the outer block exception section, so the inserted row is rolled back. Refer to the "Error Propagation" section.
6. **D** — In this block, if the UPDATE affects no rows, then the SQL%NOTFOUND will return TRUE and the user-defined exception will be raised. Part of the handling of this exception is to insert into the table the error number from SQLCODE, which will be +1 for user-defined exceptions. Then the ZERO_DIVIDE exception is raised, but this immediately propagates to the exception section of the outer block. The WHEN OTHERS handler will insert the SQLCODE –1476 for the ZERO_DIVIDE error, and the program ends. Refer to the "Error Propagation" section.
7. **D** — The pragma to associate a name to an error number is EXCEPTION_INIT, and the first argument is the previously declared name of type EXCEPTION, not ERROR. Refer to the "Non-predefined Exceptions" part of the "Exception Handling" section.

8. **D** — Any exception raised in a handler will immediately transfer control out to the calling environment and look for an exception handler. Similarly, the RAISE_APPLICATION_ERROR procedure is designed to go back to the calling environment with an error. Refer to the "Error Propagation" section.
9. **B** — It is legal to make one handler for more than one exception, but no exception can appear in more than one handler in the same block. Handlers in subblocks are allowed, even in the exception section. Refer to the "Exception Handling" section.
10. **C** — The user-defined exception e_my_err is declared in the inner block, so it cannot be used in the outer block, but it can be re-raised from the inner block. The RAISE statement can be used anywhere that a PL/SQL executable line of code is allowed. Refer to the section "Exception Handling," earlier in this chapter.

Scenarios

1. The best way to shield users from the errors that occur is to not print them to the screen. This generally means that you do not want any unhandled exceptions creeping into the PL/SQL calling environment. This goal is accomplished by adding a WHEN OTHERS clause to each PL/SQL block called from the application. In order to keep track of the exceptions that are raised, it might be prudent to use the SQLCODE and SQLERRM functions to get exception information and insert it into some sort of errors table, along with the username, error date, and possibly the application name. Your support team can periodically check this table, or you can use database alerts in an INSERT trigger on the table to tell you when something is added.
2. One way in which you might standardize the errors returned by calls is to enable only top-level application code (that which may be called from another application) to return errors via the RAISE_APPLICATION_ERROR procedure. Then you maintain a library of the error numbers (in the range –20000 to –20999) and messages that are sent. Whenever a developer makes a call to another application, he or she can look in this library to see which exceptions the developer may possibly have to build handlers for in the calling programs.

Lab Exercises

Lab 12–1 Trap a predefined exception

The whole script file should look something like this:

```
SET SERVEROUTPUT ON
ACCEPT p_num NUMBER PROMPT "Please enter the course number:
"DECLARE
  v_name Courses.CourseName%TYPE;
```

```
BEGIN
  SELECT CourseName
  INTO   v_name
  FROM   Courses
  WHERE  CourseNumber = &p_num;
  DBMS_OUTPUT.PUT_LINE('The name of the course is:');
  DBMS_OUTPUT.PUT_LINE(v_name);
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('Invalid course number');
END;
/
```

Here are the sample executions:

```
SQL> START FindCourse
Please enter the course number: 99
Invalid course number

PL/SQL procedure successfully completed.

SQL> START FindCourse
Please enter the course number: 100
The name of the course is:
Basic SQL

PL/SQL procedure successfully completed.
```

Lab 12–2 Trap a non-predefined exception

The whole script file should look something like this:

```
SET SERVEROUTPUT ON
ACCEPT p_num NUMBER PROMPT "Please enter the course number: "

DECLARE
  e_scheduled EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_scheduled, -2292);
BEGIN
  DELETE FROM Courses
  WHERE  CourseNumber = &p_num;
  DBMS_OUTPUT.PUT_LINE('The course has been deleted');
EXCEPTION
  WHEN e_scheduled THEN
    DBMS_OUTPUT.PUT_LINE('The course has already been
scheduled');
    DBMS_OUTPUT.PUT_LINE('It cannot be deleted');
END;
/
```

Here is the output from two executions, along with the contents of the Courses table after the one row is deleted:

```
SQL> START DeleteCourse
Please enter the course number: 320
The course has been deleted

PL/SQL procedure successfully completed.

SQL> START DeleteCourse
Please enter the course number: 200
The course has already been scheduled
It cannot be deleted

PL/SQL procedure successfully completed.

SQL> SELECT CourseNumber, CourseName
  2  FROM   Courses;

COURSENUMBER COURSENAME
------------ ----------------------------------------
         100 Basic SQL
         110 Advanced SQL
         201 Performance Tuning your Database
         200 Database Performance Basics
         210 Database Administration
         220 Backing up your database
         300 Basic PL/SQL
         310 Advanced PL/SQL

8 rows selected.
```

Lab 12–3 Trap a user-defined exception

The whole script file should look something like this:

```
SET SERVEROUTPUT ON
ACCEPT p_num NUMBER PROMPT "Please enter the course number: "
ACCEPT p_name PROMPT "Please enter the new course name: "

DECLARE
  e_not_found EXCEPTION;
BEGIN
  UPDATE Courses
  SET    CourseName = '&p_name'
  WHERE  CourseNumber = &p_num;
  IF SQL%NOTFOUND THEN
    RAISE e_not_found;
  END IF;
  DBMS_OUTPUT.PUT_LINE('The course has been updated');
```

```
EXCEPTION
  WHEN e_not_found THEN
    DBMS_OUTPUT.PUT_LINE('Invalid course number');
END;
/
```

Here is the output, along with the change in the database table:

```
SQL> START UpdateCourse
Please enter the course number: 210
Please enter the new course name: DBA I
The course has been updated

PL/SQL procedure successfully completed.

SQL> START UpdateCourse
Please enter the course number: 99
Please enter the new course name: Intro to SQL
Invalid course number

PL/SQL procedure successfully completed.

SQL> SELECT CourseNumber, CourseName
  2  FROM   Courses;

COURSENUMBER COURSENAME
------------ ----------------------------------------
         100 Basic SQL
         110 Advanced SQL
         201 Performance Tuning your Database
         200 Database Performance Basics
         210 DBA I
         220 Backing up your database
         300 Basic PL/SQL
         310 Advanced PL/SQL
         320 Using your PL/SQLskills

9 rows selected.
```

CHAPTER 13

Introduction to Stored Programs

CHAPTER PRE-TEST

1. Can you store a PL/SQL program in the database?
2. What is the difference between a PL/SQL procedure and a PL/SQL function?
3. What is the purpose of parameters?
4. What is the difference between a statement trigger and a row-level trigger?
5. Name the advantages of putting PL/SQL programs in a package.
6. Which data dictionary views provide information about PL/SQL programs stored in the database?
7. What are the two parts of a PL/SQL package?
8. What is the difference between IN and OUT parameters?
9. Can you call a PL/SQL program from within a SQL statement?
10. Can you access the values being changed in a database trigger?

✦ Answers to these questions can be found at the end of the chapter. ✦

The material covered in this chapter is not covered in the "Introduction to Oracle: SQL and PL/SQL" exam. The material presented here is provided so that you can apply your PL/SQL coding skills in stored programs and triggers.

This chapter outlines how to use PL/SQL in procedures and functions, database triggers, and packages. You will learn how to pass values into a PL/SQL program and how to call one PL/SQL program from another so that you can modularize your code. This chapter also covers the basics of database triggers and database packages. These topics will enable you to take your PL/SQL programming skills and apply them in real-world situations

Subprograms

Using PL/SQL code in scripts allows you to take advantage of the PL/SQL language. When you include a block of PL/SQL code within a command file, or you execute a block of PL/SQL code from the SQL*Plus prompt, you are executing an anonymous block. An anonymous block is an unnamed PL/SQL program. The following is an example of an anonymous block.

```
BEGIN
   DBMS_OUTPUT.PUT_LINE('Hello World');
END;
/
```

In order to take full advantage of PL/SQL, in this chapter we introduce subprograms. Subprograms offer additional features that are not available in anonymous blocks:

- ✦ Subprograms can be stored in the database.
- ✦ One subprogram can call another subprogram.
- ✦ Subprograms can pass values to and from other subprograms.

There are different types of subprograms: procedures, functions, triggers, and packages. Not all types of subprograms have all the capabilities listed above. This chapter introduces each type of subprogram and describes where and how they are used.

Procedures

Procedures are named PL/SQL programs that are written to perform a particular task. Because the procedure is a named program, you can call one procedure from another. This enables you to modularize your code, making it easier to maintain.

You can pass values to a procedure, and you can receive values from a procedure. This enables you to write generic programs that can be reused.

Procedures that can be stored within the Oracle database are referred to as *server-side* or *stored* procedures. If you have a client tool with a PL/SQL engine, such as Procedure Builder or Oracle Forms, procedures can be stored on the client.

Compilation errors in server-side programs

When you execute a script to create a subprogram, you may have compilation errors in your code. In the following procedure, there should be single quotes surrounding the string "Hello World".

```
CREATE OR REPLACE PROCEDURE hello_world
IS
BEGIN
      DBMS_OUTPUT.PUT_LINE(Hello World);
END;
/
```

If you have compilation errors in your code, you will receive the following error message:

```
Warning: Procedure created with compilation errors.
```

In order to get a complete list of your error messages, type the following command:

```
SQL> show errors
```

This gives you a listing of all the compilation errors in your program.

```
Errors for PROCEDURE HELLO_WORLD:

LINE/COL ERROR
-------- -----------------------------------------------------
4/28     PLS-00103: Encountered the symbol "WORLD" when expecting
one of the following: . ( ) , * @ % & | = - + < / > at in mod not
range rem => ..<an exponent (**)> <> or != or ~= >= <= <> and or
like as between from using is null is not || is dangling
```

Use the show errors command to find the compilation errors in your program, then make the necessary modifications to your SQL script so your program compiles successfully.

Client-side procedures

Most PL/SQL programs are stored in the database. If you are working with a client tool that contains a PL/SQL engine, such as Oracle Forms, Oracle Reports, or Procedure Builder, you can store a PL/SQL program in a file on the operating system. When a PL/SQL program is stored in a file on the operating system, it is referred to as a *client-side* procedure.

The Procedure Builder tool enables you to create PL/SQL programs and store them either in the database or in program libraries on the file system. Procedure Builder provides a graphical environment in which you can write your PL/SQL code, as well as a code debugging tool.

Oracle Developer, Oracle Designer, and Oracle Financials are all tools that use or generate Oracle Forms and Oracle Reports. In Oracle Forms, PL/SQL code is used in form triggers to specify what actions are performed when a button is pressed or a field is entered. In Oracle Reports, PL/SQL code is used in report triggers to complete complicated calculations or to dynamically change report appearance. The PL/SQL code can be directly specified in the form or report trigger, or the trigger can call a PL/SQL procedure.

Server-side procedures

When a PL/SQL program is stored in the database, it is called a *server-side* or *stored* subprogram. In order to create a stored PL/SQL procedure, you write a SQL script to create the procedure. In the SQL script, you write the PL/SQL program code and add a specification instructing the database to create a procedure with a specified program name.

Let's create a program to display the text "Hello World" on the screen.

```
CREATE OR REPLACE PROCEDURE hello_world
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE('Hello World');
END;
/
```

Executing a script containing the preceding code creates the program hello_world in your database. When the procedure is created, you will see the following message:

```
Procedure created.
```

Once the procedure has been successfully created, you execute it in SQL*Plus by typing the following command:

```
SQL> SET SERVEROUTPUT ON
SQL> EXECUTE hello_world
Hello World

PL/SQL procedure successfully completed.
```

EXECUTE is an SQL*Plus command so you do not need a forward slash (/) or a semicolon (;) after the command.

When you create stored programs and you want to declare variables, you do not put the word "DECLARE" in the program code as was done with anonymous blocks. Any local variables you need to declare are listed between the word "IS" and the word "BEGIN".

```
CREATE OR REPLACE PROCEDURE hello_world
IS
   v_message VARCHAR2(50) := 'Hello World';
BEGIN
   DBMS_OUTPUT.PUT_LINE(v_message);
END;
/
```

Parameters

You can pass values to and receive values from a procedure. If you want to pass parameters, you must declare all the parameters to be passed in the program specification. The parameters are specified after the program name in the program specification.

When specifying a parameter, you specify:

- ✦ A parameter name
- ✦ The datatype of the parameter
- ✦ The type of parameter

The parameter name is the name you use when referencing the parameter within the program. The datatype of the parameter is any valid datatype. Unlike variables in the declaration section, you do not specify the size, only the datatype. The type of parameter is one of the following:

- ✦ IN parameters are passed to the procedure from the calling program.
- ✦ OUT parameters are passed from the procedure back to the calling program.
- ✦ IN OUT parameters have an initial value passed in by the calling program but may return a different value back to the calling program after execution.

IN parameters

IN parameters are used to pass parameters to a program when the program is called. IN parameters cannot be modified within the called program. IN is the default parameter type and does not have to be specified but should be specified for clarity.

To create a program that enables you to specify the message you would like to see displayed on the screen, enter the following code:

```
CREATE OR REPLACE PROCEDURE display_message
(p_message IN VARCHAR2)
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE(p_message);
END;
/
```

You must specify a value for the parameter when you call the program. If you are passing a character or date value to the program, you must enclose the value in single quotes.

```
SQL> EXECUTE display_message('HELLO AGAIN!')
HELLO AGAIN!

PL/SQL procedure successfully completed.
```

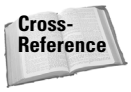
OUT parameters

OUT parameters are used to pass values from the called program back to the calling environment.

Create a program that accepts a price and returns the tax that should be charged for that price.

```
CREATE OR REPLACE PROCEDURE calculate_tax
(p_price IN NUMBER, p_tax OUT NUMBER)
IS
BEGIN
   p_tax := p_price *.15;
END;
/
```

When you call a program with an OUT parameter, you must specify a variable name to hold the value passed back by the program. When testing in SQL*Plus, use the SQL*Plus VARIABLE command to create the variable and then use it as a bind variable when executing the procedure. All the examples in this chapter are completed using the SQL*Plus tool. The syntax for creating bind variables will be different if you are calling the PL/SQL procedure from a different tool or from a programming language, such as C or COBOL.



The VARIABLE command is covered in detail in chapter 6, "The SQL*Plus Environment."

```
SQL> VARIABLE v_tax NUMBER
SQL> EXECUTE calculate_tax(20,:v_tax)

PL/SQL procedure successfully completed.

SQL> PRINT v_tax

     V_TAX
----------
         3
```

IN OUT parameters

IN OUT parameters are used to pass an initial value to the calling program but may be updated by the called program.

Create a program that accepts a price and checks whether that price is below the minimum price that may be charged. If the price is below the minimum, the program returns a corrected price.

```
CREATE OR REPLACE PROCEDURE check_minimum_price
(p_price IN OUT NUMBER)
IS
BEGIN
   IF p_price < 1000 THEN
        p_price := 1000;
   END IF;
END;
/
```

When you call a program with an IN OUT parameter, you must create a bind variable, assign the variable an initial value, and then pass that variable to the program.

```
SQL> VARIABLE v_price NUMBER
SQL> BEGIN :v_price := 500; END;
  2  /

PL/SQL procedure successfully completed.
SQL> EXECUTE check_minimum_price(:v_price)

PL/SQL procedure successfully completed.
SQL> PRINT v_price

   V_PRICE
----------
      1000
```

Nesting

Once you have created a stored procedure, it may be called from another program. This is called *nesting*. When you call the program, any program parameters must be passed to the program at the time it is called.

Go back to your program to display a message on the screen.

```
CREATE OR REPLACE PROCEDURE display_message
(p_message IN VARCHAR2)
IS
BEGIN
   DBMS_OUTPUT.PUT_LINE(p_message);
END;
/
```

Now call this program from another procedure as follows:

```
CREATE OR REPLACE PROCEDURE call_a_program
(p_language IN VARCHAR2)
IS
BEGIN
   IF p_language = 'ENGLISH' THEN
        display_message('Hello');
   ELSIF p_language = 'FRENCH' THEN
         display_message('Bonjour');
   ELSE
        display_message('Buenos Dias');
   END IF;
END;
/
```

To test the program, execute call_a_program from the command line.

```
SQL> EXECUTE call_a_program('FRENCH')
Bonjour

PL/SQL procedure successfully completed.
```

If the program being called contains an OUT or an IN OUT parameter, you must create a local variable to hold the value passed back by the called program. For IN OUT parameters, you should also populate the variable with an initial value.

Call your calculate_tax program from another PL/SQL procedure.

```
CREATE OR REPLACE PROCEDURE calculate_tax
(p_price IN NUMBER, p_tax OUT NUMBER)
IS
BEGIN
   p_tax := p_price *.15;
END;
/
```

```
CREATE OR REPLACE PROCEDURE call_tax
(p_course IN NUMBER)
IS
   v_tax NUMBER;
   v_price NUMBER;
BEGIN
   SELECT retailprice
     INTO v_price
     FROM courses
    WHERE coursenumber = p_course;

   calculate_tax(v_price, v_tax);

   DBMS_OUTPUT.PUT_LINE(
   'Tax on that course is '||TO_CHAR(v_price,'$9,990.00'));
END;
/
```

To test the call_tax program, execute it from the command line.

```
SQL> EXECUTE call_tax(100)
Tax on that course is  $2,000.00

PL/SQL procedure successfully completed.
```

Deleting Procedures

You can delete a stored procedure using the DROP PROCEDURE command.

```
SQL> DROP PROCEDURE call_tax;
Procedure dropped.
```

User-Defined Functions

PL/SQL functions are PL/SQL programs that always return a value. They are very useful for performing calculations or validation. When you create a function, you must specify the datatype of the value that will be returned in the program specification. Within the program code, you must include a RETURN statement that will return the value to the calling program and exit the function.

In chapter 3, "Using Single and Multi-Row Functions," you learned how to use functions such as TO_CHAR( ), ROUND( ) and NEXT_DAY( ) to manipulate the values in a SQL statement. You can write your own user-defined PL/SQL functions and use them to manipulate values in a SQL statement as well. There are some restrictions on the types of functions you can use in SQL statements. How to write your own functions for SQL statements and the restrictions on those functions are covered later in this section.

Create a function that will accept a first name, middle initial, and last name and will return the full name.

```
CREATE OR REPLACE FUNCTION full_name
(p_first IN VARCHAR2, p_middle IN VARCHAR2,
 p_last IN VARCHAR2)
RETURN VARCHAR2
IS
BEGIN
RETURN (p_first||' '||p_middle||' '||p_last);
END;
/
```

When you call a function from SQL*Plus, you must create a variable to hold the value returned by the function.

```
SQL> VARIABLE v_full_name VARCHAR2(100)
SQL> EXECUTE :v_full_name := full_name('John','P','Jones')

PL/SQL procedure successfully completed.

SQL> PRINT v_full_name

V_FULL_NAME
----------------------------------------------------------------
John P Jones
```

When you call a function from within a PL/SQL program, you must create a local variable to hold the value returned by the function.

The following example shows how to call a function from within a PL/SQL program. This procedure accepts a student number. The procedure fetches the first name, middle initial, and last name of the specified student and calls the function full_name to concatenate the name. The procedure prints the concatenated name on the screen.

```
CREATE OR REPLACE PROCEDURE student_name
(p_studentnumber IN NUMBER)
IS
   v_firstname     students.firstname%TYPE;
   v_middleinitial students.middleinitial%TYPE;
   v_lastname      students.lastname%TYPE;

   v_full_name     VARCHAR2(100);
BEGIN

   SELECT firstname, middleinitial, lastname
     INTO v_firstname, v_middleinitial, v_lastname
     FROM students
    WHERE studentnumber = p_studentnumber;
```

```
-- Our function is used to assign a value to the variable
   v_full_name :=  full_name(v_firstname,
                   v_middleinitial, v_lastname);

   DBMS_OUTPUT.PUT_LINE(v_full_name);

END;
/

SQL> EXECUTE student_name(1000)
John H     Smith

PL/SQL procedure successfully completed.
```

PL/SQL functions may also be called from SQL statements, enabling you to write your very own SQL functions. Some restrictions are placed on PL/SQL functions used within SQL statements. If you want to use your PL/SQL function in a SQL statement, the function cannot perform INSERT, UPDATE, or DELETE statements, and the function must return a datatype that is a valid datatype for a database table (BOOLEAN, for example, could not be used). If you follow these rules, you can write your own single-row SQL functions. You cannot create your own SQL group or aggregate functions — that is, functions that perform a calculation on several rows of the table to return a single value such as Oracle AVG( ), MIN( ), and MAX( ) functions.

The full_name function does not perform any UPDATE, INSERT, or DELETE statements and returns a datatype of VARCHAR2, so it may be called from a SQL statement. Here is an example of how to call the function from a SELECT statement using the table's columns as the actual parameters for the function:

```
SQL> SELECT full_name(firstname, middleinitial, lastname)
  2  FROM students
  3  /

FULL_NAME(FIRSTNAME,MIDDLEINITIAL,LASTNAME)
------------------------------------------------------------
John H     Smith
Davey  Jones
Jane S     Massey
Trevor J     Smith
...
```

Like all single-row SQL functions, PL/SQL functions can be called from the WHERE clause, ORDER BY clause, HAVING clause, or SELECT list of a SQL statement.

Deleting Functions

You can delete a stored procedure using the DROP PROCEDURE command.

```
SQL> DROP FUNCTION full_name;
```

Packages

Any PL/SQL programs can be put inside a package. A package is a grouping of PL/SQL programs stored together. By placing your PL/SQL programs inside a package, you can improve performance and increase functionality.

Performance can be improved by putting PL/SQL code in packages when you frequently call a number of programs one after the other. For example, you might want to write a program to enroll a student in a class:

1. Your program calls a check_status program that checks whether the course has been canceled.
2. The program calls a check_max program to ensure the class is not full.
3. The program calls a create_enrollment program that creates the enrollment record.

If you do not use a package, each of these programs is loaded into memory when it is called. If you put all these programs into a package, all the programs will be loaded into memory at once when the first program in the package is called.

When packages are loaded into memory, they are loaded into the shared pool. If you are writing a lot of packages, you may need to increase the size of the shared pool.

Increased functionality can be added when packages are used because you can create variables in a package, and they will hold their value throughout the database session. Without packages, you can create local variables in PL/SQL programs, but the values contained in these variables are lost when the program execution stops. When you create a package, values in any variables declared in the program specification are kept in memory and may be accessed at any time throughout the database session.

When you create a package, you must first create a package specification that lists all the public programs and variables, and then you create a package body that specifies all the PL/SQL code and any private variables you wish to create.

Specification

The package specification is a list of all the PL/SQL procedures and functions you are going to include in the package that will be public. *Public* programs are programs that can be accessed by anyone with permissions to execute the package. The package specification also lists any public variables you wish to create. A public variable is a variable that can be read or modified by anyone with permissions to execute the package.

To create the package specification, you must specify the following:

- ✦ A package name
- ✦ A list of programs and their parameters
- ✦ A list of public variables

```
CREATE OR REPLACE PACKAGE package_name
IS
   variable_declarations;
   program_declarations;
END;
/
```

For example, you might want to create an enrollment package that contains the public programs CHECK_CANCELED and ENROLL_STUDENT, and a variable called class_maximum.

```
CREATE OR REPLACE PACKAGE enrollment
IS
   class_maximum NUMBER := 16;

   FUNCTION check_canceled (p_classid IN NUMBER)
   RETURN BOOLEAN;

   PROCEDURE enroll_student
   (p_studentnumber IN NUMBER,
    p_classid IN NUMBER,
    p_price IN NUMBER);
END;
/
```

Package body

Once you have created the package specification, you can create the package body. The package body contains the code for all of the programs listed in the package specification.

Additional programs can be included in the package body that are not listed in the package specification. These are called *private* programs because they can be called only by other programs within the package. Additional variables can be included in the package body that are not listed in the package itself. These are called *private* variables because they cannot be accessed from outside the package.

To create the package body, you use the CREATE PACKAGE BODY command and specify the same package name you used for the package specification.

```
CREATE OR REPLACE PACKAGE BODY package_name
IS
   variable_declarations;
   program code;
END;
/
```

For example, for your enrollment package, you need three programs: the two public programs, check_canceled, enroll_student, and check_maximum as well as, Check_maximum, which is a private function because it was not declared in the package specification.

```
CREATE OR REPLACE PACKAGE BODY enrollment
IS
   --Code for the check_canceled function
   FUNCTION check_canceled (p_classid IN NUMBER)
   RETURN BOOLEAN
   IS
        --declare a variable to hold the status of the
        --enrollment
        v_status ScheduledClasses.status%TYPE;
   BEGIN
        --Fetch the status of the specified class
        SELECT status
          INTO v_status
          FROM scheduledclasses
         WHERE classid = p_classid;

        --If the class is canceled, return TRUE,
        --otherwise, return FALSE
        IF v_status = 'Cancelled' THEN
             RETURN(TRUE);
        ELSE
             RETURN(FALSE);
        END IF;
   END check_canceled;

   --Code for the check_maximum function
   FUNCTION check_maximum (p_classid IN NUMBER)
   RETURN BOOLEAN
   IS
        --declare a variable v_total to hold
        --the number of students in the class
        v_total NUMBER;
   BEGIN
        --Fetch the number of students currently
        --enrolled in the specified class
        SELECT COUNT(*)
          INTO v_total
          FROM classenrollment
         WHERE classid = p_classid;
```

```
        --Compare the number of students currently enrolled
        --to the package variable 'Class_maximum'
        IF v_total >= class_maximum THEN
             RETURN(TRUE);
        ELSE
             RETURN(FALSE);
        END IF;
   END check_maximum;

   --Code for the enroll_student procedure
   PROCEDURE enroll_student
     (p_studentnumber IN NUMBER,
      p_classid IN NUMBER,
      p_price IN NUMBER)
   IS
   BEGIN

     --Before enrolling student, check if class is canceled.
     IF check_canceled(p_classid) THEN
        RAISE_APPLICATION_ERROR
        (-20002,'Course is canceled');
     END IF;

     --Before enrolling student, check if class is full
     IF check_maximum(p_classid) THEN
        RAISE_APPLICATION_ERROR
        (-20003,'Course is full');
     END IF;

   --Enroll student by creating new record in
   --Classenrollment
   INSERT INTO ClassEnrollment
   (ClassId, StudentNumber, Status, EnrollmentDate, Price,
Grade, Comments)
   VALUES (p_classid, p_studentnumber, 'Hold',
   SYSDATE, p_price, NULL, NULL);

    --Save the enrollment record
   COMMIT;
   END enroll_student;

END enrollment;
/
```

Accessing programs and variables in packages

When you access a program or a variable in a package, you must qualify the program or variable name with the name of the package where the program or variable is located.

```
CREATE OR REPLACE PACKAGE enrollment
IS
   class_maximum NUMBER := 16;

   FUNCTION check_canceled (p_classid IN NUMBER)
   RETURN BOOLEAN;

   PROCEDURE enroll_student
   (p_studentnumber IN NUMBER,
    p_classid IN NUMBER,
    p_price IN NUMBER);
END;
/
```

When you call a program in a package, you must begin the program name with the package name as a prefix. For example, if I want to call the program enroll_student in the package enrollment, I would execute the following:

```
SQL> EXECUTE enrollment.enroll_student(1000,51, 2000)
```

The only time you do not need to use the package prefix is when you are calling a program in a package from another program within the same package.

When you want to access a variable that is declared in a package, you must begin the variable name with the package name as a prefix. For example, if I want to change the value of the package variable class_maximum in the package enrollment, I would execute the following:

```
SQL> BEGIN enrollment.class_maximum := 20; END;
   2 /
```

The only time you do not need to use the package prefix is when you are referencing a package variable from a program in the same package.

Removing Packages

After a package has been created, it can be removed with the DROP PACKAGE command.

The following example will drop the package body and the package specification for the package my_package:

```
SQL> DROP PACKAGE my_package;
```

The following example drops just the package body for the package my_package:

```
SQL> DROP_PACKAGE BODY my_package;
```

Listing Package contents

If you are using the SQL*Plus tool, you can get a list of all the public programs and variables in a package using the DESCRIBE command.

```
SQL> DESCRIBE my_package
```

Triggers

PL/SQL code can be used to write database triggers. Database triggers are PL/SQL programs that fire when certain actions occur within the database. They are very useful for ensuring database integrity.

There are several types of database triggers: statement triggers, row-level triggers, event triggers, and instead-of triggers.

Statement and row-level triggers are PL/SQL programs associated with a particular database table. They fire when an INSERT, UPDATE, or DELETE statement is issued on that table. Statement level triggers fire once when an INSERT, UPDATE, or DELETE statement is run against the table. Statement level triggers are used to prevent users from updating records in the table after hours. Row-level triggers fire once for each row affected by an INSERT, UPDATE, or DELETE statement. They are used to audit changes made to the database table, to derive column values, and to prevent certain changes to records in the table.

Instead-of triggers are PL/SQL programs associated with a particular database view. They fire when an INSERT, UPDATE, or DELETE is run against the view. Instead-of triggers are used to allow users to insert, delete, or update records in a table that is accessed through a view.

Event triggers are PL/SQL programs associated with certain commands or database actions. They fire when that command is executed or that action is performed in the database. Event triggers are used to audit activities in the database and to perform certain actions when the database starts up, shuts down, or receives an error.

All triggers are created using the CREATE TRIGGER command:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE|AFTER trigger_event ON trigger_object
```

- ✦ The trigger_name is the name of the PL/SQL trigger that is created.
- ✦ The trigger_event is the type of command or action that will cause the PL/SQL trigger to fire.
- ✦ The trigger_object is the object with which the trigger is associated.

Once a trigger is created, it will fire automatically when the triggering event occurs.

Triggering events

Different events in the database can cause database triggers to fire. The event that fires the trigger depends on the type of trigger and the triggering events listed in the trigger specification. Database triggers fire regardless of how the command is issued; this is why they are useful for ensuring database integrity. For example, if you have a Web page and an Oracle Developer application that sends updates to the database and you have a support team that uses SQL*Plus to update the database, the database triggers fire regardless of who or what application issues the command.

The triggering event or events are included in the trigger specification.

A table trigger can be triggered by a DELETE, INSERT or UPDATE:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE INSERT OR UPDATE OR DELETE ON table_name
```

If you specify UPDATE as the triggering event, you can specify that the trigger should fire only when particular columns are updated. This is done by specifiying the column name or names after the keyword UPDATE. If you are specifying more than one column, separate the column names with commas:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE INSERT OR DELETE OR UPDATE OF column1, column2 ON
table_name
```

An instead-of trigger can be triggered by a DELETE, INSERT, or UPDATE:

```
CREATE OR REPLACE TRIGGER add_course
INSTEAD OF DELETE ON view_name
```

An event trigger can be triggerd by database startup, database shutdown, server errors, DDL and DCL commands, logon or logoff:

```
CREATE OR REPLACE TRIGGER trigger_name
AFTER SERVERERROR ON DATABASE

CREATE OR REPLACE TRIGGER trigger_name
AFTER LOGON ON DATABASE
```

Statement-level triggers

Statement-level triggers fire once for each triggering event. In the trigger code, you can perform validation and calculations, execute SQL statements, and call other PL/SQL programs. When you raise an error in a trigger, the triggering UPDATE, INSERT, or DELETE is rolled back. Statement-level triggers are used to prevent users from performing modifications to a table under specified conditions.

When you create a statement-level trigger, you must specify the following in the trigger heading:

- ✦ Trigger name
- ✦ Trigger table
- ✦ Triggering event
- ✦ BEFORE or AFTER event

The trigger name is a program name you give to the trigger. The trigger table is the name of the database table that is being updated when you want the trigger to fire. The triggering event is INSERT, UPDATE, DELETE, or a combination of the three. The triggering event specifies what commands on the trigger table will cause the trigger to fire. A BEFORE or an AFTER event specifies whether the database trigger will fire before or after the update that caused the trigger to fire.

If you are creating a trigger and it does not matter whether the trigger fires before or after the triggering event, fire the trigger after the event for best performance results.

All these parameters are specified in the trigger specification as follows:

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE|AFTER trigger_event ON trigger_table
```

For example, suppose you want to prevent users from updating the classenrollment table on weekends. You want to create a database trigger that will stop users from udpating, inserting, or deleting records from the classenrollment table when it is Saturday or Sunday.

- ✦ The trigger name is check_weekend.
- ✦ The trigger table is classenrollment, because you want the trigger to fire when changes are made to the classenrollment table.
- ✦ The trigger events are INSERT, UPDATE, and DELETE, because you do not want the user to make any changes to the contents of the classenrollment table on weekends.
- ✦ The trigger should fire before the triggering event, because you do not want to waste time processing the INSERT, UPDATE, or DELETE if CLASS_STATUS is running.

This code checks if the current date falls on a Saturday or Sunday. If it is Saturday or Sunday, it will raise an error that will rollback the triggering event and prevent the user from making changes to the classenrollment table.

```
CREATE OR REPLACE TRIGGER check_weekend
BEFORE INSERT OR UPDATE OR DELETE ON classenrollment
BEGIN
```

```
    IF TO_CHAR(sysdate,'DY') IN ('SAT','SUN') THEN
         RAISE_APPLICATION_ERROR (-20001,'Error: updates are not
         allowed on weekends.');
    END IF;

END;
/
```

If you attempt to update the classenrollment table on a Saturday or Sunday, you will get an error, and the update will be rolled back.

```
SQL> UPDATE classenrollment
  2  SET status = 'Confirmed'
  3  WHERE classid = 53
  4  AND studentnumber = 1003
  5  /
UPDATE classenrollment
       *
ERROR at line 1:
ORA-20001: Error: updates are not allowed on weekends.
ORA-06512: at "STUDENT3.CHECK_WEEKEND", line 4
ORA-04088: error during execution of trigger
'STUDENT3.CHECK_WEEKEND'
```

Row-level triggers

Row-level triggers fire once for each record updated, deleted, or inserted. They are often used to perform complex validation or calculations, to populate default values, to modify data in related tables, or to modify data values.

Row-level triggers are created by adding the statement FOR EACH ROW to the trigger specification.

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE|AFTER trigger_event ON trigger_table
FOR EACH ROW
```

Correlation Names

Because row-level triggers fire for each actual record being updated, you can access the values contained in those records. To access the values in the record, you specify the name of the column whose value you wish to access with the prefixes :OLD or :NEW:

- ✦ If you are updating a record, :OLD returns the values of the original record in the database, and :NEW returns the new values being sent to the database for update.
- ✦ If you are deleting a record, :OLD returns the values contained in the record about to be deleted, and :NEW returns NULL.

- ✦ If you are inserting a record, :OLD returns NULL, and :NEW returns the values about to be inserted into the database.

You can change the prefixes used to reference :OLD and :NEW by adding the REFERENCING clause to your trigger specification before the FOR EACH ROW clause.

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE|AFTER trigger_event ON trigger_table
REFERENCING OLD AS original NEW AS changed
FOR EACH ROW
```

The following example is a trigger that will ensure that the state column in the students table is always entered in uppercase letters:

- ✦ The trigger name is upper_state.
- ✦ The trigger table is students, because you want the trigger to fire when changes are made to the students table.
- ✦ The trigger column to check on UPDATE is state.
- ✦ The triggering events are INSERT and UPDATE of state, because you want the trigger to fire when new records are inserted or updates to the state column are made.
- ✦ The trigger will fire BEFORE the triggering event, because you will modify the actual value being inserted or updated.
- ✦ The default prefixes will be used.

```
CREATE OR REPLACE TRIGGER upper_state
BEFORE INSERT OR UPDATE OF state ON students
FOR EACH ROW
BEGIN
   :NEW.state := UPPER(:NEW.state);
END;
/

SQL> UPDATE students
  2  SET city = 'Boston', state = 'ma'
  3  WHERE studentnumber = 1005
  4  /

1 row updated.

SQL> SELECT studentnumber, city, state
  2  FROM students
  3  WHERE studentnumber = 1005
  4  /

STUDENTNUMBER CITY                           ST
------------- ------------------------------ --
         1005 Boston                         MA
```

WHEN clause

The WHEN clause can be used on row-level triggers to prevent unnecessary firing of a database trigger. A condition is specified in the WHEN clause. If that condition is not met, the database trigger code will not be executed. The WHEN clause is added to the trigger specification after the FOR EACH ROW clause.

```
CREATE OR REPLACE TRIGGER trigger_name
BEFORE|AFTER trigger_event ON trigger_table
FOR EACH ROW
WHEN (condition)
```

For example, U.S. zip codes are numeric, and Canadian postal codes are alphanumeric. If a Canadian record is entered, you want to ensure the postal code is all uppercase. This is unnecessary in the United States, because only numbers are specified. To avoid wasting processing time when entering U.S. records in the students table, you add a WHEN clause so the trigger only peforms the check of postalcode when the student is Canadian:

- ✦ The trigger name is canada_postal_code.
- ✦ The trigger table is students, because you want the trigger to fire when changes are made to the students table.
- ✦ The triggering events are INSERT and UPDATE of postalcode, because you want the trigger to fire when new records are inserted or updates to the postalcode column are made.
- ✦ The trigger will fire BEFORE the triggering event, because you will modify the actual value being inserted or updated.
- ✦ The default prefixes will be used.
- ✦ The trigger should check the value for postalcode only when the record has a country of "Canada", because you do not need to convert U.S. zip codes to uppercase.

```
CREATE OR REPLACE TRIGGER canada_postal_code
BEFORE INSERT OR UPDATE OF postalcode ON students
FOR EACH ROW
WHEN (new.country = 'Canada')
BEGIN
   :NEW.postalcode := UPPER(:NEW.postalcode);
END;
/
```

When referencing the :NEW and :OLD prefixes in the WHEN clause, you do not use the colon (:) in front of the prefix.

```
SQL> UPDATE students
  2  SET postalcode = 'm5h 5f6'
  3  WHERE studentnumber = 1003
  4  /
```

```
1 row updated.

SQL> SELECT studentnumber, postalcode
  2  FROM students
  3  WHERE studentnumber = 1003
  4  /

STUDENTNUMBER POSTALCODE
------------- ----------
         1003 M5H 5F6
```

Trigger predicates

When you have a trigger that fires for multiple triggering events, such as INSERT and UPDATE, you may wish to perform different actions depending on which event actually caused the trigger to fire. Trigger predicates enable you to check which action was performed from within the trigger. There are four trigger predicates:

- ✦ INSERTING: Returns TRUE if INSERT statement fired trigger.
- ✦ DELETING: Returns TRUE if DELETE statement fired trigger.
- ✦ UPDATING: Returns TRUE if UPDATE statement fired trigger.
- ✦ UPDATING(column_name): Returns TRUE if UPDATE statement, which updates the specified column, fired trigger.

These predicates are referenced within the database trigger code. They return TRUE or FALSE depending on the action that caused the trigger to fire.

For example, you might want to use a database trigger to keep an audit of changes to the courses table. You need to track when new courses are added and deleted and when course prices are changed.

```
CREATE OR REPLACE TRIGGER audit_courses
AFTER INSERT OR UPDATE OF RetailPrice OR DELETE ON courses
FOR EACH ROW
BEGIN
   IF INSERTING THEN
        INSERT INTO courseaudit
        (coursenumber,change,price,changedby,datechanged)
        VALUES
        (:NEW.coursenumber, 'INSERT', :NEW.retailprice, USER,
SYSDATE);

   ELSIF DELETING THEN
        INSERT INTO courseaudit
        (coursenumber,change,price,changedby,datechanged)
        VALUES
        (:OLD.coursenumber,'DELETE',:OLD.retailprice,
        USER, SYSDATE);
```

```
   ELSIF UPDATING THEN
        INSERT INTO courseaudit
        (coursenumber,change,price,changedby,datechanged)
        VALUES
        (:OLD.coursenumber, 'UPDATE PRICE', :OLD.retailprice,
USER, SYSDATE);

   END IF;
END;
/

SQL>  INSERT INTO COURSES (coursenumber, coursename,
replacescourse, retailprice, description)
  2   VALUES (400,'Database Concepts',null,1000,
  3  'A course which gives an overview of database concepts and
capabilities');

1 row created.

SQL> UPDATE courses
  2  SET retailprice = 1500
  3  WHERE coursenumber = 400
  4  /

1 row updated.

SQL> DELETE FROM courses
  2  WHERE coursenumber = 400
  3  /

1 row deleted.

SQL> SELECT *
  2  FROM courseaudit;

COURSENUMBER CHANGE         DATECHANGED    PRICE CHANGEDBY
------------ -------------- -------------- ----- ---------
         400 INSERT         10-JAN-01       1000 STUDENT3
         400 UPDATE PRICE   10-JAN-01       1000 STUDENT3
         400 DELETE         10-JAN-01       1500 STUDENT3
```

Restrictions on triggers

Triggers can be very useful for maintaining database integrity, but there are restrictions on what you can do within database triggers.

COMMIT and ROLLBACK

COMMIT and ROLLBACK statements are not permitted inside database triggers. If you have an INSERT, UPDATE, or DELETE statement inside the database trigger, it is committed when the statement that triggered the database trigger is committed, or rolled back when the statement that triggered the database trigger is rolled back.

Mutating tables

Because database triggers can fire when data contained in a table is changing, the table a database trigger is assigned to is said to be *mutating* when the database trigger fires. You are not allowed to write SQL statements that access mutating tables inside database triggers. In addition, tables with foreign keys pointing to the trigger table are also said to be mutating and cannot be updated by the trigger.

The following example demonstrates the mutating table concept. When a user updates the perdiemcost for an instructor, you might want to limit the perdiemcost so that it cannot exceed the highest perdiemcost already in the database by more than 20 percent. In order to do this, you create a trigger on the instructors table that fires when the perdiemcost is updated. The code selects the highest perdiemcost in the instructors table, adds 20 percent, and compares it to the perdiemcost specified in the update. Unfortunately, the instructors table is the trigger table and is therefore mutating. You cannot execute a SQL statement against a mutating table, so you cannot use a database trigger to solve this problem.

Here is an example where the foreign key restricts a trigger. When you update a coursenumber in the courses table, you might want to automatically update all the ScheduledClasses records to reflect the new coursenumber. The trigger on the courses table would fire when coursenumber is updated. The trigger would execute an update on ScheduledClasses with the new coursenumber. Unfortunately, ScheduledClasses has a foreign key on coursenumber that points to the courses table. This means ScheduledClasses is a mutating table, so no update statements can be executed against the ScheduledClasses table within a trigger on the courses table. You cannot use a database trigger to solve this problem unless you remove the foreign key between Courses and ScheduledClasses.

Order of firing

When you create database triggers, it is important to understand when database triggers fire with respect to each other and with respect to the triggering event. The following shows the order in which triggers and events are executed:

1. User executes a DML statement (INSERT, UPDATE, or DELETE).
2. Oracle fires any BEFORE statement-level triggers.
3. Oracle fires any BEFORE row-level triggers.
4. Oracle executes DML on that row.
5. Oracle fires any AFTER row-level triggers.
6. If more than one row is modified, steps 3 through 5 are repeated until all rows have been processed.
7. Oracle fires any AFTER statement-level triggers.

If multiple database triggers on the same table are triggered by the same triggering event, the order in which those triggers will be fired cannot be determined. When you require the commands in one trigger to be fired before another, you should combine the code into one database trigger and list the code in the order you want it executed.

One database table can have many database triggers, and one database trigger can perform many actions.

A useful guideline for determining whether to create separate triggers or to combine them into one trigger is to ask yourself how many different business rules you are enforcing. Create one database trigger for each business rule.

INSTEAD OF triggers

If you are using complex views (that is, views that are based on more than one table) in your database, you may not be able to use UPDATE, DELETE, and INSERT statements on the view to update the tables accessed by the view. The INSTEAD OF trigger enables you to update the tables accessed by any view, including complex views. INSTEAD OF triggers can be placed only on views and always fire for each row.

You might have a view that shows all the ScheduledClasses and course information.

```
SQL> CREATE OR REPLACE VIEW CourseSchedule
  2  AS SELECT sc.classid, sc.coursenumber, sc.locationid,
  3  c.coursename, c.retailprice
  4  FROM ScheduledClasses sc, Courses c
  5  WHERE sc.coursenumber = c.coursenumber;
```

If you try to add a new course through the view, you will get an error because mandatory columns in the scheduledclass table were not included in the view.

```
SQL> INSERT INTO CourseSchedule (classid, coursenumber,
  2  locationid, coursename, retailprice)
  3  VALUES (50, 500, 200,
  4  'SQL for beginners', 3000);

ORA-01400: mandatory (NOT NULL) column is missing or NULL
during insert
```

Oracle will not insert into the Courses table because the primary key of the Courses table is not selected. Using an INSTEAD OF trigger, you can tell Oracle how to add a record to the courses table when a user INSERTS into the view.

```
SQL> CREATE OR REPLACE TRIGGER add_course
  2  INSTEAD OF INSERT ON CourseSchedule
  3  FOR EACH ROW
```

```
  4  INSERT INTO Courses (CourseNumber, CourseName,
  5  ReplacesCourse, RetailPrice, Description)
  6  VALUES (:NEW.CourseNumber, :NEW.CourseName, null,
  7  :NEW.retailprice, null);
  8  END;
  9  /
```

Now when you insert a row into the CourseSchedule view, a record is created in the Courses table.

```
 SQL> INSERT INTO CourseSchedule (classid, coursenumber,
  2  locationid, coursename, retailprice)
  3  VALUES (50, 500, 200,
  4  'SQL for beginners', 3000)
  5  /

SQL> SELECT *
  2  FROM courses
  3  WHERE coursenumber = 500
  4  /
COURSENUMBER COURSENAME                    REPLACESCOURSE
RETAILPRICE
------------ ------------------------------ --------------
DESCRIPTION
----------------------------------------------------------------
         500 SQL for beginners                              3000
```

Event triggers

In addition to creating triggers that fire when an INSERT, UPDATE, or DELETE is executed on a particular table or view, you can create triggers that fire when certain events occur in the database. These are called *event* triggers. There are two types of event triggers.

The first type of event triggers are *resource manager*, or *system event*, triggers. These triggers can fire after database startup, before database shutdown, and after server errors. For example, you can create a trigger that writes error information to a table whenever a server error occurs.

```
SQL> CREATE OR REPLACE TRIGGER track_errors
  2  AFTER SERVERERROR
  3  ON DATABASE
  4  INSERT INTO error_log (username, errortime, error)
  5  VALUES (ora_login_user, SYSDATE,
  6  'Error - '||ora_server_error(1))
  7  END
  7  /
```

Each event trigger has a number of attribute functions that can provide information about the triggering event. In the preceding example, ora_login_user returns the

username of the person who caused the triggering event. ora_server_error accepts a position number and returns the error number with that position in the stack. Position one is the top of the stack.

The second type of event triggers are *client* event triggers, which can fire after LOGON, before LOGOFF, or before and after most DDL and DCL commands.

For example, if you want to track when a user accesses your database, you could fire a trigger whenever a user logs on to add a record to an audit table.

```
SQL> CREATE OR REPLACE TRIGGER Track_logon
  2  AFTER LOGON
  3  ON DATABASE
  4  BEGIN
  5  INSERT INTO audit_logon (username, logontime)
  6  VALUES (ora_login_user, SYSDATE);
  7  END;
  8  /
```

Auditing can also be performed using Oracle's auditing functions. Using Oracle's built-in auditing function has the advantage that it does not require you to build your own set of auditing tables and triggers to populate them. Using event triggers to perform auditing allows you to control exactly what information you keep.

Event triggers are available only in Oracle 8*i* and higher.

You can also create triggers that log the creation, modification, or deletion of tables or views, or other database objects in your schema. For example, to log an event whenever an object in your schema is altered, you could create the following trigger:

```
SQL> CREATE OR REPLACE TRIGGER Track_alter
  2  AFTER ALTER
  3  ON SCHEMA
  4  BEGIN
  5  INSERT INTO audit_logon (username, logontime,      action)
  6  VALUES (ora_login_user, SYSDATE, 'ALTER');
  7  END;
  8  /
```

Enabling and disabling triggers

Once a database trigger has been created, it can be disabled and reenabled using the ALTER TRIGGER command. When a trigger is created, it is automatically enabled. A disabled trigger does not fire when the triggering event takes place. A disabled trigger remains disabled until it is reenabled using the ALTER TRIGGER command.

```
SQL> ALTER TRIGGER my_trig disable;
SQL> ALTER TRIGGER my_trig enable;
```

You can also use the ALTER TABLE command to enable or disable all triggers on a particular table.

```
SQL> ALTER TABLE students ENABLE ALL TRIGGERS;
SQL> ALTER TABLE students DISABLE ALL TRIGGERS;
```

Removing triggers

After a database trigger has been created, you can remove the trigger using the DROP TRIGGER command.

```
SQL> DROP TRIGGER my_trig;
```

Only use the DROP TRIGGER command if you are permanently removing the trigger or planning to rewrite the trigger. If you want to disable the trigger temporarily, use the ALTER TRIGGER command.

Data Dictionary Views

A number of data dictionary views can provide information about your stored PL/SQL programs and triggers.

USER_SOURCE

The data dictionary view USER_SOURCE contains the source code for all stored procedures, functions, and packages that are part of your schema — that is, those that you have created.

```
SQL> desc user_source
 Name                            Null?    Type
 ------------------------------- -------- ----
 NAME                                     VARCHAR2(30)
 TYPE                                     VARCHAR2(12)
 LINE                                     NUMBER
 TEXT                                     VARCHAR2(4000)
```

- ✦ NAME is the name of the procedure, function, or package.
- ✦ TYPE is the type of program: FUNCTION, PROCEDURE, PACKAGE, or PACKAGE BODY.
- ✦ LINE is the line number of the source code.
- ✦ TEXT contains the source code.

You can use the USER_SOURCE view to get a list of all procedures, functions, and packages. You can also use the USER_SOURCE view to see the source code for a procedure, function, or package.

```
SQL> SELECT text
  2  FROM user_source
  3  WHERE name = 'HELLO_WORLD'
  4  ORDER BY line
  5  /

TEXT
------------------------------------------------
PROCEDURE hello_world
IS BEGIN
DBMS_OUTPUT.PUT_LINE('Hello World');
END;
```

USER_TRIGGERS

The data dictionary view USER_TRIGGERS contains information about database triggers in your schema. Both row-level and statement-level trigger information is stored in the USER_TRIGGERS view.

```
SQL> desc user_triggers
 Name                            Null?    Type
 ------------------------------- -------- ----
 TRIGGER_NAME                             VARCHAR2(30)
 TRIGGER_TYPE                             VARCHAR2(16)
 TRIGGERING_EVENT                         VARCHAR2(216)
 TABLE_OWNER                              VARCHAR2(30)
 BASE_OBJECT_TYPE                         VARCHAR2(16)
 TABLE_NAME                               VARCHAR2(30)
 COLUMN_NAME                              VARCHAR2(4000)
 REFERENCING_NAMES                        VARCHAR2(128)
 WHEN_CLAUSE                              VARCHAR2(4000)
 STATUS                                   VARCHAR2(8)
 DESCRIPTION                              VARCHAR2(4000)
 ACTION_TYPE                              VARCHAR2(11)
 TRIGGER_BODY                             LONG
 GENERATED                                VARCHAR2(1)
 SECONDARY                                VARCHAR2(1)
```

The most useful columns in the view and their contents are the following:

- ✦ TRIGGER_NAME contains the trigger name.
- ✦ TRIGGER_TYPE indicates whether the trigger fires BEFORE or AFTER and whether the trigger is statement or row-level.
- ✦ TRIGGERING_EVENT lists the events that cause the trigger to fire.

- ✦ TABLE_OWNER contains the username of the owner of the table with the trigger.
- ✦ TABLE_NAME contains the name of the table with the trigger.
- ✦ REFERENCING_NAMES contains the prefixes for referencing old and new values.
- ✦ WHEN_CLAUSE contains the WHEN clause specified in the trigger.
- ✦ STATUS indicates whether the trigger is enabled or disabled.
- ✦ DESCRIPTION contains the full trigger specification.
- ✦ TRIGGER_BODY contains the PL/SQL code executed when the trigger fires.

The USER_TRIGGERS view can be used to get a list of all the database triggers on a particular table. It can be used to see what triggers have been disabled. It can also be used to see the specification and source code for a trigger.

```
SQL> SELECT description, trigger_body
  2  FROM user_triggers
  3  WHERE trigger_name = 'MY_TRIG'
  4  /

DESCRIPTION
------------------------------------------------------------
my_trig
BEFORE UPDATE OF lastname ON instructors
FOR EACH ROW
TRIGGER_BODY
------------------------------------------------------------
BEGIN
        :NEW.lastname := UPPER(:NEW.lastname);
END;

SQL> ALTER TRIGGER my_trig disable;

Trigger altered.

SQL> select trigger_name, status
  2  FROM user_triggers
  3  WHERE trigger_name = 'MY_TRIG'
  4  /

TRIGGER_NAME                   STATUS
------------------------------ --------
MY_TRIG                        DISABLED
```

USER_OBJECTS

The data dictionary view USER_OBJECTS contains information about all the database objects contained in the database that are in your schema, including PL/SQL programs.

```
SQL> desc user_objects
 Name                           Null?    Type
 ------------------------------ -------- ----
 OBJECT_NAME                             VARCHAR2(128)
 SUBOBJECT_NAME                          VARCHAR2(30)
 OBJECT_ID                               NUMBER
 DATA_OBJECT_ID                          NUMBER
 OBJECT_TYPE                             VARCHAR2(18)
 CREATED                                 DATE
 LAST_DDL_TIME                           DATE
 TIMESTAMP                               VARCHAR2(19)
 STATUS                                  VARCHAR2(7)
 TEMPORARY                               VARCHAR2(1)
```

The most useful columns in the view and their contents are the following:

- ✦ OBJECT_NAME contains the name of the database object.
- ✦ OBJECT_ID contains a unique identifier assigned to each database object.
- ✦ OBJECT_TYPE indicates the type of database object.
- ✦ CREATED indicates when the database object was created.
- ✦ LAST_DDL_TIME indicates the last time the database object was modified.
- ✦ STATUS indicates whether the database object is VALID or INVALID.

Of particular interest in the USER_OBJECTS view is the STATUS column. This column indicates if a program is VALID or INVALID. A program with a status of INVALID must be recompiled. This can occur when you make a structural change to a table that is called by a stored program, or when you change the program specification of a program that is called by a stored program.

For example, when you add a new column to the ScheduledClasses table, the ENROLL_STUDENT program that inserts records to that table will be marked as INVALID. If you do not recompile the program yourself, Oracle attempts to recompile the program the next time the program is called. Depending on the change made to the database table, the program may or may not recompile successfully. You should check the USER_OBJECTS view whenever you change the structure of a database table or the specification of a program and recompile any programs that are marked as INVALID.

```
SQL> SELECT object_name, object_type, status
  2  FROM user_objects
  3  WHERE status = 'INVALID'
  4  /
```

Key Point Summary

This chapter showed how to use PL/SQL in procedures and functions, database triggers, and packages. You saw how to pass values into a PL/SQL program and how to call one PL/SQL program from another. This chapter also introduced the basics of database triggers and database packages.

- ✦ PL/SQL programs can be stored in the database.
- ✦ You can name your PL/SQL programs and call one PL/SQL program from another.
- ✦ Parameters can be passed to and from PL/SQL programs.
- ✦ PL/SQL procedures are programs that can accept and return parameters. PL/SQL functions are programs that can accept parameters and always return a value.
- ✦ You can create statement-level and row-level triggers in the database that will fire when you perform updates to records. Statement-level triggers fire once for each DML statement executed. Row-level triggers fire once for each record modified by the DML statement.
- ✦ PL/SQL programs can be combined into packages and stored together in the database.
- ✦ The data dictionary views USER_SOURCE and USER_OBJECTS provide useful information about stored PL/SQL programs.
- ✦ The data dictionary view USER_TRIGGERS provides useful information about database triggers.

✦ ✦ ✦

STUDY GUIDE

Now that you have learned about stored programs, you should test your understanding by reviewing the assessment questions and performing the exercises that follow.

Assessment Questions

1. Which data dictionary view contains the source code for PL/SQL procedures?

A. USER_SOURCE

B. USER_PROCEDURES

C. USER_PROGRAMS

D. USER_CODE

E. USER_OBJECTS

2. When you create a database trigger with the following specification, when will it fire?

CREATE OR REPLACE TRIGGER change_enrollment

BEFORE INSERT, UPDATE OF classid ON ClassEnrollment

A. When you delete a record from the ClassEnrollment table.

B. When you update the Price of a record in the ClassEnrollment table.

C. When you update the classid in the ScheduledClasses table.

D. When you insert a record into the ClassEnrollment table.

E. When you create the ClassEnrollment table.

3. When you include a user-defined a function in a SQL statement, which of the following is not allowed?

A. Executing a SELECT statement in the function

B. Executing an UPDATE statement in the function

C. Passing parameters to the function

D. Using an array in the function

E. Storing the function in the database

4. When you create a package called ENROLLMENT that contains a program called ADD_STUDENT, which of the following commands will run the program ADD_STUDENT?

A. RUN ADD_STUDENT

B. RUN ENROLLMENT.ADD_STUDENT

C. START ENROLLMENT.ADD_STUDENT

D. EXECUTE ADD_STUDENT

E. EXECUTE ENROLLMENT.ADD_STUDENT

5. When you have the following two triggers defined and you insert a record into the students table, which triggers will fire and in what order?

CREATE OR REPLACE TRIGGER all_changes

BEFORE INSERT, UPDATE, DELETE OF students

CREATE OR REPLACE TRIGGER row_updated

AFTER UPDATE of students

FOR EACH ROW

A. Statement-level trigger fires, then row-level trigger fires,

B. Only the statement-level trigger fires.

C. Only the rowleveltrigger fires.

D. Row-level trigger fires, then statement-level trigger fires.

E. No triggers fire.

6. You want to reference the record values in a row-level trigger with the prefixes BEFORE and AFTER. Which of the following clauses should you add to your trigger specification?

A. REFERENCING OLD AS :OLD NEW AS :NEW.

B. REFERENCING :BEFORE AND :AFTER.

C. REFERENCING PREVIOUS AS :BEFORE AND :NEW AS :AFTER.

D. REFERENCING OLD AS :BEFORE AND NEW AS :AFTER.

E. No changes required, :BEFORE and :AFTER are the default prefixes.

7. You want to create a procedure called ENROLL_STUDENT that accepts studentid and classid and returns a flag indicating success or failure. What type of parameters should you use for each parameter?

A. studentid IN NUMBER, classid IN NUMBER, flag OUT BOOLEAN

B. studentid IN NUMBER, classid OUT NUMBER, flag OUT BOOLEAN

C. studentid OUT NUMBER, classid OUT NUMBER, flag OUT BOOLEAN

D. studentid IN OUT NUMBER, classid IN OUT NUMBER, flag IN BOOLEAN

E. studentid IN NUMBER, classid IN NUMBER, flag IN BOOLEAN

8. You want to create a package that contains two public procedures called ADD_COURSE and SCHEDULE_COURSE and one private procedure called CHECK_LOCATION. Which procedures should be listed in the package specification?

   A. ADD_COURSE, SCHEDULE_COURSE and CHECK_LOCATION

   B. ADD_COURSE and SCHEDULE_COURSE

   C. ADD_COURSE and CHECK_LOCATION

   D. SCHEDULE_COURSE and CHECK_LOCATION

   E. CHECK_LOCATION

9. Which data dictionary view tells you whether a program needs to be recompiled with a status of VALID or INVALID?

   A. USER_SOURCE

   B. USER_OBJECTS

   C. USER_CODE

   D. USER_PROGRAMS

   E. USER_COMPILE

10. You want to call the procedure CHECK_STATUS from a PL/SQL program. CHECK_STATUS has two parameters. You pass in a classid, and the program returns a status of canceled, hold, or confirmed. Which line of code will call the CHECK_STATUS program successfully?

    A. EXECUTE CHECK_STATUS(v_studentid, v_status)

    B. v_status := CHECK_STATUS(v_studentid)

    C. CHECK_STATUS(v_studentid)

    D. RUN CHECK_STATUS(v_studentid, v_status)

    E. CHECK_STATUS(v_studentid, v_status)

Scenarios

1. You need to write a PL/SQL program called INSTRUCTOR_AVAILABILITY that will accept an instructor ID and class date. This program will check the ScheduledClasses table to see if the instructor is available to teach a class on the given data and will return an availability flag and the per diem cost for that instructor.

   A. Should INSTRUCTOR_AVAILABILITY be a procedure or a function?

   B. What parameters should be passed to the program?

   C. For each parameter, determine if the parameters should be IN, IN OUT, or OUT?

2. Your manager has found some inconsistencies in the financial report. The status of certain classes in the ScheduledClasses table is being updated after the class has run. She has asked you to write a database trigger that will prevent anyone from updating the status column in the ScheduledClasses table after the class has run (that is, the StartDate column of the class being updated is less than SYSDATE).
   A. On which table should you put the trigger?
   B. Should this trigger be a statement-level trigger or a row-level trigger?
   C. What event(s) should fire the trigger?
   D. What command should you use in the trigger to stop the user from changing a status and to return an error message to the user explaining why they cannot make the change?
3. You have been asked to make code changes to a procedure called STUDENT_ADDRESS. Unfortunately, the original SQL script used to create the program is nowhere to be found.
   A. Which data dictionary view contains the source code for the program?
   B. Write a SELECT statement that will return the program code for STUDENT_ADDRESS from that view.

Lab Exercises

Lab 13-1 Creating functions

1. Write the specification for a function called INSTRUCTOR_COST. The function will calculate the amount owed to an instructor for teaching a class. The function should accept an instructorid and a classid and should return a numeric value.
2. Add a SELECT statement to get the perdiemcost and the perdiemexpenses for the instructor.
3. Add a SELECT statement to get the daysduration of the class.
4. Add a calculation to determine total cost. Assume that totalcost = daysduration * perdiemcost + daysduration * perdiemexpenses.
5. Return the totalcost.
6. Call the function from the command line.
7. Call the function from a SQL statement.

Lab 13-2 Creating triggers

1. Write the specification for a trigger called CHECKDATE. CHECKDATE will prevent users from scheduling classes more than four months in advance. The trigger should go on the ScheduledClasses table. The trigger should fire before records are inserted or the StartDate column is updated. The trigger should be a row-level trigger
2. Use the ADD_MONTHS FUNCTION to return the date four months in the future.
3. Compare the date being added to the date four months in the future. If the new date is more than four months in the future, use RAISE_APPLICATION_ERROR to raise an error and prevent the change.
4. Write an UPDATE statement to UPDATE classid 53 with a startdate one year in the future.
5. What error message is returned?

Lab 13-3 Querying data dictionary views

1. Write a SELECT statement to return the specification and the code for the trigger CHECKDATE using the data dictionary view USER_TRIGGERS.
2. Write a SELECT statement to return the code for the function INSTRUCTOR_COST.

Answers to Chapter Questions

Chapter Pre-Test

1. Yes, PL/SQL programs can be stored in the database.
2. PL/SQL procedures can accept parameters and can return parameters. PL/SQL functions must always return a value.
3. Parameters enable you to pass values to and from your PL/SQL programs.
4. Statement-level triggers fire only once for each INSERT, UPDATE, or DELETE statement. Row-level triggers fire once for each record modified by the INSERT, UPDATE, or DELETE statement.
5. Putting programs into a package can improve performance and increase functionality.
6. The data dictionary views USER_SOURCE and USER_OBJECTS provide information about programs stored in the database.
7. A PL/SQL package is made up of a package specification and a package body.

8. IN parameters are passed to a program; OUT parameters are passed back from a program.
9. Yes, PL/SQL functions can be called from within SQL statements if they do not perform any updates, deletions, or insertions and they return a datatype recognized by SQL.
10. Yes, when you create a row-level trigger, you can access the values in the records being changed.

Assessment Questions

1. **A** — The data dictionary view USER_SOURCE contains the code for all PL/SQL programs stored in the database.
2. **D** — This trigger will fire when you insert a record into the ClassEnrollment table or when you update the classid of a record in the ClassEnrollment table.
3. **B** — UPDATE statements are not allowed in functions that are called from within SQL statements.
4. **E** — To call the program ADD_STUDENT within the package ENROLLMENT, you must use the EXECUTE command and begin the program name with the package name as a prefix.
5. **B** — Only the statement-level trigger will fire if you insert a record into the students table. The row-level trigger fires only when rows are updated.
6. **D** — You must change the default prefixes from :OLD and :NEW to :BEFORE and :AFTER using the REFERENCING clause.
7. **A** — Studentid and classid should be IN parameters because they are passing values in to the program; flag should be an OUT parameter because it is passing values back out from the program.
8. **B** — ADD_COURSE and SCHEDULE_COURSE should be listed in the package specification because they are public procedures. Private procedures are not listed in the package specification.
9. **B** — The USER_OBJECTS data dictionary view contains a column that gives you a status of VALID or INVALID, indicating whether the program should be recompiled.
10. **E** — You call the program by specifying the program name and passing two variables to the program: one to pass the studentid, one to receive the status.

Scenarios

1. INSTRUCTOR_AVAILABILITY should be a procedure because functions can return only one value, and this program must return two values: Availability flag and per diem cost. This program will have four parameters: InstructorId, ClassDate, AvailabilityFlag, and PerDiemCost. InstructorId and ClassDate are IN parameters. AvailabilityFlag and PerDiemCost are OUT parameters.

2. The database trigger should be on the ScheduledClasses table. This trigger must be a row-level trigger because you need to know information about the record being changed. This trigger should fire when the status column is updated. Within the code, you should use the RAISE_APPLICATION_ERROR command to prevent the user from making the change and return your specified error message.
3. The USER_SOURCE view contains the source code for the procedure STUDENT_ ADDRESS. The SELECT statement that will return the code for that procedure is

```
SELECT text
  FROM user_source
 WHERE name = 'STUDENT_ADDRESS'
 ORDER BY line
/
```

Lab Exercises

Lab 13-1 Creating functions

Write the function INSTRUCTOR_COST:

```
CREATE OR REPLACE FUNCTION INSTRUCTOR_COST
(p_instructorid IN NUMBER, p_classid IN NUMBER)
RETURN NUMBER
IS
   v_perdiemcost         instructors.perdiemcost%TYPE;
   v_perdiemexpenses     instructors.perdiemexpenses%TYPE;
   v_daysduration        scheduledclasses.daysduration%TYPE;
   v_totalcost           NUMBER;
BEGIN
   /* SELECT instructor perdiemcost and perdiemexpenses */
   SELECT perdiemcost, perdiemexpenses
     INTO v_perdiemcost, v_perdiemexpenses
    FROM instructors
    WHERE instructorid = p_instructorid;

   /* SELECT class duration */
   SELECT daysduration
     INTO v_daysduration
     FROM scheduledclasses
    WHERE classid = p_classid;

   /* Calculate total instructor cost */
   v_totalcost := v_daysduration * NVL(v_perdiemcost,0) +
                  v_daysduration * NVL(v_perdiemexpenses,0);

   /* Return total instructor cost */
   RETURN (v_totalcost);
END;
/
```

Execute the program from SQL*Plus:

```
SQL> VARIABLE totalcost NUMBER
SQL> EXECUTE :totalcost := INSTRUCTOR_COST(100,50);

PL/SQL procedure successfully completed.

SQL> PRINT totalcost

TOTALCOST
---------
     3200
```

Execute the function from a SQL statement:

```
SQL> SELECT classid, instructorid,
INSTRUCTOR_COST(instructorid, classid)
  2  FROM scheduledclasses
  3  WHERE startdate > '01-JAN-2000'
  4  /

  CLASSID INSTRUCTORID INSTRUCTOR_COST(INSTRUCTORID,CLASSID)
--------- ------------ ------------------------------------
       50          100                                 3200
       51          200                                 5000
       53          110                                 2800
```

Lab 13-2 Creating triggers

Write the trigger CHECKDATE:

```
CREATE OR REPLACE TRIGGER checkdate
BEFORE INSERT OR UPDATE OF startdate ON ScheduledClasses
FOR EACH ROW
DECLARE
   v_date_limit DATE;
BEGIN
   v_date_limit := ADD_MONTHS(sysdate,4);
   IF :NEW.startdate > v_date_limit THEN
        RAISE_APPLICATION_ERROR(-20001,'Cannot schedule class
more than 4 months in future.');
   END IF;
END;
/
```

Try to update scheduledclasses class 53 to a date one year in the future:

```
SQL> UPDATE scheduledclasses
  2  SET startdate = '01-JAN-2004'
  3  WHERE classid = 53
  4  /
```

```
UPDATE scheduledclasses
       *
ERROR at line 1:
ORA-20001: Cannot schedule class more than 4 months in future.
ORA-06512: at "STUDENT3.CHECKDATE", line 6
ORA-04088: error during execution of trigger
'STUDENT3.CHECKDATE'
```

Lab 13-3 Querying data dictionary views

1. Write a SELECT statement to get the code and specification for the trigger CHECKDATE:

```
SQL> SELECT description, trigger_body
  2  FROM user_triggers
  3  WHERE trigger_name = 'CHECKDATE'
  4  /

DESCRIPTION
-------------------------------------------------------------
checkdate
BEFORE INSERT OR UPDATE OF startdate ON ScheduledClasses
FOR EACH ROW

TRIGGER_BODY
-------------------------------------------------------------
DECLARE
 v_date_limit DATE;
BEGIN
 v_date_limit := ADD_MONTHS(sysdate,4);
 IF :NEW.startdate > v_date_limit THEN
     RAISE_APPLICATION_ERROR(-20001,'Cannot schedule class more
     than 4 months in future.');
 END IF;
END;
```

2. Write a SELECT statement to get the code for the function INSTRUCTOR_COST:

```
SQL> SELECT text
  2  FROM user_source
  3  WHERE name = 'INSTRUCTOR_COST'
  4  ORDER BY line
  5  /

TEXT
-----------------------------------------------
FUNCTION INSTRUCTOR_COST
(p_instructorid IN NUMBER, p_classid IN NUMBER)
RETURN NUMBER
```

```
IS
   v_perdiemcost         instructors.perdiemcost%TYPE;
   v_perdiemexpenses     instructors.perdiemexpenses%TYPE;
   v_daysduration        scheduledclasses.daysduration%TYPE;
   v_totalcost           NUMBER;
BEGIN
   /* SELECT instructor perdiemcost and perdiemexpenses */
   SELECT perdiemcost, perdiemexpenses
    INTO v_perdiemcost, v_perdiemexpenses
     FROM instructors
   WHERE instructorid = p_instructorid;

   /* SELECT class duration */
   SELECT daysduration
     INTO v_daysduration
     FROM scheduledclasses
    WHERE classid = p_classid;

   /* Calculate total instructor cost */
   v_totalcost := v_daysduration * NVL(v_perdiemcost,0) +
                  v_daysduration * NVL(v_perdiemexpenses,0);

   /* Return total instructor cost */
   RETURN (v_totalcost);
END;
```

APPENDIX A

What's on the CD-ROM?

This appendix provides information on the contents of the CD-ROM that accompanies this book.

The CD contains 12 programs. Some of the applications are full working versions of software, while many others are evaluation or trial versions of some of the most useful Oracle utilities.

The software included on the CD-ROM is:

- Adobe Acrobat Reader 5.0
- Bible Series Certification Test Engine and "Introduction to Oracle: SQL & PL/SQL" sample exam from Hungry Minds
- Introduction to Oracle: SQL & PL/SQL (1Z0-001) Prep Exam from Self Test Software (Trial Version)
- Aria ZIM 2.1 and Oracle SAM from ZIM Technologies (Evaluation Version)
- ER/Studio 4.21 from Embarcadero Technologies (Trial Version)
- Rapid/SQL 5.6 from Embarcadero Technologies (Trial Version)
- PL/Formatter 3.1.2 from RevealNet (Trial Version)
- Knowledge Base for Active PL/SQL from RevealNet, which includes Instant Message for Oracle v2.4 (Trial Version)
- SQL Programmer 2001 from Sylvain Faust International (Trial Version)

Also included are scripts and source code examples from the book and an electronic, searchable version of the book that can be viewed with Adobe Acrobat Reader.

The CD-ROM does *not* include Oracle 8*i*. If you do not have a copy of Oracle 8*i* Enterprise Edition, which is recommended to successfully complete the labs, you can download a trial copy from Oracle's Web site after joining the Oracle Technology Network at http://technet.oracle.com. The link to the download location is `http://technet.oracle.com/software/content.html`. You can also order a trial copy of Oracle 8*i* Enterprise Edition for free (if you live in the United States) from the Oracle Store at http://store.oracle.com. Finally, if you do not want to download the software or want to receive free updates for a year and CD-ROMs with the Oracle software on them, you can order a Technology Track through the Oracle Technology network at `http://technet.oracle.com/software/track.html`. Technology Tracks cost $200 each (at the time of writing) and are available for Windows NT/2000, Linux, and Sun SPARC platforms.

System Requirements

Make sure that your computer meets the minimum system requirements listed in this section. If your computer doesn't match up to most of these requirements, you may have a problem using the contents of the CD.

For Microsoft Windows NT (Service Pack 4 or later) or Windows 2000:

- ✦ PC with a Pentium II processor running at 300 MHz or faster
- ✦ At least 64MB of RAM
- ✦ At least 2GB of free hard disk space
- ✦ A CD-ROM drive

Although many of the software products included on the CD work with Windows 98 or Windows ME, if you want to install them and Oracle 8*i* Enterprise Edition on the same computer, you need either Windows NT or Windows 2000.

For Linux:

- ✦ PC with a Pentium II processor running at 300 MHz or faster
- ✦ At least 64MB of RAM
- ✦ At least 2GB of free hard disk space
- ✦ A CD-ROM drive

Even though the software included on the CD-ROM runs in Windows environments only, you can make use of the scripts and sample code on a Linux computer that has Oracle 8*i* Enterprise Edition for Linux installed.

Using the CD with Microsoft Windows

To install the items from the CD to your hard drive, follow these steps:

1. Insert the CD into your computer's CD-ROM drive.
2. Click Start ➪ Run.
3. In the dialog box that appears, type ***d*:\setup.exe**, where *d* is the letter of your CD-ROM drive.
4. Click OK.

Using the CD with Linux

To install the items from the CD to your hard drive, follow these steps:

1. Log in as root.
2. Insert the CD into your computer's CD-ROM drive.
3. Mount the CD-ROM.
4. Launch a graphical file manager.

What's on the CD

The CD-ROM contains source code examples, applications, and an electronic version of the book. Following is a summary of the contents of the CD-ROM arranged by category.

Source code

The scripts required to create the Student schema, used by the labs in each chapter of the book, in your database can be found in the SCRIPTS folder at the root of the CD-ROM. In the SCRIPTS folder, you will also find a SOLUTIONS folder, which lists the solutions for some of the more complex lab questions. In some cases, labs also require that you run specific scripts, which can be found in the SCRIPTS folder.

Applications

The SOFTWARE folder contains all of the trial and other software referenced earlier in this appendix. Each application is housed in its own folder within the SOFTWARE folder, as follows (product descriptions are provided by the respective vendor):

- **SELFTEST:** Introduction to Oracle: SQL & PL/SQL (1Z0-001) Prep Exam from Self Test Software (Trial Version). This program enables you to closely simulate the experience of taking the exam. The version provided is an evaluation of a version of the exam that you can purchase from the vendor and includes a smaller number of questions than the actual retail product.
- **ACROBAT:** Adobe Acrobat Reader 5.0. This is a fully functional version of the Adobe Acrobat Reader 5.0 that can be used to read the electronic version of this book provided on the CD-ROM in the HUNGRYMINDS folder.
- **TESTEXAM:** Bible Series Certification Test Engine and "Introduction to Oracle: SQL & PL/SQL" sample exam from Hungry Minds. This test engine and exam include 300 questions that can be used to prepare for the exam. This software is a fully working version.
- **ARIAZIM:** Aria ZIM 2.1 and Oracle SAM from ZIM Technologies (Evaluation Version). The Oracle Server Access Module (SAM) is in the ORASAM folder within the ARIAZIM folder of the CD-ROM. ZIM is a powerful and flexible environment for developing and deploying all types of database applications. ZIM's entity-relationship model and fully integrated Object Dictionary permit progressive program development, whether your information processing system is among the simplest or the most complex. The straightforward, English-like syntax of the Application Development Language and its provision for customized user interfaces and programs, permit you to make any application easy to use.

  ZIM has the added capability of manipulating and retrieving data from third-party data sources as well as ZIM's own database. Client applications for SQL database servers are designed and developed as complete ZIM application systems. The objective of the ZIM product is 100 percent source code portability of applications including seamless access to databases being managed by SQL relational database management systems (SQL servers) supplied by independent vendors. A ZIM client/server database application may consist of an arbitrary mixture of tables managed by an SQL server and entity sets and data relationships managed by ZIM. The Oracle SAM enables you to use ZIM with an Oracle database.

  After you install the software, you must contact ZIM Technologies to activate it. Instructions are provided in the ARIAZIM folder on the CD-ROM.
- **ERSTUDIO:** ER/Studio 4.21 from Embarcadero Technologies (Trial Version). ER/Studio is a data modeling application for logical and physical database design and construction. Its powerful, multi-level design environment addresses the everyday needs of database administrators (DBAs), developers, and data architects who build and maintain large, complex database applications.

ER/Studio's progressive interface and processes have been logically organized to effectively address the ease-of-use issues that have plagued data modeling tools for the past decade. The application equips the user to create, understand, and manage mission-critical database designs within an enterprise. It offers strong logical design capabilities, bi-directional synchronization of logical and physical designs, automatic database construction, Java application generation, accurate reverse-engineering of databases, and powerful HTML-based documentation and reporting facilities.

- **RAPIDSQL:** Rapid/SQL 5.6 from Embarcadero Technologies (Trial Version). Rapid SQL is an integrated development environment that enables developers to create, edit, version, tune, and deploy server-side objects residing on Oracle, Microsoft SQL Server, IBM DB2, and Sybase databases. Its HTML and Java programming facilities create a unified development environment for database and web programming, while its extensive graphical facilities simplify SQL scripting, object management, reverse-engineering, database project management, version control, and schema deployment. With Rapid SQL, programmers can develop and maintain high-quality, high-performance client/server and Web-based applications in less time and with greater accuracy.
- **PLFORMAT:** PL/Formatter 3.1.2 from RevealNet (Trial Version). Formatter Plus is a powerful and flexible tool for analyzing and formatting entire PL/SQL applications. You get instant code formatting for an entire application or just a single file. You get instant code reviews — while you code — with best-practice recommendations for code correctness, maintainability, efficiency, readability, and program structure. You get best-practice technical knowledge, coupled with real-world examples.
- **PLSQLKNOW:** Knowledge Base for Active PL/SQL from RevealNet (Trial Version). RevealNet's Knowledge Base for Active PL/SQL combines 1,600 topics on PL/SQL with an extensive code library of over 1,000 PL/SQL functions. It merges expert knowledge of PL/SQL with a vast library of best-practice functions — with source code — that extend the power of the PL/SQL language. Now with V2001.1, the Knowledge Base for Active PL/SQL extends its comprehensive coverage to give you code advice and recommendations. And it's all only a keystroke away while you are developing your applications.

  Knowledge Base for Active PL/SQL includes Instant Message for Oracle v2.4. RevealNet's Instant Message lookup is the fastest way to access over 25,000 Oracle messages — including Oracle 8*i* error messages. Within seconds of encountering an error message, you can access the full description with recommended actions.
- **SQLPROG:** SQL Programmer 2001 from Sylvain Faust International (Trial Version). SQL-Programmer 2001 provides a complete development environment for Oracle PL/SQL, Sybase, and Microsoft Transact-SQL programmers. Its intuitive user interface and extensive toolset enable easy creation, management, and testing of all programmable objects such as stored procedures, triggers, functions, views, and indexes. With SQL Programmer 2001, database developers and DBAs are more productive, can find and fix bugs faster, improve the quality of applications, and better manage business-critical information.

Shareware programs are fully functional, free trial versions of copyrighted programs. If you like particular programs, register with their authors for a nominal fee and receive licenses, enhanced versions, and technical support.

Freeware programs are free, copyrighted games, applications, and utilities. You can copy them to as many PCs as you like — free — but they do not provide technical support.

GNU software is governed by its own license, which is included inside the folder of the GNU software. Distribution of this software is unrestricted. See the GNU license for more details.

Trial, demo, or evaluation versions are usually limited either by time or functionality (such as being unable to save projects).

Electronic version of *Oracle8i DBA: SQL and PL/SQL Certification Bible*

The complete (and searchable) text of this book is on the CD-ROM in Adobe's Portable Document Format (PDF), readable with the Adobe Acrobat Reader (also included). For more information on Adobe Acrobat Reader, go to `www.adobe.com`.

Troubleshooting

If you have difficulty installing or using the CD-ROM programs, try the following solutions:

- **Turn off any anti-virus software that you may have running.** Installers sometimes mimic virus activity and can make your computer incorrectly believe that it is being infected by a virus. (Be sure to turn the anti-virus software back on later.)
- **Close all running programs.** The more programs you're running, the less memory is available to other programs. Installers also typically update files and programs; if other programs are running, installation may not work properly.

If you still have trouble with the CD, call the Hungry Minds Worldwide Customer Service phone number: (800) 762-2974; outside the United States, call (317) 572-3993. Hungry Minds provides technical support only for installation and other general quality-control issues; for technical support on the applications themselves, consult the program's vendor or author.

✦ ✦ ✦

APPENDIX B

Practice Exam

The "Introduction to Oracle: SQL and PL/SQL" exam has 57 questions. You have 120 minutes to answer all the questions. All the questions are multiple choice. To answer certain questions you are given extra information that can be brought up in a separate window by pressing a button. Read the entire question, review any extra information, and read all answers before selecting your answer. When you take the exam, you can skip over questions and return to answer them later., You can also return to review questions you have already answered. This practice exam will help you gauge how well prepared you are to take the "Introduction to Oracle: SQL and PL/SQL" exam. Answers to the practice exam and explanations are provided at the end of this appendix.

1. The boss is giving all employees a 20 percent raise on their annual salary and a $200 bonus. The employee table stores monthly salaries and contains the following columns:

```
EMPLOYEE
Name                    Null?    Type
------------------------------- --------
ID                      NOT NULL NUMBER(9)
NAME                             VARCHAR2(40)
SALARY                           NUMBER(10,2)
```

Which of the following SELECT statements will return a list of each employee's ID, name, and total compensation for the year including raise and bonus.

A. SELECT id,name,(salary+200)+(salary*.2) "Total compensation"

FROM employee;

B. SELECT id, name, salary*.2+200 "Total compensation"

FROM employee;

C. SELECT id, name, (salary*12*.2)+200 “Total compensation”
FROM employee;

D. SELECT id, name, (salary*12*.2) + (salary*12+200) “Total compensation
FROM employee;

E. SELECT id, name, salary*12*.2+salary*(12+200) “Total compensation”
FROM employee;

2. The email address column in the employee table should be a NOT NULL column. If all the existing records in the table have a value specifed for email address, which of the following commands will make the email address column NOT NULL?

```
EMPLOYEE
 Name                           Null?    Type
 ------------------------------ -------- ----
 ID                             NOT NULL NUMBER(9)
 NAME                                    VARCHAR2(40)
 SALARY                                  NUMBER(10,2)
 EMAIL                                   VARCHAR2(50)
```

A. ALTER TABLE employee(email) NOT NULL;

B. MODIFY TABLE employee ADD CONSTRAINT employee_email_nn NOT NULL(email);

C. ALTER TABLE employee ADD CONSTRAINT employee_email_nn NOT NULL(email);

D. ALTER TABLE employee ADD NOT NULL(email);

E. ALTER TABLE employee MODIFY (email NOT NULL);

3. The following statements are executed in order:

```
INSERT INTO dept VALUES (10,'Admin', null);
SAVEPOINT A;
INSERT INTO dept VALUES (20,'Finance','USA');
SAVEPOINT B;
INSERT INTO emp VALUES (45, 'Smith', 500,
'smith@produce.com');
ROLLBACK TO SAVEPOINT B;
SAVEPOINT C;
INSERT INTO emp VALUES (51, 'Jones', 400, null);
COMMIT;
INSERT INTO dept VALUES(30,'Billing','USA');
ROLLBACK;
COMMIT;
```

Which of these rows will be saved to the database? Select all that apply.

A. INSERT INTO dept VALUES (10,'Admin', null);

B. INSERT INTO dept VALUES (20,'Finance','USA');

C. INSERT INTO emp VALUES (45, 'Smith', 500, 'smith@produce.com');

D. INSERT INTO emp VALUES (51, 'Jones', 400, null);

E. INSERT INTO dept VALUES(30,'Billing','USA');

4. Employee names are stored in mixed case in the database.

```
EMPLOYEE
 Name                              Null?    Type
 --------------------------------- -------- ----
 ID                                NOT NULL NUMBER(9)
 NAME                                       VARCHAR2(40)
 SALARY                                     NUMBER(10,2)
 EMAIL                                      VARCHAR2(50)
```

Which of the following SELECT statements will return the employee with the name "Smith"?

A. SELECT * FROM employee WHERE UPPER(name) = 'SMITH';

B. SELECT * FROM employee WHERE name LIKE '%SMITH%';

C. SELECT * FROM employee WHERE name = UPPER('SMITH');

D. SELECT * FROM employee WHERE name = 'SMITH'

E. SELECT * FROM employee WHERE name = "Smith"

5. Which two of the following PL/SQL variable declarations are invalid?

```
EMPLOYEE
 Name                              Null?    Type
 --------------------------------- -------- ----
 ID                                NOT NULL NUMBER(9)
 NAME                                       VARCHAR2(40)
 SALARY                                     NUMBER(10,2)

 EMAIL                                      VARCHAR2(50)
```

A. v_id NUMBER(9) := 0;

B. name VARCHAR2(40);

C. v_salary employee.salary%TYPE;

D. v_email VARCHAR2(50) NOT NULL;

E. v_name name.employee%TYPE;

6. What will be the value of the Boolean variable v_flag after the following PL/SQL code is executed?

```
DECLARE
     v_flag BOOLEAN;
     v_sold BOOLEAN := TRUE;
     v_paid BOOLEAN;
BEGIN
     v_flag := v_sold AND v_paid;
END;
```

A. TRUE

B. FALSE

C. NULL

D. 0

7. Which of the following SELECT statements will select all the employees with a NULL salary?

```
EMPLOYEE
 Name                            Null?    Type
 ------------------------------- -------- ----
 ID                              NOT NULL NUMBER(9)
 NAME                                     VARCHAR2(40)
 SALARY                                   NUMBER(10,2)
 EMAIL                                    VARCHAR2(50)
```

A. SELECT * FROM employee WHERE salary = NULL;

B. SELECT * FROM employee WHERE salary IS NULL;

C. SELECT * FROM employee WHERE salary IS 'NULL';

D. SELECT * FROM employee WHERE salary = 0;

E. SELECT * FROM employee WHERE salary = 'NULL';

8. What will happen when you execute the following subquery?

```
SELECT ename, salary
FROM employee
WHERE salary IN (SELECT salary
                 FROM employee
                 WHERE ename = 'SMITH');
```

A. You will get an error because you should not have parentheses around the subquery.

B. The query will give you an error if there is more than one employee with the name "Smith."

C. The query will execute successfully and list the names and salaries of all employees with the same salary as anyone named "Smith."

D. The query will execute successfully and list the names and salaries of all employees with the same salary as anyone named "Smith" except "Smith" himself.

E. The query will give you an error if there is only one employee named "Smith" because you are using a multi-row operator.

9. Which of these commands will create a variable called "p_id" and will allow the user to enter a value for the variable?

A. ACCEPT p_id NUMBER PROMPT "Enter a value for p_id: "

B. DEFINE p_id PROMPT "Enter a value for p_id: "

C. ACCEPT p_id NUMBER MESSAGE "Enter a value for p_id"

D. DEFINE p_id MESSAGE "Enter a value for p_id"

10. When will this PL/SQL program exit the loop?

```
DECLARE
      CURSOR dept_cursor IS
      SELECT id, name FROM dept;
BEGIN
     FOR dept_record IN dept_cursor LOOP
           EXIT WHEN dept_cursor%ROWCOUNT =5;
           DBMS_OUTPUT.PUT_LINE(dept_record.name);
     END LOOP;
END;
```

A. After all the rows in the cursor are processed.

B. Never, it is an infinite loop.

C. After five rows in the cursor are processed.

D. It will exit the loop immediately if the cursor contains exactly five rows.

E. After all the rows in the cursor are processed, or five rows are processed, whichever comes first.

11. Which two messages will be printed on the screen if the SELECT statement in the following PL/SQL program returns no rows?

```
DECLARE
     v_name employee.name%TYPE;
BEGIN
     BEGIN
           SELECT name
             INTO v_name
             FROM employee
            WHERE id = 10;
     EXCEPTION
           WHEN no_data_found THEN
                DBMS_OUTPUT.PUT_LINE('No data found');
           WHEN too_many_rows THEN
                DBMS_OUTPUT.PUT_LINE('too many rows');
      END;

      DBMS_OUTPUT.PUT_LINE('No errors');
```

```
EXCEPTION
      WHEN no_data_found THEN
           DBMS_OUTPUT.PUT_LINE('Still no data found');
      WHEN others THEN
           DBMS_OUTPUT.PUT_LINE('An error occurred');
END;
```

A. No data found.

B. Still no data found.

C. An error occurred.

D. No errors.

E. Too many rows.

12. Which command will give the user DAVE the ability to write a SELECT statement to return data from the employee table in SCOTT's schema?

A. GRANT SELECT ON dave.employee TO scott;

B. CREATE SYNONYM employee FOR scott.employee;

C. CREATE PUBLIC SYNONYM employee FOR scott.employee;

D. GRANT READ ON scott.employee TO dave WITH GRANT OPTION;

E. GRANT SELECT on scott.employee TO dave;

13. Which of the following SELECT statements will list the total salary of employees in each department with total salaries over $5,000?

A. SELECT SUM(salary), dept_id
FROM employee
HAVING SUM(salary) > 5000;

B. SELECT SUM(salary), dept_id
FROM employee
GROUP BY dept_id
WHERE SUM(salary) > 5000;

C. SELECT SUM(salary), dept_id
FROM employee
WHERE SUM(salary) > 5000
GROUP BY dept_id;

D. SELECT SUM(salary), dept_id
FROM employee
GROUP BY dept_id
HAVING SUM(salary) > 5000;

E. SELECT SUM(salary), dept_id

FROM employee

WHERE SUM(salary) > 5000;

14. Which of the following SELECT statements will list all the departments and the employees in each department including departments with no employees?

A. SELECT d.name, e.name

FROM employee e, dept d

WHERE e.dept_id = d.id

B. SELECT d.name, e.name

FROM employee e, dept d

WHERE e.dept_id = d.id (+)

C. SELECT d.name, e.name

FROM employee e, dept d

WHERE e.dept_id (+) = d.id

D. SELECT d.name, e.name

FROM employee e, dept d

WHERE e.dept_id (+) = d.id (+)

E. SELECT d.name, e.name

FROM employee e, dept d

WHERE e.dept_id IN d.id

15. Which of the following commands will successfully create the following table? The ID column should be the primary key of the table.

```
EMPLOYEE
 Name                           Null?    Type
 ------------------------------ -------- ----
 ID                             NOT NULL NUMBER(9)
 NAME                                    VARCHAR2(40)
 SALARY                                  NUMBER(10,2)

 EMAIL                                   VARCHAR2(50)
```

A. CREATE TABLE employee

(id NUMBER(9)

name VARCHAR2(40)

salary NUMBER(10,2)

email VARCHAR2(50)

CONSTRAINT employee_id_pk PRIMARY KEY(id));

B. CREATE TABLE employee
(id NUMBER(9),
name VARCHAR2(40),
salary NUMBER(10,2),
email VARCHAR2(50),
CONSTRAINT employee_id_pk PRIMARY KEY(id));

C. CREATE TABLE employee
(id NUMBER(9),
CONSTRAINT employee_id_pk PRIMARY KEY
name VARCHAR2(40),
salary NUMBER(10,2),
email VARCHAR2(50));

D. CREATE TABLE employee
(id NUMBER(9)
CONSTRAINT employee_id_pk PRIMARY KEY(id)
name VARCHAR2(40),
salary NUMBER(10,2),
email VARCHAR2(50));

E. CREATE TABLE employee
(id NUMBER(9)
name VARCHAR2(40),
salary NUMBER(10,2),
email VARCHAR2(50),
CONSTRAINT employee_id_pk PRIMARY KEY);

16. Which of the following commands or set of commands will create the record emp_record used in the following PL/SQL program? The SELECT statement selects from the following table:

```
EMPLOYEE
 Name                           Null?    Type
 ------------------------------ -------- ----
 ID                             NOT NULL NUMBER(9)
 NAME                                    VARCHAR2(40)
 SALARY                                  NUMBER(10,2)

 EMAIL                                   VARCHAR2(50)
```

```
BEGIN
       SELECT name, salary
  INTO emp_record
  FROM employee
 WHERE id = 500;
END;
```

A. TYPE emp_record IS RECORD
(name employee.name%TYPE,
salary employee.salary%TYPE);

B. emp_record employee%ROWTYPE;

C. emp_record employee%TYPE;

D. TYPE emp_record_type IS RECORD
(salary employee.salary%TYPE,
name employee.name%TYPE);
emp_record emp_record_type;

E. TYPE emp_record IS RECORD
(name employee.name%TYPE,
salary employee.salary%TYPE);
emp_record emp_record_type;

17. Which of these statements is true about the following DELETE statement?

```
DELETE FROM employee;
```

A. The statement cannot be rolled back.

B. All rows in the employee table will be deleted.

C. The employee table will be dropped.

D. The statement will not execute without a WHERE clause.

E. This statement will be automatically committed to the database.

18. Which message will be printed on the screen if the UPDATE statement in the following PL/SQL program updates no rows?

```
DECLARE
     v_name employee.name%TYPE := 'Davis'
BEGIN
     UPDATE employee
       SET name = v_name
      WHERE id = 10;

     IF SQL%NOTFOUND THEN
           RAISE_APPLICATION_ERROR
                (-20001,'NOTFOUND is true');
     END IF;
```

```
     DBMS_OUTPUT.PUT_LINE('No errors');
EXCEPTION
     WHEN no_data_found THEN
          DBMS_OUTPUT.PUT_LINE('No data found');
     WHEN too_many_rows THEN
          DBMS_OUTPUT.PUT_LINE('Too many rows');
END;
```

A. NOTFOUND is true.

B. No errors.

C. No data found.

D. Too many rows.

E. No message is displayed.

19. Which of the following ORDER BY clauses will sort the data returned by the SELECT statement by department ID in descending order and by salary in descending order for employees in the same department?

```
SELECT id, name, salary, dept_id AS department
  FROM employee
```

A. ORDER BY dept_id, salary

B. ORDER BY dept_id, salary DESC

C. ORDER BY department, salary DESC

D. ORDER BY department DESC, salary DESC

E. ORDER BY salary DESC, dept_id DESC

20. What number will be returned by the SELECT orderid_seq.CURRVAL statement, if the following statements are executed in order?

```
CREATE OR REPLACE SEQUENCE orderid_seq
START WITH 10
INCREMENT BY 1
MAXVALUE 999
MINVALUE 1
CACHE 5;

SELECT orderid_seq.NEXTVAL
FROM dual;

SELECT orderid_seq.CURRVAL
FROM dual;
```

A. 1

B. 10

C. 11

D. 12

E. 15

21. How many times will the statement "I am in the loop" be displayed on the screen when you execute the following PL/SQL code?

```
DECLARE
     v_counter  NUMBER :=1;
BEGIN
     WHILE v_counter <1 LOOP
           DBMS_OUTPUT.PUT_LINE('I am in the loop');
           v_counter := v_counter + 1;
           EXIT WHEN v_counter >5;
     END LOOP;
END;
```

A. Zero

B. Once

C. Four times

D. Five times

E. Infinitely

22. Which of the following statements will return today's date in the following format: 12 June, 2001?

A. SELECT currentdate

FROM DUAL;

B. SELECT sysdate;

C. SELECT TO_CHAR(sysdate,'fmdd Month, YYYY')

FROM dual;

D. SELECT TO_DATE(sysdate,'fmdd Month, YYYY')

FROM dual;

E. SELECT FORMAT sysdate 'fmdd Month, YYYY'

FROM dual;

23. What is the maximum size of a column declared as VARCHAR2 in an Oracle 8i database?

A. 100 characters

B. 1000 characters

C. 2000 characters

D. 4000 characters

E. 2 gigabytes

24. Which data dictionary view lists only the tables you own?

A. USER_TABLES

B. ALL_TABLES

C. DBA_TABLES

D. USER_VIEWS

E. ALL_VIEWS

25. Which of the following statements will successfully create a column alias for salary*12? Pick two.

A. SELECT id, salary*12 AS ANNUAL SALARY FROM emp;

B. SELECT id, salary*12 ANNUAL SALARY FROM emp;

C. SELECT id, salary*12 'ANNUAL SALARY' FROM emp;

D. SELECT id, salary*12 AS Annual_Salary FROM emp;

E. SELECT id, salary*12 "ANNUAL SALARY" FROM emp;

26. What datatype should you choose for a column called "notes" that will hold up to 10,000 characters of data?

A. CHAR

B. VARCHAR2

C. BLOB

D. CLOB

E. BFILE

27. Which of the following commands will change the password for the user scott to "newpass"?

A. ALTER USER scott MODIFY PASSWORD newpass;

B. ALTER USER scott IDENTIFIED BY newpass;

C. MODIFY USER scott ALTER PASSWORD newpass;

D. UPDATE scott SET PASSWORD = 'newpass';

E. UPDATE USER_PASSWORDS SET PASSWORD = 'newpass' WHERE USER = 'SCOTT'

28. Which of the following keywords are not required in a PL/SQL program? Choose all that apply.

A. DECLARE

B. BEGIN

C. EXCEPTION

D. END

29. In the following program, which lines of code can be removed because their functions are performed by the cursor FOR loop? Choose all that apply.

```
DECLARE
     CURSOR emp_cursor
     IS SELECT id, name, salary
     FROM emp
     ORDER BY salary DESC;

     emp_record     emp_cursor%ROWTYPE;
BEGIN
     OPEN emp_cursor

     FOR emp_record IN emp_cursor LOOP
           FETCH emp_cursor INTO emp_record;

           EXIT WHEN emp_cursor%NOTFOUND;

           DBMS_OUTPUT.PUT_LINE(emp_record.ename);
     END LOOP;

END;
```

A. emp_record emp_cursor%ROWTYPE;

B. OPEN emp_cursor;

C. FETCH emp_cursor INTO emp_record;

D. EXIT WHEN emp_cursor%NOTFOUND;

E. END LOOP;

30. Which of the following are SQL*Plus commands? Choose all that apply.

A. DESCRIBE

B. SELECT

C. SPOOL

D. RUN

E. UPDATE

31. The following SELECT statement is executed against a table containing the following data. How many rows will be returned by the SELECT statement?

```
SELECT id, name
  FROM emp
 WHERE deptno = 10
    OR deptno = 20
   AND salary > 1000;
```

```
ID   NAME       SALARY     DEPTNO
---- ---------  ---------  -------
1    JONES      1000       10
2    SMITH      1000       20
3    DAVIS      2000       10
4    CUTLER     500        10
5    MICHAELS   2000       20
```

A. One row

B. Two rows

C. Three rows

D. Four rows

E. Five rows

32. What will be displayed for employee SMITH when the following SELECT statement is executed?

```
SELECT id, name, NVL(salary,'No Salary') "Salary"
  FROM emp;

ID   NAME       SALARY     DEPTNO
---- ---------  ---------  -------
1    JONES      1000       10
2    SMITH                 20
3    DAVIS      2000       10
4    CUTLER     500        10
5    MICHAELS   2000       20
```

A. 2 SMITH 0

B. 2 SMITH

C. 2 SMITH No Salary

D. 2 SMITH Salary

E. This select statement will return an error.

33. What will be returned by the following SELECT statement?

```
SELECT id, name,
  TO_CHAR(ADD_MONTHS(hiredate,3),'dd/mm/yyyy')
  FROM emp
  WHERE id = 1;

ID    NAME       SALARY     HIREDATE
----  ---------  ---------  -------
1     JONES      1000       01-DEC-2001
2     SMITH                 03-FEB-2000
```

A. 1 JONES 01/12/2001

B. 1 JONES 04/12/2001

C. 1 JONES 01/03/2001

D. 1 JONES 01/03/2002

E. This select statement will return an error.

34. What will be returned by the following SELECT statement?

```
SELECT AVG(salary)
  FROM emp;

ID    NAME       SALARY
----  ---------  ---------
1     JONES      1000
2     SMITH
3     MICHAELS   2000
```

A. 1000

B. 1500

C. 2000

D. 3000

E. NULL

35. Consider the data in the following tables:

```
EMPLOYEE
-------------
ID    NAME       SALARY     DEPT_ID
----  ---------  ---------  -------
1     JONES      1000       10
2     SMITH      1000       20
3     DAVIS      2000       10
4     CUTLER     500        10
5     MICHAELS   2000       20

DEPARTMENT
-----------
ID    NAME
----  ---------
10    FINANCE
20    ADMINISTRATION
30    HUMAN RESOURCES
40    TRAINING
```

Which of the SELECT statements would return the following data?

```
ID   DEPT_NAME       EMP_NAME
---- ------------    ---------
10   FINANCE         JONES
10   FINANCE         DAVIS
10   FINANCE         CUTLER
20   ADMINISTRATION  SMITH
20   ADMINISTRATION  MICHAELS
30   HUMAN RESOURCES
40   TRAINING
```

A. SELECT d.id, d.name DEPT_NAME, e.name EMP_NAME
FROM employee e, department d
WHERE e.id (+) = d.id;

B. SELECT d.id, d.name DEPT_NAME, e.name EMP_NAME
FROM employee e, department d
WHERE e.id = d.id (+);

C. SELECT d.id, d.name DEPT_NAME, e.name EMP_NAME
FROM employee e, department d
WHERE e.dept_id = d.id (+);

D. SELECT d.id, d.name DEPT_NAME, e.name EMP_NAME
FROM employee e, department d
WHERE e.dept_id (+) = d.id;

36. How many rows will be returned by the following query if the table contains the following data?

```
SELECT id, name, salary, commission
FROM employee
WHERE commission = (SELECT commission
                    FROM employee
                    WHERE name = 'SMITH');
EMPLOYEE
-------------
ID   NAME       SALARY     COMMISSION      DEPTNO
---- ---------  ---------  ----------      -------
1    JONES      1000       15              10
2    SMITH      1000                       20
3    DAVIS      2000       10              10
4    CUTLER     500                        10
5    MICHAELS   2000       10              20
```

A. None

B. One

C. Two

D. Three

E. Five

37. How many times will the word “Hi” appear on the screen if you execute the following code?

```
DECLARE
      x     NUMBER := 0;
```

```
BEGIN
     FOR i IN 1..4 LOOP
          WHILE x < 3 LOOP
               DBMS_OUTPUT.PUT_LINE('Hi');
               x := x+1;
          END LOOP;
     END LOOP;
END;
```

A. Three times

B. Four times

C. Eight times

D. Twelve times

E. An infinite number of times

38. Which line of code in the following program will cause an error?

```
   DECLARE
          TYPE emp_table_type IS TABLE OF VARCHAR2(30)
          INDEX BY BINARY_INTEGER;

          emp_table    emp_table_type;
BEGIN
   FOR i IN 1..4 LOOP
          emp_table_type(i) := 'A name';
   END LOOP;

   DBMS_OUTPUT.PUT_LINE(emp_table(4));
END;
```

A. TYPE emp_table_type IS TABLE OF VARCHAR2(30)
INDEX BY BINARY_INTEGER;

B. emp_table emp_table_type;

C. emp_table_type(i) := 'A name';

D. DBMS_OUTPUT.PUT_LINE(emp_table(4));

E. There are no errors in this program

39. Which of the following implicit cursor attributes can be used to determine if a DELETE statement in a PL/SQL program deleted any rows? Choose three.

A. SQL%FOUND

B. SQL%NOTFOUND

C. SQL%DELETED

D. SQL%SUCCESS

E. SQL%ROWCOUNT

40. Which of the following statements regarding the FOR UPDATE clause is false?

A. You can specify the names of the columns that will be updated in the FOR UPDATE clause.

B. The FOR UPDATE clause allows you to use the WHERE CURRENT OF clause in subsequent update statements.

C. The FOR UPDATE clause automatically updates all the records selected by the cursor.

D. The FOR UPDATE clause will lock all the records selected so no other users can update the records selected by the cursor.

41. Which statement below regarding primary keys is false?

A. A column identified as a primary key must contain unique values.

B. A column identified as a primary key cannot contain a NULL value.

C. You can identify a combination of columns as a primary key.

D. You cannot create a table without a primary key.

E. When you identify a column as the primary key, an index is automatically created based on that column.

42. Which of the following statements are Data Manipulation Language commands?

A. INSERT

B. UPDATE

C. GRANT

D. TRUNCATE

E. CREATE

43. What will be returned for employee SMITH if you execute the following SELECT statement?

```
SELECT id, name, DECODE(contract,1,'Yes',0,'No') CONTRACT
  FROM employee;


  EMPLOYEE
  -------------
  ID     NAME          CONTRACT
  ----   ----------  ---------
  1      JONES         1
  2      SMITH         2
  3      DAVIS         0
```

A. 2 SMITH Yes

B. 2 SMITH No

C. 2 SMITH 2

D. 2 SMITH 0

E. 2 SMITH

44. Which of the following statements will add the following row to the employee table?

```
EMPLOYEE
-------------
ID     NAME          SALARY COMMISSION          DEPTNO
----   ---------     ---------    ----------    -------
1      JONES         1000                            10
```

A. INSERT INTO employee

VALUES (1,JONES,1000,10);

B. INSERT INTO employee

VALUES (1,'JONES',1000',10);

C. INSERT INTO employee

VALUES (id=1,name='JONES',salary=1000,commission=null, deptno=10)

D. INSERT INTO employee (id, name, salary, deptno)

VALUES (1,'JONES',1000,10);

E. INSERT INTO employee (name, id, salary, commission, deptno)

VALUES (1,'JONES',1000,null,10);

45. Which of the following statements will not lock a row in the database? Choose two.

A. SELECT ...FROM table;

B. INSERT INTO table VALUES (...);

C. UPDATE table SET column = value;

D. DELETE FROM table;

E. CREATE TABLE (...)

46. Which of the following SQL*Plus commands will ensure that a message is always displayed saying how many rows were selected?

A. SET FEEDBACK ON

B. SET FEEDBACK 0

C. SET FEEDBACK 1

D. SET FEEDBACK TRUE

47. How many SQL commands are stored in the SQL buffer?

A. 0

B. 1

C. 5

D. 10

E. Depends on SQL*Plus environment variable.

48. Consider the following database table and SELECT statement:

```
EMPLOYEE
----------
ID     NAME          SAL
----   ---------     ---------
1      JONES         5321.6
2      SMITH           1000
3      DAVIS            200
4      CUTLER    .50
```

```
SELECT id, name, sal
  FROM employee;
```

Which of the following SQL*Plus commands will format the salary column so the output appears as follows?

```
ID     NAME          SALARY
----   ---------     ---------
1      JONES         $5,321.60
2      SMITH         $1,000.00
3      DAVIS           $200.00
4      CUTLER     $0.50
```

A. COLUMN salary FORMAT $9,990.00

B. COLUMN sal FORMAT $9,990.00

C. COLUMN sal HEADING salary FORMAT $9,990.00

D. COLUMN sal HEADING salary FORMAT $0,000.00

49. Which of the following commands will execute the script myscript.sql? Choose two.

A. START myscript

B. /

C. RUN myscript.sql

D. @myscript.sql

E. EXECUTE myscript.sql

50. What message(s) will be displayed on the screen if the following program selects a record with a salary that is less than zero?

```
  DECLARE
         e_neg_salary  EXCEPTION;
         v_sal  emp.sal%TYPE;
BEGIN
         SELECT sal
           INTO v_sal
  FROM emp
 WHERE id = 1;

IF v_sal < 0 THEN
                RAISE_APPLICATION_ERROR(
-20001,'Negative salary');
END IF;

DBMS_OUTPUT.PUT_LINE('Program completed');
EXCEPTION
WHEN e_neg_salary THEN
                DBMS_OUTPUT.PUT_LINE
('Employee has negative salary');
END;
```

A. Negative salary

B. Program completed

C. Employee has negative salary

D. Negative salary

Employee has negative salary

E. Negative salary

Program completed

51. If the following code is executed, what is printed on the screen?

```
DECLARE
  v_name        VARCHAR2(30) := 'Smith';
BEGIN
  v_name := 'Jones';

  DECLARE
     v_name        VARCHAR2(30) := 'Davis';
  BEGIN
     v_name := 'Morris';
  END;

  DBMS_OUTPUT.PUT_LINE('v_name is '||v_name);
END;
```

A. v_name is Smith

B. v_name is Jones

C. v_name is Davis

D. v_name is Morris

52. Which of the following commands will cause any pending transactions to be committed. Choose all that apply.

A. COMMIT

B. EXIT

C. SAVE

D. GRANT CREATE TABLE TO scott

E. CREATE VIEW my_emp_view AS SELECT * FROM emp;

53. If you create the following view on the employee table, which of the following commands will execute successfully?

```
DESC employee
Name                  Null?    Type
--------------------- -------- ----
ID                    NOT NULL NUMBER(10)
NAME                  NOT NULL VARCHAR2(20)
SALARY                NOT NULL NUMBER(9,2)
DEPARTMENT            NOT NULL NUMBER(4)

CREATE VIEW emp_view AS
SELECT id emp_id, name emp_name, department
  FROM employee
 WHERE department = 10 WITH CHECK OPTION;
```

A. SELECT id, name, department

FROM emp_view;

B. INSERT INTO emp_view

VALUES (99,'Smith',10);

C. UPDATE emp_view

SET department = 20

WHERE emp_id = 1;

D. SELECT emp_id, emp_name, department

FROM employee;

E. DELETE FROM emp_view;

54. Which condition is an argument for creating an index on a table?

A. The table contains a small amount of data.

B. The SELECT statement will return more than 20 percent of the rows in the table.

C. You are doing a lot of inserts, updates, and deletions on the table.

D. The column contains many NULL values.

E. The column is not used in the WHERE clause of a SELECT statement.

55. Which of the following commands will insert a row and use the default value for the salary column of the employee table?

```
DESC employee
Name                  Null?    Type             Default
--------------------- -------- ---------------- ---------
ID                    NOT NULL NUMBER(10)
NAME                  NOT NULL VARCHAR2(20)
SALARY                         NUMBER(9,2)      1000
DEPARTMENT                     NUMBER(4)
```

A. INSERT INTO employee

VALUES (1,'Smith',10);

B. INSERT INTO employee (id, name, department)

VALUES (1,'Smith',10);

C. INSERT INTO employee

VALUES (1,'Smith',null, 10);

D. INSERT INTO employee (id, name, salary, department)

VALUES (1,'Smith',null, 10);

56. Which of the following PL/SQL programs will fail? Choose two.

```
EMPLOYEE
-------------
ID     NAME
----   ----------
1      JONES
2      SMITH
3      DAVIS
```

A. BEGIN

CREATE TABLE persons (id NUMBER(4), name VARCHAR2(30));

END;

B. BEGIN
COMMIT;
END;

C. DECLARE
v_id employee.id%TYPE;
v_name employee.name%TYPE;
BEGIN
SELECT id, name
INTO v_id, v_name
FROM employee
WHERE id = 1;
END;

D. BEGIN
INSERT INTO employee (id, name)
VALUES(5,'Morris');
END;

E. BEGIN
GRANT SELECT ON employee TO public;
END;

57. Which of the following SELECT statements will execute successfully in your PL/SQL program?

```
DECLARE
       v_id          employee.id%TYPE;
       v_name employee.name%TYPE;
       v_dept employee.dept%TYPE;
BEGIN
       select_statement;
END;
```

A. SELECT id, name, dept
FROM employee
WHERE id =1;

B. SELECT id, name, dept
INTO v_id, v_name, v_dept
FROM employee;

C. SELECT id, name, dept
INTO v_id, v_name, v_dept
FROM employee
WHERE id = 1;

D. SELECT id, name, dept
FROM employee
INTO v_id, v_name, v_dept
WHERE id = 1;

Answers

1. **D —** Because the salary stored in the employee table is a monthly salary, you must multiply the values in the salary column by 12. To get the total compensation, you need the annual salary (salary*12), added to the raise (salary*12*.2), added to the bonus (200). For more information, see the section on arithmetic operations in Chapter 2, "Retrieving Data Using Basic SQL Statements."

2. **E —** You use the ALTER TABLE command to change the structure of a table. You cannot make a column NOT NULL if there is data in the table and that column already contains NULL values. Since all the existing records in the table have values specified for email address, use the MODIFY option to make the column NOT NULL. You can also use the MODIFY option to change the datatype of a column, or add a default value to a columnFor more information on the ALTER TABLE command, see the section on creating and managing tables in Chapter 7, "Creating and Managing Oracle Database Objects."

3. **A, B,** and **D** will be saved to the database. The ROLLBACK to SAVEPOINT B rolls back employee "Smith." The COMMIT after 'Jones' saves employee "Jones" and departments "Admin" and "Finance." The ROLLBACK after 'Billing' rolls back the "Billing" department. The final commit has no effect. For more information on COMMIT and ROLLBACK, see the section on transaction controls in Chapter 5, "Adding, Updating, and Deleting Data."

4. **A —** The name in the database is stored in mixed case, but the UPPER function converts the name to uppercase before comparing the name to the string "SMITH", which is entirely in uppercase. The name must also be enclosed in single quotes because it is a character string. For more information on the UPPER function, see the section on single-row functions in Chapter 3, "Using Single- and Multi-Row Functions."

5. **D** and **E** — D is invalid because you cannot declare a variable as NOT NULL without providing an initial value. E is invalid because when using the %TYPE option, you specify the table name first and the column name second. B is in fact valid because although using column names as variable names is not recommended, it is valid. For more information on declaring PL/SQL variables, see the section on variables in Chapter 9, "Introduction to PL/SQL."

6. **C** — The initial value of v_sold is TRUE; because no initial value is specified for v_paid, it will be NULL. With an AND condition, when one condition is TRUE and the other is NULL, the expression returns NULL. For more information on IF statements, see the section on conditional processing in Chapter 10, "Controlling Program Execution in PL/SQL."

7. **B** — When searching for a NULL value in a column, you must use the keyword IS. No quotes are required around the keyword NULL. For more information, see the section on the WHERE clause in Chapter 2, "Retrieving Data Using Basic SQL Statements."

8. **C** — The query will execute successfully and list all the names and salaries of all employee with the same salary as anyone named "Smith." "Smith" will be included in the list of employee names, and the query will execute successfully regardless of how many rows are returned by the subquery because the IN operator will accept one or more values. For more information, see the section on subqueries in Chapter 4, "Advanced SELECT Statements."

9. **A** — Use the ACCEPT command to create a variable in SQL*Plus and prompt the user for a value. The DEFINE command will create a variable but will not prompt the user for a value. The clause for specifying the message prompt is PROMPT. For more information, see the section on substitution variables in Chapter 6, "The SQL*Plus Environment."

10. **E** — The cursor FOR loop will exit automatically when all rows in the cursor have been processed. The EXIT statement will also cause you to exit the loop when five rows have been processed. %ROWCOUNT increments each time a row is fetched from the cursor. For more information, see the section on explicit cursor attributes in Chapter 11, "Interacting with the Database Using PL/SQL."

11. **A** and **D** — If the SELECT statement returns no rows, the exception NO_DATA_FOUND will be raised and trapped in the sub-block, which displays the message "No data found". After the sub-block handles the error, the code continues to execute where the sub-block ends, and the message "No errors" will be displayed just before the program ends. For more information, see the section on nested blocks in Chapter 12, "Handling Errors and Exceptions in PL/SQL."

12. **E** — The SELECT privilege must be granted to Dave on the table scott.employee. Creating a synonym does not give a user privileges to read from a table. For more information, see the section on object privileges in Chapter 8, "Configuring Security in Oracle Databases."

13. D — Because you have a group function and a column in the SELECT statement, you must have a GROUP BY clause. When you want to restrict the values displayed and your condition involves a group function, you must use the HAVING clause. For more information, see the section on the HAVING clause in Chapter 3, "Using Single- and Multi-Row Functions."

14. C — In order to list departments that have no employees, you must use an outer join. The plus sign (+) goes on the side of the WHERE clause that references the table that has no records to join to — in this case, the employee table. You cannot specify two outer joins on either side of the WHERE clause. For more information, see the section on outer joins in Chapter 4, "Advanced SELECT Statements."

15. B — You should have a comma after each column declaration. If you have a column-level constraint, the comma comes after the column-level constraint. If you have a table-level constraint, you put a comma after the last column and then list the table-level constraints. When using table-level constraints, you must specify in parentheses the name of the column upon which the constraint is being placed. For more information, see the section on creating tables in Chapter 7, "Creating and Managing Oracle Database Objects."

16. D — You must declare a record structure and then the actual record based on the record structure. The structure of the record must have fields that match in order the datatypes and sizes of the columns selected. You cannot use %ROWTYPE in this example because %ROWTYPE creates a record that matches the structure of the entire table, and this table has additional columns that are not selected. For more information, see the section on records in Chapter 11, "Interacting with the Database Using PL/SQL."

17. B — All rows in the employee table will be deleted because no WHERE clause is specified. The statement can be rolled back and is not committed automatically. The DELETE statement deletes rows from a table without dropping the actual table. For more information, see the section on the DELETE statement in Chapter 5, "Adding, Updating, and Deleting Data."

18. A — "NOTFOUND is true" will be displayed because the UPDATE statement will not automatically raise an exception when no rows are updated, but the code checks to see if SQL%NOTFOUND is true after the update is executed. SQL%NOTFOUND will be true if no rows were updated so the RAISE_APPLICATION_ERROR will display the message "NOTFOUND is true". For more information, see the section on user-defined exceptions in Chapter 12, "Handling Errors and Exceptions in PL/SQL."

19. D — You must specify DESC to change the default sort order from ascending to descending for each column you want sorted in descending order. You must list the columns in the order in which you want them sorted. You can use column aliases in the ORDER BY clause. For more information, see the section on the ORDER BY clause in Chapter 2, "Retrieving Data Using Basic SQL Statements."

20. B — 10. The SELECT orderid.NEXTVAL statement will return the number 10, the first value handed out by the sequence. The SELECT orderid_seq.CURRVAL will return the last number handed out by the sequence that was 10. For more information, see the section on sequences in Chapter 7, "Creating and Managing Oracle Database Objects."

21. A — The variable v_counter is equal to one when the program starts, so you never enter the WHILE loop because the WHILE condition is FALSE the first time you reach the loop. For more information, see the section on loops in Chapter 10, "Controlling Program Execution in PL/SQL."

22. C — To return the current date, you select SYSDATE from the table dual. To format the date, you must use the TO_CHAR function. For more information, see the section on single-row functions in Chapter 3, "Using Single- and Multi-Row Functions."

23. D — In Oracle 8*i*, a VARCHAR2 column can hold up to 4,000 characters. For more information, see the section on datatypes in Chapter 7, "Creating and Managing Oracle Database Objects."

24. A — The USER_TABLES data dictionary view lists all the tables you own. The ALL_TABLES view lists all tables you have access to, and the DBA_TABLES view lists all the tables in the database. USER_VIEWS and ALL_VIEWS list information about views. For more information, see the section on data dictionary views in Chapter 7, "Creating and Managing Oracle Database Objects."

25. D and **E** — The AS keyword is optional when specifying a column alias. You must enclose the column alias in double quotes when the alias contains a space or lowercase letters. If you specify an alias in lowercase letters without double quotes, the alias will appear in uppercase. For more information, see the section on aliases in Chapter 2, "Retrieving Data Using Basic SQL Statements."

26. D — CLOB can hold up to 4 gigabytes of character data. You cannot use CHAR or VARCHAR because they can only hold up to 4,000 characters. BFILE and BLOB are for binary, not character, data. For more information, see the section on datatypes in Chapter 7, "Creating and Managing Oracle Database Objects."

27. B — The ALTER USER command with the IDENTIFIED BY option is used to change a user's password. For more information, see the section on changing passwords in Chapter 8, "Configuring Security in Oracle Databases."

28. A and **C** — Not all PL/SQL programs require a DECLARE or EXCEPTION section, so these keywords are not required in a PL/SQL program. For more information, see the section on block structure in Chapter 9, "Introduction to PL/SQL."

29. A, B, C, and **D** — The cursor FOR loop will open the cursor, declare the emp_record record, fetch a row from the cursor each time through the loop, exit when all rows are fetched, and close the cursor after exiting the loop. For more information, see the section on cursor FOR loops in Chapter 11, "Interacting with the Database Using PL/SQL."

30. **A, C,** and **D** are SQL*Plus commands. DESCRIBE, SPOOL, and RUN are all SQL*Plus commands; SELECT and UPDATE are SQL commands. For more information, see the section on SQL*Plus commands in Chapter 6, "The SQL*Plus Environment."

31. **D** — Four rows will be returned. The AND statement will be executed before the OR statement; therefore, all the employees with deptno=20 and a salary greater than 1000 will be returned as well as all employees with deptno=10.

If you wanted to return a list of all employees with a salary greater than 1000 who work in departments 10 or 20, you need to use parentheses to ensure the OR statement is evaluated before the AND statement.

```
SELECT id, name
  FROM emp
 WHERE (deptno = 10
    OR deptno = 20)
   AND salary > 1000;
```

For more information, see the section on the WHERE clause in Chapter 2, "Retrieving Data Using Basic SQL Statements."

32. **E** — This SELECT statement will return an error because, when you use the NVL function, the value passed to the NVL function must be the same datatype as the column passed to the NVL function. In this case, salary is a NUMBER column, and "No salary" is a character string. If you want to display the string "No salary" when the salary value is NULL, you must use the TO_CHAR function to convert the salary a character string.

```
SELECT id, name, NVL(TO_CHAR(salary),'No Salary') "Salary"
  FROM emp;
```

For more information, see the section on NULLS in Chapter 2, "Retrieving Data Using Basic SQL Statements."

33. **D** — This SELECT statement will add three months to the hire date of "01-DEC-2001", which will return "01-MAR-2002", then format the date to appear as "01/03/2002". For more information, see the sections on date and character functions in Chapter 3, "Using Single- and Multi-Row Functions."

34. **B** — The SELECT statement will return 1,500 because the AVG function will return an average and will ignore any NULL values. For more information, see the section on group and multi-row functions in Chapter 3, "Using Single- and Multi-Row Functions."

35. **D** — You must use an outer join to list the departments with no employees. Because it is the employee table that has no records to join with, you must put the plus sign (+)on the side of the join that references the employee table. The JOIN statement itself should compare the dept_id column in the employee table to the id column in the department table. For more information, see the section on outer joins in Chapter 4, "Advanced SELECT Statements."

36. A — No rows will be returned by the query because the subquery will return a commission of NULL. The main query will return no rows because you cannot use an equal sign (=) to check for NULL values in a WHERE clause. For more information, see the section on NULL behavior in Chapter 4, "Advanced SELECT Statements."

37. A — The word "Hi" will appear on the screen three times. You enter the FOR loop, and i equals one. When you first enter the WHILE loop, x equals zero, the word "Hi" is printed, and x is incremented to one. The WHILE loop re-executes, prints "Hi", and increments x to two. The WHILE loop re-executes, prints "Hi", and increments x to three. At this point, you exit the WHILE loop because x is no longer less than three. The FOR loop executes a second time, but you never re-enter the WHILE loop because x is not less than three. For more information, see the section on loops in Chapter 10, "Controlling Program Execution in PL/SQL."

38. C — This line of code will cause an error because the table name (emp_table), not the table type name (emp_table_type), should be used when trying to populate a PL/SQL table. For more information, see the section on PL/SQL tables in Chapter 11, "Interacting with the Database Using PL/SQL."

39. A, B, and **E** — SQL%FOUND will be TRUE if any rows were deleted, SQL%NOTFOUND will be false if any rows were deleted, SQL%ROWCOUNT will be greater than zero if any rows were deleted. For more information, see the section on implicit cursor attributes in Chapter 11, "Interacting with the Database Using PL/SQL."

40. C — The FOR UPDATE clause will not automatically update all the records selected by the cursor. For more information, see the section on the FOR UPDATE clause in Chapter 11, "Interacting with the Database Using PL/SQL."

41. D — Although identifying a primary key on every table you create is recommended, you can create and populate a table without identifying a primary key. For more information, see the section on constraints in Chapter 1, "The Oracle Database," or Chapter 7, "Creating and Managing Oracle Database Objects."

42. A and **B** — The INSERT and UPDATE statements are Data Manipulation Language (DML) commands. GRANT is a Data Control Language (DCL) command. TRUNCATE and CREATE are Data Definition Language (DDL) commands. For more information, see Chapter 2, "Retrieving Data Using Basic SQL Statements."

43. E — Employee SMITH has a contract value of two. Because that value does not match any of the choices listed in the DECODE function, and no default for other values is specified in the DECODE function, a default value of NULL is used. For more information, see the section on DECODE in Chapter 3, "Using Single- and Multi-Row Functions."

44. D — The name column is a character value and must be specified in single quotes. If no column names are specified, then values for all columns must be specified in the order in which they appear in the table. If column names are specified after the table name, values for each of the column names must be provided in the order in which they appear, any columns not listed will have a NULL value. For more information, see the section on inserts in Chapter 5, "Adding, Updating, and Deleting Data."

45. A and **E** — The SELECT statement will not lock a row in the database unless you specify the FOR UPDATE clause. The CREATE TABLE command does not lock any rows. For more information, see the section on locks in Chapter 5, "Adding, Updating, and Deleting Data."

46. C — Setting the FEEDBACK to 1 ensures that if one or more rows are selected, you see a message telling you how many records are displayed. For more information, see the section on customizing the SQL*Plus environment in Chapter 6, "The SQL*Plus Environment."

47. B — Only one SQL command is stored in the SQL buffer. For more information, see the section on the SQL buffer in Chapter 6, "The SQL*Plus Environment."

48. C — The COLUMN command should reference the column name "sal". The HEADING option should specify the new column title "salary". The FORMAT option should use nines in the format mask to suppress leading zeros. For more information, see the section on formatting in Chapter 6, "The SQL*Plus Environment."

49. A and **D** — The START command and the at sign (@) are used to execute scripts. The file extension will be defaulted to .sql if it is not specified. For more information, see the section on scripts in Chapter 6, "The SQL*Plus Environment."

50. A — "Negative salary" will be displayed on the screen. When negative salary is selected, the IF statement will be true, and the RAISE_APPLICATION_ERROR statement will be executed. RAISE_APPLICATION_ERROR will cause the program to jump to exception handling, and because the error raised is not handled in exception handling, the error is passed back to SQL*Plus, and the message passed to RAISE_APPLICATION_ERROR is displayed on the screen. For more information, see the section on RAISE_APPLICATION_ERROR in Chapter 12, "Handling Errors and Exceptions in PL/SQL."

51. B — v_name will be equal to "Jones" when the code is executed. The subblock modifies a local copy of a variable called "v_name" and not the variable v_name owned by the parent block. For more information, see the section on nested blocks in Chapter 10, "Controlling Program Execution in PL/SQL."

52. A, B, D, and **E** — A COMMIT statement will be issued to the database if you type the command COMMIT, if you exit normally with the EXIT command, if you issue a Data Control Language (DCL) statement such as GRANT, or if you issue a Data Definition Language (DDL) command such as CREATE. The SAVE command does not exist. For more information, see the section on transaction controls in Chapter 5, "Adding, Updating, and Deleting Data."

53. E — The DELETE statement will execute successfully. The first SELECT statement will fail because the column names in the view are emp_id and emp_name not id and name. The INSERT statement will fail because no value is provided for the salary column in the employee table, and the salary column cannot be NULL. The UPDATE statement will fail because the CHECK OPTION was specified, which prevents you from updating a row in a view to a value that can no longer be seen though the view, and the view only shows records with a department id of 10. The last SELECT statement will fail because you are using the column names from the view but you are selecting from the table. For more information, see the section on views in Chapter 7, "Creating and Managing Oracle Database Objects."

54. D — If the column contains many NULL values, then an index may speed up queries using that column. For more information, see the section on indexes in Chapter 7, "Creating and Managing Oracle Database Objects."

55. B — To use the default value for a column when inserting a row, you must not specify any value for that column. If you are not going to specify for all the columns in the table, you must list the columns you are populating after the table name. For more information, see the section on DEFAULT values in Chapter 7, "Creating and Managing Oracle Database Objects," and the section on the INSERT statement in Chapter 5, "Adding, Updating, and Deleting Data."

56. A and **E** — Data Definition Language (DDL) commands such as CREATE TABLE and Data Control Language (DCL) commands such as GRANT cannot be issued directly from PL/SQL programs. For more information, see Chapter 11, "Interacting with the Database Using PL/SQL."

57. C — You must specify an INTO clause after the SELECT clause that lists variables to hold the values returned. The SELECT statement should contain a WHERE clause because if a SELECT statement returns more than one row, you will get an exception. For more information, see the section on SELECT statements in Chapter 11, "Interacting with the Database Using PL/SQL."

APPENDIX C

Objective Mapping

The Oracle 8*i* "Introduction to Oracle: SQL and PL/SQL" exam has a clear set of defined objectives. This book covers all of the material that is required for the exam, with a little extra, such as the coverage of stored procedures, triggers, and other PL/SQL program units in Chapter 13, "Introduction to Stored Programs." Generally, the knowledge you gain by reading this book and working through the exercises should place you in good stead when it comes time to take the exam. This book can also serve as a reference as you continue to work with Oracle, since the use of functions, PL/SQL, advanced SQL syntax, and the like will always be needed.

Table C-1 can assist you in ensuring that you have properly prepared for the exam by outlining which sections of the book cover which parts of the exam. Use this table as a guideline to ensure that you have covered all the bases. It can also be used to help you determine which parts of the book to review in case you may have to take the exam again.

Table C-1
Objective Mapping

Exam Objective	*Chapter Covering Objective*	*Section Covering Objective*
Overview of Relational Databases, SQL, and PL/SQL	Chapter 1: "The Oracle Database"	
✦ Discuss the theoretical and physical aspects of a relational database	Chapter 1: "The Oracle Database"	"Overview"
✦ Describe the Oracle implementation of RDBMS and ORDBMS	Chapter 1: "The Oracle Database"	"Overview"
✦ Describe the use and benefits of PL/SQL	Chapter 9: "Introduction to PL/SQL"	"Uses and Benefits of PL/SQL"
Write Basic SQL Statements	Chapter 2: "Retrieving Data Using Basic SQL Statements"	
✦ List the capabilities of SQL SELECT statements	Chapter 2: "Retrieving Data Using Basic SQL Statements"	"Basic SELECT Statement"
✦ Execute a basic SELECT statement	Chapter 2: "Retrieving Data Using Basic SQL Statements"	"Basic SELECT Statement"
✦ Differentiate between SQL statements and SQL*Plus commands	Chapter 6: "The SQL*Plus Environment"	"SQL*Plus Commands"
Restrict and Sort Data	Chapter 2: "Retrieving Data Using Basic SQL Statements"	
✦ Limit the rows retrieved by a query	Chapter 2: "Retrieving Data Using Basic SQL Statements"	"WHERE Clause"
✦ Sort the rows retrieved by a query	Chapter 2: "Retrieving Data Using Basic SQL Statements"	"Ordering Data in the SELECT Statement"
Single-Row Functions	Chapter 3: "Using Single- and Multi-Row Functions"	
✦ Describe various types of functions available in SQL	Chapter 3: "Using Single- and Multi-Row Functions"	
✦ Use character, number, and date functions in SELECT statements	Chapter 3: "Using Single- and Multi-Row Functions"	"Single-Row Functions"
✦ Describe the use of conversion functions	Chapter 3: "Using Single- and Multi-Row Functions"	"Single-Row Functions"

Exam Objective	*Chapter Covering Objective*	*Section Covering Objective*
Displaying Data from Multiple tables	Chapter 4: "Advanced SELECT Statements"	
✦ Write SELECT statements to access data from more than one table using equality and nonequality joins	Chapter 4: "Advanced SELECT Statements"	"Joins"
✦ View data that generally does not meet a join condition by using outer joins	Chapter 4: "Advanced SELECT Statements"	"Joins"
✦ Join a table to itself	Chapter 4: "Advanced SELECT Statements"	"Joins"
Aggregating Data Using Group Functions	Chapter 3: "Using Single- and Multi-Row Functions"	
✦ Identify the available group functions	Chapter 3: "Using Single- and Multi-Row Functions"	"Group/Aggregate Functions"
✦ Describe the use of group functions	Chapter 3: "Using Single- and Multi-Row Functions"	"Group/Aggregate Functions"
✦ Group data using the GROUP BY clause	Chapter 3: "Using Single- and Multi-Row Functions"	"GROUP BY Clause"
✦ Include or exclude grouped rows by using the HAVING clause	Chapter 3: "Using Single- and Multi-Row Functions"	"GROUP BY Clause"
Subqueries	Chapter 4: "Advanced SELECT Statements"	"Subqueries"
✦ Describe the types of problems that subqueries can solve	Chapter 4: "Advanced SELECT Statements"	"Subqueries"
✦ Define subqueries	Chapter 4: "Advanced SELECT Statements"	"Subqueries"
✦ List the types of subqueries	Chapter 4: "Advanced SELECT Statements"	"Subqueries"
✦ Write single-row and multiple-row subqueries	Chapter 4: "Advanced SELECT Statements"	"Subqueries"

Continued

Table C-1 *(continued)*

Exam Objective	*Chapter Covering Objective*	*Section Covering Objective*
Multiple-Column Subqueries	Chapter 4: "Advanced SELECT Statements"	"Subqueries"
✦ Write multiple-column subqueries	Chapter 4: "Advanced SELECT Statements"	"Subqueries"
✦ Describe and explain the behavior of subqueries when null values are retrieved	Chapter 4: "Advanced SELECT Statements"	"Subqueries"
✦ Write subqueries in a FROM clause	Chapter 4: "Advanced SELECT Statements"	"Subqueries"
Producing Readable Output with SQL*Plus	Chapter 6: "The SQL*Plus Environment"	
✦ Produce queries that require an input variable	Chapter 4: "Advanced SELECT Statements"	"Using Substitution Variables"
	Chapter 6: "The SQL*Plus Environment"	"Defining Variables"
✦ Customize the SQL*Plus environment	Chapter 6: "The SQL*Plus Environment"	"Customizing SQL* Plus with SET Commands"
✦ Produce more readable output	Chapter 6: "The SQL*Plus Environment"	"Formatting Output with SQL*Plus"
✦ Create and execute script files	Chapter 6: "The SQL*Plus Environment"	"Scripts"
✦ Save customizations	Chapter 6: "The SQL*Plus Environment"	"Customizing SQL*Plus with SET Commands"
	Chapter 6: "The SQL*Plus Environment"	"PRODUCT_USER_ PROFILE"
Manipulating Data	Chapter 5: "Adding, Updating, and Deleting Data"	
✦ Describe each DML statement	Chapter 5: "Adding, Updating, and Deleting Data"	"DML Statements"
✦ Insert rows into a table	Chapter 5: "Adding, Updating and Deleting Data"	"DML Statements: Inserting Data into Tables"
✦ Update rows in a table	Chapter 5: "Adding, Updating, and Deleting Data"	"DML Statements Modifying Existing Data"

Exam Objective	*Chapter Covering Objective*	*Section Covering Objective*
✦ Delete rows from a table	Chapter 5: "Adding, Updating, and Deleting Data"	"DML Statements: Removing Data from Tables"
✦ Control transactions	Chapter 5: "Adding, Updating, and Deleting Data"	"Controlling Transactions"
Creating and Managing Tables	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating and Managing Tables"
✦ Describe the main database objects	Chapter 1: "The Oracle Database"	"Database Objects"
✦ Create tables	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating and Managing Tables"
✦ Describe the datatypes that can be used when specifying column definitions	Chapter 1: "The Oracle Database"	"Database Objects"
✦ Alter table definitions	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating and Managing Tables"
✦ Drop, rename, and truncate tables	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating and Managing Tables"
Including Constraints	Chapter 7: "Creating and Managing Oracle Database Objects"	"Data Integrity Using Constraints"
✦ Describe constraints	Chapter 1: "The Oracle Database"	"Database Objects"
	Chapter 7: "Creating and Managing Oracle Database Objects"	"Data Integrity Using Constraints"
✦ Create and maintain constraints	Chapter 7: "Creating and Managing Oracle Database Objects"	"Data Integrity Using Constraints"

Continued

Table C-1 *(continued)*

Exam Objective	*Chapter Covering Objective*	*Section Covering Objective*
Creating Views	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating Other Database Objects," "Views"
✦ Describe a view	Chapter 1: "The Oracle Database"	"Database Objects"
	Chapter 7: Creating and Managing Oracle Database Objects"	"Creating Other Database Objects," "Views"
✦ Create a view	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating Other Database Objects," "Views"
✦ Retrieve data through a view	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating Other Database Objects," "Views"
✦ Insert, update, and delete data through a view	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating Other Database Objects," "Views"
	Chapter 5: "Adding, Updating and Deleting Data"	
✦ Drop a view	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating Other Database Objects," "Views"
Oracle Data Dictionary	Chapter 1: "The Oracle Database"	"The Oracle Data Dictionary"
	Appendix F: "Data Dictionary Views"	
✦ Describe the data dictionary views a user may access	Chapter 1: "The Oracle Database"	"The Oracle Data Dictionary"
	Appendix F: "Data Dictionary Views"	
	This information is also presented in each appropriate chapter. For example, when discussing creating and maintaining tables, you will be shown the views that will tell you how to get information on tables.	

Exam Objective	*Chapter Covering Objective*	*Section Covering Objective*
✦ Query data from the data dictionary	Chapter 1: "The Oracle Database"	"Oracle's Data Dictionary"
	This information is also presented in each appropriate chapter. For example, when discussing creating and maintaining tables, you will be shown the views that will tell you how to get information on tables.	
Oracle Database Objects	Chapter 1: "The Oracle Database"	"Database Objects"
	Chapter 7: "Creating and Managing Oracle Database Objects"	
✦ Describe database objects and their uses	Chapter 1: "The Oracle Database"	"Database Objects"
	Chapter 7: "Creating and Managing Oracle Database Objects"	
✦ Create, maintain, and use sequences	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating Other Database Objects," "Sequences"
✦ Create and maintain indexes	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating Other Database Objects," "Indexes"
✦ Create private and public synonyms	Chapter 7: "Creating and Managing Oracle Database Objects"	"Creating Other Database Objects," "Synonyms"
Controling User Access	Chapter 8: "Configuring Security in Oracle Databases"	
✦ Create users	Chapter 8: "Configuring Security in Oracle Databases"	"Users and Schemas"
✦ Create roles to ease setup and maintenance of the security model	Chapter 8: "Configuring Security in Oracle Databases"	"Roles"
✦ Use the GRANT and REVOKE statements to grant and revoke object privileges	Chapter 8: "Configuring Security in Oracle Databases"	"Object Privileges"

Continued

Table C-1 *(continued)*

Exam Objective	*Chapter Covering Objective*	*Section Covering Objective*
Declaring Variables	Chapter 9: "Introduction to PL/SQL"	"Variables"
	Chapter 4: "Advanced SELECT Statements"	"Runtime Variables"
✦ List the benefits of PL/SQL	Chapter 9: "Introduction to PL/SQL"	"Uses and Benefits of PL/SQL"
✦ Describe the basic PL/SQL block and its structure	Chapter 9: "Introduction to PL/SQL"	"Block Structure"
✦ Describe the significance of variables in PL/SQL	Chapter 9: "Introduction to PL/SQL"	"Variables"
✦ Declare PL/SQL variables	Chapter 9: "Introduction to PL/SQL"	"Variables"
✦ Execute a PL/SQL block	Chapter 9: "Introduction to PL/SQL"	"Block Structure"
Writing Executable Statements	Chapter 9: "Introduction to PL/SQL"	"Block Structure"
✦ Describe the significance of the executable section	Chapter 9: "Introduction to PL/SQL"	"Block Structure"
✦ Write statements in the executable section	Chapter 9: "Introduction to PL/SQL"	"Block Structure"
✦ Describe the rules of nested blocks	Chapter 10: "Controlling Program Execution in PL/SQL"	"Nesting Blocks"
✦ Execute and test a PL/SQL block	Chapter 9: "Introduction to PL/SQL"	
	Chapter 10: "Controlling Program Execution in PL/SQL"	
	Chapter 11: "Interacting with the Database Using PL/SQL"	
	Chapter 12: "Handling Errors and Exceptions in PL/SQL"	
✦ Use coding conventions	Chapter 9: "Introduction to PL/SQL"	
	Chapter 12: "Handling Errors and Exceptions in PL/SQL"	"Coding Standards"

Exam Objective	*Chapter Covering Objective*	*Section Covering Objective*
Interacting with the Oracle Server	Chapter 11: "Interacting with the Database Using PL/SQL"	
✦ Write a successful SELECT statement in PL/SQL	Chapter 11: "Interacting with the Database Using PL/SQL"	
✦ Declare the datatype and size of a PL/SQL variable dynamically	Chapter 11: "Interacting with the Database Using PL/SQL"	"Composite Datatypes"
	Chapter 11: "Interacting with the Database Using PL/SQL"	"Cursor Variables"
	Chapter 9: "Introduction to PL/SQL"	"Variables"
✦ Write DML statements in PL/SQL	Chapter 11: "Interacting with the Database Using PL/SQL"	"DML"
✦ Control transactions in PL/SQL	Chapter 10: "Controlling Program Execution in PL/SQL"	"Transaction Control"
✦ Determine the outcome of SQL DML statements	Chapter 11: "Interacting with the Database Using PL/SQL"	"DML"
Writing Control Structures	Chapter 10: "Controlling Program Execution in PL/SQL"	
✦ Identify the use and type of control structures	Chapter 10: "Controlling Program Execution in PL/SQL"	
✦ Construct an IF statement	Chapter 10: "Controlling Program Execution in PL/SQL"	"Conditional Processing"
✦ Construct and identify different loop statements	Chapter 10: "Controlling Program Execution in PL/SQL"	"Loops"
✦ Use logic tables	Chapter 10: "Controlling Program Execution in PL/SQL"	"Conditional Processing"
✦ Control block flow using nested loops and labels	Chapter 10: "Controlling Program Execution in PL/SQL"	"Loops"

Continued

Table C-1 *(continued)*

Exam Objective	*Chapter Covering Objective*	*Section Covering Objective*
Working with Composite Datatypes	Chapter 9: "Introduction to PL/SQL"	"Variables"
	Chapter 11: "Interacting with the Database Using PL/SQL"	"Composite Datatypes"
✦ Create user-defined PL/SQL records	Chapter 11: "Interacting with the Database Using PL/SQL"	"Composite Datatypes," "PL/SQL Records"
✦ Create a record with the %ROWTYPE attribute	Chapter 9: "Introduction to PL/SQL"	"Variables," "%ROWTYPE"
	Chapter 11: "Interacting with the Database Using PL/SQL"	"Composite Datatypes," "PL/SQL Records"
✦ Create a PL/SQL table	Chapter 11: "Interacting with the Database Using PL/SQL"	"Composite Datatypes," "PL/SQL Tables"
✦ Create a PL/SQL table of records	Chapter 11: "Interacting with the Database Using PL/SQL"	"Composite Datatypes," "PL/SQL Table of Records"
✦ Describe the differences between records, tables, and tables of records	Chapter 11: "Interacting with the Database Using PL/SQL"	"Composite Datatypes"
Write Explicit cursors	Chapter 11: "Interacting with the Database Using PL/SQL"	"Cursors"
✦ Distinguish between an implicit and an explicit cursor	Chapter 11: "Interacting with the Database Using PL/SQL"	"Cursors"
✦ Use a PL/SQL record variable	Chapter 11: "Interacting with the Database Using PL/SQL"	"PL/SQL Records"
	Chapter 11: "Interacting with the Database Using PL/SQL"	"Cursors"
✦ Write a cursor FOR loop		"Cursor FOR Loops"

Exam Objective	*Chapter Covering Objective*	*Section Covering Objective*
Advanced Explicit Cursor Concepts	Chapter 11: "Interacting with the Database Using PL/SQL"	
✦ Write a cursor that uses parameters	Chapter 11: "Interacting with the Database Using PL/SQL"	"DML," "Cursor Parameters"
✦ Determine when a FOR UPDATE clause in a cursor is required	Chapter 11: "Interacting with the Database Using PL/SQL"	"DML," "Cursor Parameters"
✦ Determine when to use the WHERE CURRENT OF clause	Chapter 11: "Interacting with the Database Using PL/SQL"	"DML," "Cursor Parameters"
✦ Write a cursor that uses a subquery	Chapter 11: "Interacting with the Database Using PL/SQL"	"Cursors"
✦ Handling exceptions	Chapter 12: "Handling Errors and Exceptions in PL/SQL"	
✦ Define PL/SQL exceptions	Chapter 12: "Handling Errors and Exceptions in PL/SQL"	
✦ Recognize unhandled exceptions	Chapter 12: "Handling Errors and Exceptions in PL/SQL"	"Debugging"
✦ List and use different types of PL/SQL exception handlers	Chapter 12: "Handling Errors and Exceptions in PL/SQL"	"Defining, Trapping, and Handling Errors"
✦ Trap unanticipated errors	Chapter 12: "Handling Errors and Exceptions in PL/SQL"	"Defining, Trapping, and Handling Errors"
✦ Describe the effect of exception propagation in nested blocks	Chapter 12: "Handling Errors and Exceptions in PL/SQL"	"Defining, Trapping, and Handling Errors"
✦ Customize PL/SQL exception messages	Chapter 12: "Handling Errors and Exceptions in PL/SQL"	"Defining, Trapping, and Handling Errors"

✦ ✦ ✦

APPENDIX D

Exam Tips

Before you sit down and take the "Introduction to Oracle: SQL & PL/SQL" exam, it is important to understand what to expect and how to approach the exam with the highest probability of success.

This appendix provides pointers to help you prepare for the exam. This appendix also outlines the processes for registering to take the exam, as well as what happens if you do not succeed on your first attempt.

Preparing to the Exam

While some of the points that are outlined in this section may seem common sense, they are included here to remind you of what others have used to successfully prepare to pass the "Introduction to Oracle: SQL & PL/SQL" exam. In preparing for the exam, follow these steps:

- **Read the material in this book.** This book was designed to act as a study guide to prepare you to take and pass the exam. All of the information that you will be tested on can be found in this book. While there are no guarantees, being completely familiar with the contents of this book will go a long way to help you pass the exam.
- **Get a copy of Oracle.** Appendix A outlines where you can acquire the Oracle software if you do not currently own it. Passing the exam without working with the product is possible, but extremely difficult and quite unlikely. Not only is it important to work with the Oracle software in preparation for the exam, but you'll also need this kind of experience when you are asked to work on a real live database.
- **Do the labs.** The best way to prepare to perform any task well is to do it over and over again. It has been proven that performing a task many times reinforces the concepts associated with the task. The labs in this book are designed to reinforce the lessons presented in each chapter.

- **Hit a few roadblocks.** When doing the labs in this book, or when actually working with the Oracle software, and you hit a roadblock and do not know how to proceed, try to figure it out on your own before turning to the lab answers, the book text, or the Oracle manuals. The way most database administrators (DBAs) learn how to overcome a problem is by running into it at least once, and sometimes a few more times than that and then figuring out the solution.
- **Test yourself.** Each chapter starts with questions to test your knowledge. Each chapter has exam-style assessment questions. Each chapter also has scenarios in which you are required to solve a problem. Review these questions and scenarios and test your knowledge continuously, not just before you take the exam. If you find you are weak in a certain area, review the material and test again.
- **Take the practice exams.** This book has a practice exam in Appendix B, as well as one on the CD-ROM. Self Test software also publishes practice exams for Oracle certification exams. Exposure to many questions will better prepare your test-taking abilities.
- **Understand the material.** The goal is not just to pass the exam but also to understand the material. Passing the exam is easy if you are comfortable with the subject matter of the exam
- **Work with it.** Practice makes perfect, so work with Oracle every chance you get. Ask your friends and colleagues to give you assignments that you can solve with Oracle, SQL, and PL/SQL. The more you do work with these assignments, the easier it gets. This method works for improving your golf swing or understanding Oracle.

Registering for the Exam

Oracle certification exams are offered through Sylvan Prometric testing centers worldwide. Consult the *Oracle Certified Professional Program Candidate Guide* available on Oracle's Web site at `www.oracle.com/education/certification/index.html?ocpguides.html` or the Oracle Certification Web site at `www.oracle.com/education/certification/index.html?content.html`.

Taking the Exam

When your day to take the exam arrives, get to the testing center at least a few minutes early. This gives you some time to relax before going into the test room. You need to bring two types of identification with you (one with a photo and signature and the other with your signature). You will be asked to sign in and then be escorted to the computer you will use to take the test. You will also get a brief orientation to the testing process if you are new to it.

When taking the test, keep these points in mind:

- **Pace yourself.** The exam is timed to allow you about a minute or so to answer each question. Take the time to read each question completely and then select your answers. Do not rush. Finishing the exam in record time and failing is obviously not as good as taking all the time allowed and passing.
- **Read the entire question and all the answers.** This may seem like a common sense thing to do, but it is amazing how many people do not follow this simple rule. Before jumping to answer a question, make sure you have read it completely and have read each of the possible answers.
- **Mark questions.** The exam allows you to mark questions whose answers you are unsure of and come back to them later. Use this feature. Even if you are 90 percent sure of an answer, mark the question and come back to it. Who knows? The answer to the question you marked may be found later in the test or be triggered by something later in the test.
- **Don't get bogged down.** Do not spend too much time trying to answer a single question. You still have many more to answer, so mark the question and move on.
- **Relax.** Stress and panic work against you. Take a deep breath and relax.

After the Test

After you have taken the test, you will know your results right away. When you confirm that you want to end the test, your score and pass or fail grade is presented on the screen and a hard copy is printed for you to take home. If you passed, CONGRATULATIONS!!

If you were not successful, don't despair. While this book is designed to help you pass the exam, it is not guaranteed. Use the information on your test score report to identify the areas where you are weak, and focus on those before retaking the exam. Don't neglect other areas though, because the exam will still test your knowledge of the complete set of objectives outlined in Appendix C.

Oracle has a mandatory waiting period of 30 days before you can retake the exam. Use this time to make yourself even more comfortable with the material.

Good luck!!

✦ ✦ ✦

Database Schema for Labs

Throughout this book, you have been working with a number of Oracle database objects. The labs and the examples within the various chapters of the book call for a number of tables and other objects. This appendix provides information about the structure of the objects being used, as well as a hard copy of the scripts found in the DBSETUP folder on the CD-ROM that you can use to create this same structure in your database.

Table Structures

The tables used in the book are as follows:

Table E-1
Courses Table

Column Name	*Datatype and Length*	*Null?*	*Primary Key?*	*Foreign Key?*	*Table/Column*
CourseNumber	number(5)	No	Yes		
CourseName	varchar2 (200)	No	No		
ReplacesCourse	number(5)	Yes	No		
RetailPrice	number (9,2)	No	No		
Description	varchar2 (2000)	Yes	No		

Table E-2
Instructors Table

Column Name	*Datatype and Length*	*Null?*	*Primary Key?*	*Foreign Key?*	*Table/Column*
InstructorID	number(5)	No	Yes		
Salutation	char (4)	Yes	No		
LastName	varchar2 (30)	No	No		
FirstName	varchar2 (30)	No	No		
MiddleInitial	varchar2 (5)	Yes	No		
Address1	varchar2 (50)	Yes	No		
Address2	varchar2 (50)	Yes	No		
City	varchar2 (30)	Yes	No		
State	char (2)	Yes	No		
Country	varchar2 (30)	Yes	No		
PostalCode	char (10)	Yes	No		
OfficePhone	char (15)	Yes	No		
HomePhome	char (15)	Yes	No		
CellPhone	char (15)	Yes	No		
EMail	varchar2 (50)	Yes	No		
InstructorType	char (10)	No	No		
PerDiemCost	number (9,2)	Yes	No		
PerDiemExpenses	number (9,2)	Yes	No		
Comments	varchar2 (2000)	Yes	No		

Table E-3
Locations Table

Column Name	*Datatype and Length*	*Null?*	*Primary Key?*	*Foreign Key?*	*Table/Column*
LocationID	number(5)	No	Yes		
LocationName	varchar2 (50)	No	No		
Address1	varchar2 (50)	Yes	No		
Address2	varchar2 (50)	Yes	No		
City	varchar2 (30)	Yes	No		
State	char (2)	Yes	No		
Country	varchar2 (30)	Yes	No		
PostalCode	char (10)	Yes	No		
Telephone	char (15)	Yes	No		
Fax	char (15)	Yes	No		
Contact	varchar2 (50)	Yes	No		
Description	varchar2 (2000)	Yes	No		

Table E-4
Students Table

Column Name	*Datatype and Length*	*Null?*	*Primary Key?*	*Foreign Key?*	*Table/Column*
StudentNumber	number(5)	No	Yes		
Salutation	char (4)	Yes	No		
LastName	varchar2 (30)	No	No		
FirstName	varchar2 (30)	No	No		
MiddleInitial	varchar2 (5)	Yes	No		
Address1	varchar2 (50)	Yes	No		
Address2	varchar2 (50)	Yes	No		
City	varchar2 (30)	Yes	No		
State	char (2)	Yes	No		

Continued

Table E-4 *(continued)*

Column Name	*Datatype and Length*	*Null?*	*Primary Key?*	*Foreign Key?*	*Table/Column*
Country	varchar2 (30)	Yes	No		
PostalCode	char (10)	Yes	No		
HomePhone	char (15)	Yes	No		
WorkPhome	char (15)	Yes	No		
EMail	varchar2 (50)	Yes	No		
Comments	varchar2 (2000)	Yes	No		

Table E-5
Scheduled Classes Table

Column Name	*Datatype and Length*	*Null?*	*Primary Key?*	*Foreign Key?*	*Table/ Column*
ClassID	number(5)	No	Yes		
CourseNumber	number(5)	No	No	Courses (CourseNumber)	
LocationID	number(5)	No	No	Locations (LocationID)	
ClassRoomNumber	number(3)	No	No		
InstructorID	number(5)	No	No	Instructors (InstructorID)	
StartDate	date	No	No		
DaysDuration	number(3)	No	No		
Status	char (10)	No	No		
Comments	varchar2 (2000)	Yes	No		

Table E-6
Class Enrollment Table

Column Name	*Datatype and Length*	*Null?*	*Primary Key?*	*Foreign Key?*	*Table/ Column*
ClassID	number(5)	No	Yes	ScheduledClasses (ClassID)	
StudentNumber	number(5)	No	Yes	Students (StudentNumber)	
Status	char (10)	No	No		
EnrollmentDate	date	No	No		
Price	number (9,2)	No	No		
Grade	char (40)	Yes	No		
Comments	varchar2 (2000)	Yes	No		

Table E-7
Batch Jobs Table

Column Name	*Datatype and Length*	*Null?*	*Primary Key?*	*Foreign Key?*	*Table/Column*
JobId	number (6)	No	Yes		
JobName	varchar2 (30)	No	No		
Status	varchar2 (30)	No	No		
LastUpdated	date	No	No		

Table E-8
Course Audit Table

Column Name	*Datatype and Length*	*Null?*	*Primary Key?*	*Foreign Key?*	*Table/ Column*
CourseNumber	number(5)	No	Yes	Courses (CourseNumber)	
Change	varchar2 (30)	No	Yes		
DateChanged	date	No	Yes		
Price	number (9,2)	Yes	No		
ChangedBy	varchar2 (15)	Yes	No		

Scripts Used to Create Database Objects

The CD-ROM accompanying this book has a number of scripts in the DBSETUP folder that can be used to create the tables and load sample data into them. The scripts and their purposes are:

- **CREATEUSER.SQL:** Creates the STUDENT user and the CERTDB tablespace.
- **CERTDBOBJ.SQL:** Creates the tables and adds constraints.
- **INSERT_DATA.SQL:** Adds sample data to the database.

Before Running the scripts

In order to successfully run the scripts, ensure that the following tasks have been completed:

1. You have installed, according to the installation instructions provided by Oracle, Oracle 8*i* version 8.1.6 or later, on the computer that you will use to perform the lab exercises and other steps in the book.

The book assumes you are running Oracle 8.1.6 Enterprise Edition or later because this is what the Oracle exam is based on. Although most labs will work with Oracle 8.1.6 Server or later, it is recommended that you download the latest version of Oracle 8i Enterprise Edition for your platform by joining the Oracle Technology Network at `http://otn.oracle.com` and going to the Download section. Oracle 8i Personal Edition is not recommended or supported.

2. You have created a database to hold the objects and data that the scripts will create, and configured Net8 to be able to connect to the server.

3. You have been granted the DBA role in the database, or you know the password for the SYSTEM user in the database.
4. You have created a directory on your C: drive (for Windows NT/2000/9x) or off your root (for Linux) to hold the Oracle datafile that will be created by the scripts. (See the scripts for the actual folder name.)
5. You are not running these scripts on a production database used for other critical purposes in your organization.

The scripts themselves are simply ASCII files that may be modified. To ensure that they will work properly in your environment, you should verify their contents and make any necessary changes. It is recommended that you copy the script files from the DBSETUP folder of the CD-ROM to a new folder you create called CERTDB on your hard drive (this is the folder where the datafile will be created by default). Before running the scripts, check their contents for the following:

1. The CREATEUSER.SQL script will connect to the default instance as the user SYSTEM with a password of "manager". If this is not correct, or you wish to connect as another user who has been assigned the DBA role, or you wish to connect to a different instance, change the appropriate line in the script.
2. The CREATEUSER.SQL script will first delete and then create a tablespace called CERTDB and place the datafile in a folder called CERTDB on your hard disk. If you want to place the datafile in a different location, change the appropriate lines in the script — the line that deletes the data file, as well as the one that creates the tablespace.
3. The CREATEUSER.SQL script will create an Oracle user called STUDENT with a password of "oracle". If you already have a user called STUDENT created in the database, that user will be dropped by the script before being re-created. Modify the appropriate lines of the script to drop and create a user with a different name.
4. The CERTDBOBJ.SQL script will connect to the database as the user STUDENT by default. If you changed the username in the CREATEUSER.SQL script, modify the appropriate line of CERTDBOBJ.SQL to have the objects created by the user you specified.

Running the scripts

To run the scripts, perform the following tasks:

1. Log on to the computer that you will be running Oracle from as an administrator (preferred).
2. If you are running WindowsNT/2000/9x, create a folder off of the root of your C: drive called **CERTDB**. If you are running Linux or another Unix variant, create a folder off the root called **CERTDB**. In the Linux/Unix world, the folder must be named with all uppercase letters.

3. Insert the CD into your CD-ROM drive and locate the **DBSETUP** folder. Copy the contents of the **DBSETUP** folder to the **CERTDB** folder you created.
4. Invoke Server Manager line mode from the command line and execute the scripts in the following order:

   CREATEUSER.SQL: To create the user and tablespace.

   CERTDBOBJ.SQL: To create the database objects.

   INSERT_DATA.SQL: To load the sample data.

   To execute the scripts, at the command prompt, issue the following commands:

```
C:\CERTDB> svrmgrl

Oracle Server Manager Release 3.1.7.0.0 - Production

Copyright (c) 1997, 1999, Oracle Corporation. All Rights
Reserved.

Oracle8i Enterprise Edition Release 8.1.7.0.0 - Production
With the Partitioning option
JServer Release 8.1.7.0.0 - Production

SVRMGRL> @createuser.sql;
...
SVRMGRL> @certdbobj.sql;
...
SVRMGRL> @insert_data.sql;
...
SVRMGRL> quit
Server Manager Complete

C:\CERTDB>
```

Text of scripts used to create the database objects

This section provides the source code for the preceding scripts. You can use this source code in case the CD-ROM that came with the book is not easily available. You can enter the commands in these scripts into SVRMGRL or SQL*Plus and then perform the same tasks.

This information is also useful if you modified the scripts and want to see what the original looked like. Having a print version of the original may make it easier to undo your changes.

CREATEUSER.SQL

```
# This script creates the STUDENT user and CERTDB tablespace.
#
# The script first connects as system/manager and then drops the objects,
```

```
# after which it creates them. If you have errors on dropping the objects,
# these are normal the first time you run the script because the objects
# do not yet exist.
#
# REVISION HISTORY:
#
# 25-jan-2001       Initial Creation    Damir Bersinic
#
#
#
# Connect to the default instance
connect system/manager;
#
# Drop the user STUDENT, if it exists
DROP USER Student CASCADE;
#
# Drop the CERTDB tablespace, if it exists
DROP TABLESPACE CERTDB INCLUDING CONTENTS;
#
# Delete the file from the hard drive (for Windows).
# Comment out for running on Unix.
# Change the path if required.
HOST "DEL C:\CERTDB\CERTDB01.DBF";
#
# Delete the file from the hard drive (for Linux/Unix).
# Uncomment the HOST command to run on Unix.
# Change the path if required.
# HOST "rm /CERTDB/CERTDB01.DBF";
#
# Create the tablespace (for Windows)
# Comment out for running on Unix.
# Change the path if required.
CREATE TABLESPACE CERTDB DATAFILE 'C:\CERTDB\CERTDB01.DBF' SIZE 10M;
#
# Create the tablespace (for Linux/Unix)
# Uncomment the CREATE TABLESPACE command to run on Unix.
# Change the path if required.
# CREATE TABLESPACE CERTDB DATAFILE '/CERTDB/CERTDB01.DBF' SIZE 10M;
#
# Create the USER and grant a quota on the CERTDB tablespace.
# Make sure a TEMP tablespace already exists in your database.
CREATE USER Student IDENTIFIED BY oracle
       DEFAULT TABLESPACE CERTDB
       TEMPORARY TABLESPACE TEMP
       QUOTA UNLIMITED ON CERTDB;
#
# Allow the STUDENT user to connect and logon to the instance.
GRANT CONNECT,RESOURCE TO Student;
```

CERTDBOBJ.SQL

```
# This script creates the tables in the STUDENT user schema.
#
# The script first connects as student/oracle and then drops the objects,
```

```
# after which it creates them. If you have errors on dropping the objects,
# these are normal the first time you run the script because the objects
# do not yet exist.
#
# REVISION HISTORY:
#
# 25-jan-2001       Initial Creation     Damir Bersinic
#
#
# Connect to the default instance as user STUDENT.
connect student/oracle;
#
#
# Drop all tables and cascade constraints
DROP TABLE Courses CASCADE CONSTRAINTS;
DROP TABLE Instructors CASCADE CONSTRAINTS;
DROP TABLE Locations CASCADE CONSTRAINTS;
DROP TABLE Students CASCADE CONSTRAINTS;
DROP TABLE ScheduledClasses CASCADE CONSTRAINTS;
DROP TABLE ClassEnrollment CASCADE CONSTRAINTS;
DROP TABLE BatchJobs CASCADE CONSTRAINTS;
DROP TABLE CourseAudit CASCADE CONSTRAINTS;
#
#
# Create each table in the appropriate order on the CERTDB
# tablespace. All tables will be owned by STUDENT.
#
CREATE TABLE Courses (
       CourseNumber number(5) NOT NULL
             CONSTRAINT PK_CourseNumber PRIMARY KEY ,
       CourseName varchar2 (200) NOT NULL ,
       ReplacesCourse number(5) NULL ,
       RetailPrice number (9,2) NOT NULL ,
       Description varchar2 (2000) NULL
) TABLESPACE CERTDB;

CREATE TABLE Instructors (
       InstructorID number(5) NOT NULL
              CONSTRAINT PK_InstructorID PRIMARY KEY,
       Salutation char (4) NULL ,
       LastName varchar2 (30) NOT NULL ,
       FirstName varchar2 (30) NOT NULL ,
       MiddleInitial varchar2 (5) NULL ,
       Address1 varchar2 (50) NULL ,
       Address2 varchar2 (50) NULL ,
       City varchar2 (30) NULL ,
       State char (2) NULL ,
       Country varchar2 (30) NULL ,
       PostalCode char (10) NULL ,
       OfficePhone char (15) NULL ,
       HomePhone char (15) NULL ,
       CellPhone char (15) NULL ,
       EMail varchar2 (50) NULL ,
       InstructorType char(10) NOT NULL,
```

```
       PerDiemCost number (9,2) NULL ,
       PerDiemExpenses number (9,2) NULL ,
       Comments varchar2 (2000) NULL
) TABLESPACE CERTDB;

CREATE TABLE Locations (
       LocationID number(5) NOT NULL
              CONSTRAINT PK_LocationID PRIMARY KEY ,
       LocationName varchar2 (50) NOT NULL ,
       Address1 varchar2 (50) NULL ,
       Address2 varchar2 (50) NULL ,
       City varchar2 (30) NULL ,
       State char (2) NULL ,
       Country varchar2 (30) NULL ,
       PostalCode char (10) NULL ,
       Telephone char (15) NULL ,
       Fax char (15) NULL ,
       Contact varchar2 (50) NULL ,
       Description varchar2 (2000) NULL
) TABLESPACE CERTDB;

CREATE TABLE Students (
       StudentNumber number(5) NOT NULL
              CONSTRAINT PK_StudentNumber PRIMARY KEY,
       Salutation char (4) NULL ,
       LastName varchar2 (30) NOT NULL ,
       FirstName varchar2 (30) NOT NULL ,
       MiddleInitial varchar2 (5) NULL ,
       Address1 varchar2 (50) NULL ,
       Address2 varchar2 (50) NULL ,
       City varchar2 (30) NULL ,
       State char (2) NULL ,
       Country varchar2 (30) NULL ,
       PostalCode char (10) NULL ,
       HomePhone char (15) NULL ,
       WorkPhone char (15) NULL ,
       EMail varchar2 (50) NULL ,
       Comments varchar2 (2000) NULL
) TABLESPACE CERTDB;

CREATE TABLE ScheduledClasses (
       ClassID number(5) NOT NULL
             CONSTRAINT PK_ClassID PRIMARY KEY,
       CourseNumber number(5) NOT NULL ,
       LocationID number(5) NOT NULL ,
       ClassRoomNumber number(3) NOT NULL ,
       InstructorID number(5) NOT NULL ,
       StartDate date NOT NULL ,
       DaysDuration number(3) NOT NULL ,
       Status char (10) NOT NULL ,
       Comments varchar2 (2000) NULL
) TABLESPACE CERTDB;
```

```
CREATE TABLE ClassEnrollment (
       ClassID number(5) NOT NULL ,
       StudentNumber number(5) NOT NULL ,
       Status char (10) NOT NULL ,
       EnrollmentDate date NOT NULL,
       Price number (9,2) NOT NULL ,
       Grade char (4) NULL ,
       Comments varchar2 (2000) NULL
) TABLESPACE CERTDB;

CREATE TABLE BatchJobs (
       JobId number (6) NOT NULL,
       JobName      varchar2 (30) NOT NULL,
       Status varchar2 (30) NOT NULL,
       LastUpdated date NOT NULL
) TABLESPACE CERTDB;

CREATE TABLE CourseAudit (
       CourseNumber number(5) NOT NULL,
       Change varchar2 (30) NOT NULL,
       DateChanged date NOT NULL,
       Price number (9,2),
       ChangedBy varchar2 (15)
) TABLESPACE CERTDB;
#
#
#
# Alter the tables to include foreign keys and composite primary keys

ALTER TABLE ClassEnrollment ADD
       (CONSTRAINT PK_ClassID_StudentNumber
                PRIMARY KEY (ClassID, StudentNumber)) ;

ALTER TABLE ScheduledClasses ADD
       (CONSTRAINT FK_SchedClass_CourseNum
                FOREIGN KEY (CourseNumber) REFERENCES Courses (CourseNumber));

ALTER TABLE ScheduledClasses ADD
       (CONSTRAINT FK_SchedClasses_LocationID
                FOREIGN KEY (LocationID) REFERENCES Locations (LocationID));

ALTER TABLE ScheduledClasses ADD
       (CONSTRAINT FK_SchedClasses_InstID
                FOREIGN KEY (InstructorID)
                    REFERENCES Instructors (InstructorID));

ALTER TABLE ClassEnrollment ADD
       (CONSTRAINT FK_ClassEnrollment_ClassID
                FOREIGN KEY (ClassID) REFERENCES ScheduledClasses (ClassID));

ALTER TABLE ClassEnrollment ADD
       (CONSTRAINT FK_ClassEnrollment_StudentNum
               FOREIGN KEY (StudentNumber) REFERENCES Students (StudentNumber));
```

```
ALTER TABLE BatchJobs ADD
       (CONSTRAINT BatchJobs_JobId_pk PRIMARY KEY(JobId));

ALTER TABLE CourseAudit ADD
       (CONSTRAINT CourseAudit_PK
             PRIMARY KEY(CourseNumber, Change, DateChanged));

ALTER TABLE CourseAudit ADD
       (CONSTRAINT FK_CourseAudit_CourseNumber
              FOREIGN KEY (CourseNumber) REFERENCES Courses(CourseNumber));
```

INSERT_DATA.SQL

```
# This script populates the tables in the STUDENT user schema.
#
# This script assumes you are connected as student/oracle.
#
# The script first deletes data from each of the tables and then performs
inserts
# to load the sample data.
#
#
# REVISION HISTORY:
#
# 25-jan-2001       Initial Creation           Damir Bersinic
# 11-feb-2001       New instructors/Updates           Damir Bersinic
#
#
#
#
# Disable foreign key constraints to ensure TRUNCATE works.
#
ALTER TABLE ScheduledClasses DISABLE CONSTRAINT FK_SchedClass_CourseNum;

ALTER TABLE ScheduledClasses DISABLE CONSTRAINT FK_SchedClasses_LocationID;

ALTER TABLE ScheduledClasses DISABLE CONSTRAINT FK_SchedClasses_InstID;

ALTER TABLE ClassEnrollment DISABLE CONSTRAINT FK_ClassEnrollment_ClassID;

ALTER TABLE ClassEnrollment DISABLE CONSTRAINT FK_ClassEnrollment_StudentNum;

ALTER TABLE CourseAudit DISABLE CONSTRAINT FK_CourseAudit_CourseNumber;
#
#
# Truncate all tables in the appropriate order.
# This is required to ensure primary keys are not violated.
#
TRUNCATE TABLE BatchJobs;
TRUNCATE TABLE ClassEnrollment;
TRUNCATE TABLE ScheduledClasses;
TRUNCATE TABLE CourseAudit;
TRUNCATE TABLE Locations;
TRUNCATE TABLE Courses;
```

```
TRUNCATE TABLE Instructors;
TRUNCATE TABLE Students;
#
#
# Enable foreign key constraints to ensure proper data load
#
ALTER TABLE ScheduledClasses ENABLE CONSTRAINT FK_SchedClass_CourseNum;

ALTER TABLE ScheduledClasses ENABLE CONSTRAINT FK_SchedClasses_LocationID;

ALTER TABLE ScheduledClasses ENABLE CONSTRAINT FK_SchedClasses_InstID;

ALTER TABLE ClassEnrollment ENABLE CONSTRAINT FK_ClassEnrollment_ClassID;

ALTER TABLE ClassEnrollment ENABLE CONSTRAINT FK_ClassEnrollment_StudentNum;

ALTER TABLE CourseAudit ENABLE CONSTRAINT FK_CourseAudit_CourseNumber;
#
#
# Insert sample data into the Students table.
#
INSERT INTO STUDENTS VALUES
(1000,'Mr','Smith','John','H','34 Anystreet',null, 'Victoria','BC','Canada','V5F
3E8',
'904-567-8889','904-787-8888','james@emailrus.com',null);

INSERT INTO STUDENTS VALUES
(1001,'Mr','Jones','Davey',null,'10 Main St',null, 'New
York','NY','USA','87653',
'312-334-8889','312-642-5134','djones@hitech.com',null);

INSERT INTO STUDENTS VALUES
(1002,'Mrs','Massey','Jane','S','723 Church St',null, 'New
York','NY','USA','87654',
'412-324-0880','412-887-7489','jmassey@hitech.com',null);

INSERT INTO STUDENTS VALUES
(1003,'Mr','Smith','Trevor','J','13 Crosswood Cres',null,
'Toronto','ON','Canada','M5T 5F6',
'416-456-7890',null,'trevorsmtih@comptel.com',null);

INSERT INTO STUDENTS VALUES
(1004,null,'Hogan','Mike',null,'49 Bentbrook Cres',null,
'Ottawa','ON','Canada','K4M 1Y5',
'613-765-4321','613-567-1234','mhogan@consulters.com',null);

INSERT INTO STUDENTS VALUES
(1005,'Mr','Hee','John','K','Apt 7','90th Street', 'New
York','NY','USA','76990',
'412-567-8673','412-747-6543','johnhee@emailrus.com',null);

INSERT INTO STUDENTS VALUES
(1006,'Mrs','Andrew','Susan','M','15 King St',null, 'Dallas','TX','USA','87654',
'492-667-8889','492-875-9876','sandrew@bigtime.com',null);
```

```
INSERT INTO STUDENTS VALUES
(1007,'Mrs','Holland','Roxanne',null,'212 Lorne St',null, 'San
Francisco','CA','USA','77765',
'721-557-8567','721-787-5538','rholland@bigtime.com',null);

INSERT INTO STUDENTS VALUES
(1008,'Mr','Jones','Gordon',null,'17 Nisku',null, 'Toronto','ON','Canada','T2L
4R8',
'416-663-5689','416-645-5246','gordonjones@wesell.ca',null);

INSERT INTO STUDENTS VALUES
(1009,'Mrs','Colter','Sue','J','1112 Queen St',null, 'San
Francisco','CA','USA','56443',
'721-566-8645','721-744-4756','suecolter@compstore.com',null);

INSERT INTO STUDENTS VALUES
(1010,'Mr','Patterson','Chris','M','72 Regent St',null, 'San
Fransisco','CA','USA',
'57572','721-445-5239','721-547-3256','cpatterson@emailrus.com',null);
#
#
# Insert sample data into the Courses table.
#
INSERT INTO COURSES VALUES
(100,'Basic SQL',null,2000,'An introduction to basic SQL statements and
commands');

INSERT INTO COURSES VALUES
(110,'Advanced SQL',null,2000,
'Advanced SQL statements and commands for exerienced users');

INSERT INTO COURSES VALUES
(201,'Performance Tuning your Database',200,4000,
'Concepts and tricks to tune your database for optimum performance');

INSERT INTO COURSES VALUES
(200,'Database Performance Basics',null,4000,
'How to tune your database for maximum performance');

INSERT INTO COURSES VALUES
(210,'Database Administration',null,4500,
'Everything the DBA needs to know to start building a database');

INSERT INTO COURSES VALUES
(220,'Backing up your database',null,3000,
'The essentials for backing up and recovering the database after a failure');

INSERT INTO COURSES VALUES
(300,'Basic PL/SQL',null,2500,
'An introduction to the PL/SQL programming language');

INSERT INTO COURSES VALUES
(310,'Advanced PL/SQL',null,2000,
```

```
'A follow-up to the basic PL/SQL course that introduces complicated PL/SQL
programming techniques');

INSERT INTO COURSES VALUES
(320,'Using your PL/SQLskills',null,1750,
'Introduces database triggers and database packages');
#
#
# Insert sample data into the Locations table.
#
INSERT INTO locations VALUES
(100,'New York Park Ave','80 Park Ave',null,'New York','NY','USA','66578',
'412-389-8889','412-389-8859','Charlene Moore',
'Beautiful location overlooking fabulous central park, easy access to subway');

INSERT INTO locations VALUES
(200,'San Francisco Downtown','40 Bay St',null,'San
Francisco','CA','USA','85763',
'721-765-0987','721-765-9421','James Madison',
'Located in downtown San Francisco, you can see Alcatraz on a clear day');

INSERT INTO locations VALUES
(300,'Downtown Toronto','40 Yonge Street',null,'Toronto','ON','Canada','M6H
5K8',
'416-543-8768','416-544-3965','Joanne Matthews',
'Convenient downtown location, easy access to subway');
#
#
# Insert sample data into the Instructors table.
#
INSERT INTO instructors VALUES
(300,'Mr','Harrison','Michael','H','8899 Eglinton
Ave',null,'Toronto','ON','Canada','M7H 6H5',
'416-543-8769','416-778-5366',null,'michaelharrison@trainers.com','ORACLE',500,2
00,null);

INSERT INTO instructors VALUES
(310,'Mrs','Keele','Susan','J','42 Bloor St',null,'Toronto','ON','Canada','M5T
5F7',
'416-543-8775','416-857-9876',null,'susankeele@trainers.com','UNIX',450,200,null
);

INSERT INTO instructors VALUES
(100,'Mr','Ungar','David','J','995 White Plains Ave',null,'New
York','NY','USA','98750',
'412-389-6557','412-345-6543',null,'davidungar@trainers.com','ORACLE',600,200,nu
ll);

INSERT INTO instructors VALUES
(110,'Mr','Jamieson','Kyle','L','Apt 86','95 Cornerbrook St','New
York','NY','USA','87653',
'412-389-7683','412-889-0987','412-987-0423','kylejamieson@trainers.com','ORACLE
',500,200,null);
```

```
INSERT INTO instructors VALUES
(200,'Miss','Cross','Lisa','M','45 Sunny Drive',null,'Palo
Alto','CA','USA','89075',
'721-765-9985','721-649-0944',null,'lisacross@trainers.com','UNIX',750,250,null)
;

INSERT INTO instructors VALUES
(210, 'Mr.', 'Williams', 'Geoff', 'R', '432 Main St.', null, 'Toronto', 'ON',
'Canada','M3A 1W1',
'416-543-8778', '416 555-1345' , null, 'gwilliams@trainers.com', 'UNIX', 400,
null,
'Has extensive real world experience with Borne Shell scripting');

INSERT INTO Instructors VALUES
(410, 'Ms.', 'Chiu', 'Lana', '', '30 High Gate', null, 'New York', 'NY',
'USA','07653',
'212-555-8744', '212 555-7345' , null, 'LChiu@trainers.com', 'Prog', 400, null,
'Has 10 years experience with C and C++.');

INSERT INTO Instructors VALUES
(450, 'Ms.', 'LaPoint', 'Adele', 'S', '1435 Redwood Dr.', 'Apt 1212', 'New
York', 'NY', 'USA',
'07653','212-555-8754', '212 555-7546' , null, 'ALapoint@trainers.com', 'Prog',
450, 200,
'Is fluent in French');

#
#
# Insert sample data into the ScheduledClasses table.
#
INSERT INTO scheduledclasses VALUES
(50,100,100,4,100,'06-jan-2001',4,'Confirmed',null);

INSERT INTO scheduledclasses VALUES
(51,200,300,1,200,'13-jan-2001',5,'Confirmed',null);

INSERT INTO scheduledclasses VALUES
(53,100,300,2,110,'14-feb-2001',4,'Hold',null);
#
#
# Insert sample data into the ClassEnrollment table.
#
INSERT INTO classenrollment (ClassId, StudentNumber, Status, EnrollmentDate,
Price, Grade, Comments)
VALUES (50, 1001, 'Confirmed','01-JAN-2001',2000, 'B', null);

INSERT INTO classenrollment (ClassId, StudentNumber, Status, EnrollmentDate,
Price, Grade, Comments)
VALUES (50, 1002, 'Confirmed','12-DEC-2000',1750, 'A', null);

INSERT INTO classenrollment (ClassId, StudentNumber, Status, EnrollmentDate,
Price, Grade, Comments)
VALUES (50, 1005, 'Confirmed','21-DEC-2000',2000, 'F',
'Missed last two days of class - will resit in March');
```

```
INSERT INTO classenrollment (ClassId, StudentNumber, Status, EnrollmentDate,
Price, Grade, Comments)
VALUES (51, 1003, 'Cancelled','01-JAN-2001',4000, null, null);

INSERT INTO classenrollment (ClassId, StudentNumber, Status, EnrollmentDate,
Price, Grade, Comments)
VALUES (51, 1004, 'Confirmed','5-JAN-2001',4000, 'A', null);

INSERT INTO classenrollment (ClassId, StudentNumber, Status, EnrollmentDate,
Price, Grade, Comments)
VALUES (51, 1008, 'Confirmed','02-DEC-2000',3500, 'A', null);

INSERT INTO classenrollment (ClassId, StudentNumber, Status, EnrollmentDate,
Price, Grade, Comments)
VALUES (53, 1003, 'Hold','02-JAN-2001',1500, null, null);
#
#
# Insert sample data into the BatchJobs table.
#
INSERT INTO BatchJobs (JobId, JobName, Status, LastUpdated)
VALUES (100, 'CLASS_STATUS','RUNNING','01-MAR-2001');

INSERT INTO BatchJobs (JobId, JobName, Status, LastUpdated)
VALUES (101, 'PRINT_REGISTRATION','COMPLETED','12-MAR-2001');

INSERT INTO BatchJobs (JobId, JobName, Status, LastUpdated)
VALUES (102, 'CALCULATE_REVENUE','COMPLETED','01-MAR-2001');
#
#
# Perform some updates to the data required in chapter 2.
UPDATE Students
SET City = 'San Francisco'
WHERE StudentNumber = 1010;

UPDATE Instructors
SET Comments = 'Has experience with Solaris and HP-UX'
WHERE InstructorID = 200;

UPDATE Instructors
SET Comments = 'Has extensive shell scripting experience.  Has also programmed
with C and C++'
WHERE InstructorID = 210;
#
#
# Commit the changes to the database.
COMMIT;
```

APPENDIX F

Data Dictionary Views

Oracle 8*i* includes a number of data dictionary views that can be used to get information from the data dictionary about the various objects created and their properties. Throughout this book, you have seen how to extract information about these objects using **USER_** views. Table F-1 lists these views, followed by their structure.

Table F-1 does not list all of the **USER_** views that are available but concentrates on those views mentioned in the book or that are in other ways useful. Not all of the views listed in the table were mentioned in the text because many objects may be created in Oracle in addition to the ones to which you were introduced. For more information on the objects not covered in the book, refer to the Oracle documentation.

Table F-1
USER_ Views in Oracle 8*i*

View	*Description*
USER_ALL_TABLES	Lists the names and additional attributes of all tables in the user's schema, i.e., all tables owned by the user. This view has the same structure as USER_TABLES.
USER_ARGUMENTS	Displays arguments for parts of a package in the user's schema.
USER_CATALOG	Lists all tables, sequences, and views in the user's schema.
USER_CLUSTERS	Lists all clusters and additional attributes of clusters in the user's schema.
USER_CLUSTER_ HASH_EXPRESSIONS	Lists the code used to calculate the hash value for a hash cluster in the user's schema when a user-defined hash expression is used.
USER_CLU_COLUMNS	Lists the cluster columns for all clusters in the user's schema, as well as the corresponding column names that map to the cluster column for all tables on the cluster.
USER_COL_COMMENTS	Lists the comments added to the dictionary for all columns in all tables in the user's schema.
USER_COL_PRIVS	Lists the privileges granted to the user by others or granted by the user to others on columns for a table.
USER_COL_PRIVS_MADE	Lists the privileges granted by the user on columns for a table.
USER_COL_PRIVS_RECD	Lists the privileges granted to the user on columns for a table in other user's schemas.
USER_CONSTRAINTS	Displays information about all constraints created by the user in any schema.
USER_CONS_COLUMNS	Displays information about the columns on which the user has created constraints.
USER_ERRORS	Lists errors for PL/SQL blocks executed by the user. This includes anonymous PL/SQL blocks as well as stored programs like procedures, functions, and packages.
USER_EXTENTS	Lists information on the extents allocated to segments created by the user. Segments include tables, indexes, clusters, partitions, and materialized views.
USER_FREE_SPACE	Lists the free space available to the user on the tablespaces to which he/she has been granted a quota.
USER_INDEXES	Lists information on all indexes created by the user on objects in the database.
USER_IND_COLUMNS	Lists the columns and the position (in a composite index) of objects in the database on which the user created indexes.

View	*Description*
USER_IND_EXPRESSIONS	For function-based indexes, lists the column, its position in the index, and expression applied for the index created by the user.
USER_IND_PARTITIONS	Displays information about the partitions for partitioned indexes created by in the user.
USER_IND_SUBPARTITIONS	Displays information about the subpartitions for subpartitioned indexes created by the user.
USER_LIBRARIES	Displays information on the PL/SQL code libraries created by the user. Libraries are used to store common pieces of code that may be used in different parts of the application.
USER_LOBS	Displays information on LOB columns (i.e., those with a datatype of BLOB, CLOB, NCLOB, or BFILE) belonging to tables created in the user's schema.
USER_LOB_PARTITIONS	Displays information on partitions of tables in the user schema containing LOBs.
USER_LOB_SUBPARTITIONS	Displays information on sub partitions of tables in the user's schema containing LOBs.
USER_METHOD_PARAMS	Lists information on parameters for methods on objects created in the user's schema. Methods are PL/SQL or Java program units that belong to the object.
USER_METHOD_RESULTS	Lists information on results for methods on objects created in the user's schema.
USER_NESTED_TABLES	Displays the parent table and other information for nested tables within tables created in the user's schema.
USER_OBJECTS	Displays information about all objects in the user's schema.
USER_OBJECT_SIZE	Displays information on the size of source code, parsed code, compiled code, and error code for program units created in the user's schema.
USER_OBJECT_TABLES	Displays information on tables in the user's schema that are based on user-defined objects.
USER_PART_ COL_STATISTICS	Displays information on statistics for partition columns in the user's schema as calculated by running the ANALYZE command.
USER_PART_HISTOGRAMS	Displays information on histograms calculated by the ANALYZE command on partitions in the user's schema.

Continued

Table F-1 *(continued)*

View	*Description*
USER_PART_INDEXES	Displays information on partitioned indexes in the user's schema.
USER_PART_ KEY_COLUMNS	Displays information on the key columns for partitioned tables in the user's schema.
USER_PART_LOBS	Displays information on storage parameters of LOBs contained in partitioned tables in the user's schema.
USER_PART_TABLES	Displays information on storage parameters for partitioned tables in the user's schema.
USER_PASSWORD_LIMITS	Lists the current password management limits in effect for the user.
USER_RESOURCE_LIMITS	Lists the current resource limits (CPU, I/O, etc.) in effect for the user.
USER_ROLE_PRIVS	Lists the roles and their options currently available for the user.
USER_SEGMENTS	Displays storage properties of segments created by the user. Segments may include tables, indexes, materialized views, partitions, and LOBs.
USER_SEQUENCES	Displays information on sequences created in the user's schema.
USER_SOURCE	Displays the lines for the source code of procedures, functions, triggers, methods, and other PL/SQL stored objects in the user's schema.
USER_SUBPART_ COL_STATISTICS	Displays information on statistics for subpartition columns in the user's schema as calculated by running the ANALYZE command.
USER_SUBPART_ HISTOGRAMS	Displays information on histograms calculated by the ANALYZE command on subpartitions in the user's schema.
USER_SUBPART_ KEY_COLUMNS	Displays information on the key columns for subpartitioned tables in the user's schema
USER_SYNONYMS	Lists synonyms in the user's schema and the objects to which they refer.
USER_SYS_PRIVS	Lists the system privileges granted to the users and their options.
USER_TABLES	Displays information on tables created in the user's schema and, if an ANALYZE has been performed, statistics for them.
USER_TABLESPACES	Displays information on tablespaces to which the user has access and their default characteristics.

View	*Description*
USER_TAB_COLUMNS	Displays information on columns in tables created in the user's schema. May also provide statistical information if an ANALYZE has been performed.
USER_TAB_COL_STATISTICS	Displays statistics on columns of tables created in the user's schema after an ANALYZE has been performed.
USER_TAB_COMMENTS	Displays comments added to the data dictionary for tables in the user's schema.
USER_TAB_HISTOGRAMS	Displays the results after histograms have been created for columns on a table by running ANALYZE.
USER_TAB_MODIFICATIONS	Displays information on the number of changes of each type made to tables in the user's schema since the last time the ANALYZE command was run.
USER_TAB_PARTITIONS	Displays information on table partitions in the user's schema, including statistics if an ANALYZE was run.
USER_TAB_PRIVS	Lists all permissions that were granted on tables to others by the user, granted on tables by others to the user, or granted on tables in the user's schema. This includes tables owned by the user, those owned by the others to which the user has been given permissions, or those tables to which the user has granted others permissions.
USER_TAB_PRIVS_MADE	Lists permissions granted to others by the user for tables in the user's schema.
USER_TAB_PRIVS_RECD	Lists permissions granted to the user by others for tables not in the user's schema.
USER_TAB_SUBPARTITIONS	Displays information on the subpartitions on tables owned by the user.
USER_TRIGGERS	Displays information on triggers created by the user, as well as the trigger source code.
USER_TRIGGER_COLS	Lists the columns, and their usage, specified in triggers in the user's schema.
USER_TS_QUOTAS	Displays information on tablespaces in which the user has been granted a quota (i.e., storage space).
USER_TYPES	Displays information on object types in the user's schema.
USER_TYPE_ATTRS	Displays information on attributes of object types in the user's schema.

Continued

Table F-1 *(continued)*

View	*Description*
USER_TYPE_METHODS	Displays information on object type methods for object types created in the user's schema.
USER_UNUSED_COL_TABS	Lists the number of unused columns for tables in the user's schema. Unused columns are those that have been marked unused in the table but not yet deleted from the table.
USER_UPDATABLE_COLUMNS	Lists the columns in a join view that can be updated by the user and the types of operations that can be performed (e.g, INSERT, UPDATE, DELETE).
USER_USERS	Displays information on the current user such as default and temporary tablespace settings and password expiration date.
USER_VARRAYS	Displays information on all VARRAYs in the user's schema.
USER_VIEWS	Displays information on all views created in the user's schema.

Structure of USER_ Views in Oracle

USER_ALL_TABLES

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                              VARCHAR2(30)
TABLESPACE_NAME                         VARCHAR2(30)
CLUSTER_NAME                            VARCHAR2(30)
IOT_NAME                                VARCHAR2(30)
PCT_FREE                                NUMBER
PCT_USED                                NUMBER
INI_TRANS                               NUMBER
MAX_TRANS                               NUMBER
INITIAL_EXTENT                          NUMBER
NEXT_EXTENT                             NUMBER
MIN_EXTENTS                             NUMBER
MAX_EXTENTS                             NUMBER
PCT_INCREASE                            NUMBER
FREELISTS                               NUMBER
FREELIST_GROUPS                         NUMBER
LOGGING                                 VARCHAR2(3)
BACKED_UP                               VARCHAR2(1)
NUM_ROWS                                NUMBER
BLOCKS                                  NUMBER
EMPTY_BLOCKS                            NUMBER
AVG_SPACE                               NUMBER
CHAIN_CNT                               NUMBER
```

```
AVG_ROW_LEN                              NUMBER
AVG_SPACE_FREELIST_BLOCKS                NUMBER
NUM_FREELIST_BLOCKS                      NUMBER
DEGREE                                   VARCHAR2(10)
INSTANCES                                VARCHAR2(10)
CACHE                                    VARCHAR2(5)
TABLE_LOCK                               VARCHAR2(8)
SAMPLE_SIZE                              NUMBER
LAST_ANALYZED                            DATE
PARTITIONED                              VARCHAR2(3)
IOT_TYPE                                 VARCHAR2(12)
OBJECT_ID_TYPE                           VARCHAR2(16)
TABLE_TYPE_OWNER                         VARCHAR2(30)
TABLE_TYPE                               VARCHAR2(30)
TEMPORARY                                VARCHAR2(1)
SECONDARY                                VARCHAR2(1)
NESTED                                   VARCHAR2(3)
BUFFER_POOL                              VARCHAR2(7)
ROW_MOVEMENT                             VARCHAR2(8)
GLOBAL_STATS                             VARCHAR2(3)
USER_STATS                               VARCHAR2(3)
DURATION                                 VARCHAR2(15)
SKIP_CORRUPT                             VARCHAR2(8)
MONITORING                               VARCHAR2(3)
CLUSTER_OWNER                            VARCHAR2(30)
```

USER_ARGUMENTS

```
Column Name                    Null?    Type
------------------------------ -------- ----
OBJECT_NAME                             VARCHAR2(30)
PACKAGE_NAME                            VARCHAR2(30)
OBJECT_ID                      NOT NULL NUMBER
OVERLOAD                                VARCHAR2(40)
ARGUMENT_NAME                           VARCHAR2(30)
POSITION                       NOT NULL NUMBER
SEQUENCE                       NOT NULL NUMBER
DATA_LEVEL                     NOT NULL NUMBER
DATA_TYPE                               VARCHAR2(30)
DEFAULT_VALUE                           LONG
DEFAULT_LENGTH                          NUMBER
IN_OUT                                  VARCHAR2(9)
DATA_LENGTH                             NUMBER
DATA_PRECISION                          NUMBER
DATA_SCALE                              NUMBER
RADIX                                   NUMBER
CHARACTER_SET_NAME                      VARCHAR2(44)
TYPE_OWNER                              VARCHAR2(30)
TYPE_NAME                               VARCHAR2(30)
TYPE_SUBNAME                            VARCHAR2(30)
TYPE_LINK                               VARCHAR2(128)
PLS_TYPE                                VARCHAR2(30)
```

USER_CATALOG

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                     NOT NULL VARCHAR2(30)
TABLE_TYPE                              VARCHAR2(11)
```

USER_CLUSTERS

```
Column Name                    Null?    Type
------------------------------ -------- ----
CLUSTER_NAME                   NOT NULL VARCHAR2(30)
TABLESPACE_NAME                NOT NULL VARCHAR2(30)
PCT_FREE                                NUMBER
PCT_USED                       NOT NULL NUMBER
KEY_SIZE                                NUMBER
INI_TRANS                      NOT NULL NUMBER
MAX_TRANS                      NOT NULL NUMBER
INITIAL_EXTENT                          NUMBER
NEXT_EXTENT                             NUMBER
MIN_EXTENTS                    NOT NULL NUMBER
MAX_EXTENTS                    NOT NULL NUMBER
PCT_INCREASE                            NUMBER
FREELISTS                               NUMBER
FREELIST_GROUPS                         NUMBER
AVG_BLOCKS_PER_KEY                      NUMBER
CLUSTER_TYPE                            VARCHAR2(5)
FUNCTION                                VARCHAR2(15)
HASHKEYS                                NUMBER
DEGREE                                  VARCHAR2(10)
INSTANCES                               VARCHAR2(10)
CACHE                                   VARCHAR2(5)
BUFFER_POOL                             VARCHAR2(7)
SINGLE_TABLE                            VARCHAR2(5)
```

USER_CLUSTER_HASH_EXPRESSIONS

```
Column Name                    Null?    Type
------------------------------ -------- ----
OWNER                          NOT NULL VARCHAR2(30)
CLUSTER_NAME                   NOT NULL VARCHAR2(30)
HASH_EXPRESSION                         LONG
```

USER_CLU_COLUMNS

```
Column Name                    Null?    Type
------------------------------ -------- ----
CLUSTER_NAME                   NOT NULL VARCHAR2(30)
CLU_COLUMN_NAME                NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
TAB_COLUMN_NAME                         VARCHAR2(4000)
```

USER_COL_COMMENTS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                     NOT NULL VARCHAR2(30)
COLUMN_NAME                    NOT NULL VARCHAR2(30)
COMMENTS                                VARCHAR2(4000)
```

USER_COL_PRIVS

```
Column Name                    Null?    Type
------------------------------ -------- ----
GRANTEE                        NOT NULL VARCHAR2(30)
OWNER                          NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
COLUMN_NAME                    NOT NULL VARCHAR2(30)
GRANTOR                        NOT NULL VARCHAR2(30)
PRIVILEGE                      NOT NULL VARCHAR2(40)
GRANTABLE                               VARCHAR2(3)
```

USER_COL_PRIVS_MADE

```
Column Name                    Null?    Type
------------------------------ -------- ----
GRANTEE                        NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
COLUMN_NAME                    NOT NULL VARCHAR2(30)
GRANTOR                        NOT NULL VARCHAR2(30)
PRIVILEGE                      NOT NULL VARCHAR2(40)
GRANTABLE                               VARCHAR2(3)
```

USER_COL_PRIVS_RECD

```
Column Name                    Null?    Type
------------------------------ -------- ----
OWNER                          NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
COLUMN_NAME                    NOT NULL VARCHAR2(30)
GRANTOR                        NOT NULL VARCHAR2(30)
PRIVILEGE                      NOT NULL VARCHAR2(40)
GRANTABLE                               VARCHAR2(3)
```

USER_CONSTRAINTS

```
Column Name                    Null?    Type
------------------------------ -------- ----
OWNER                          NOT NULL VARCHAR2(30)
CONSTRAINT_NAME                NOT NULL VARCHAR2(30)
CONSTRAINT_TYPE                         VARCHAR2(1)
TABLE_NAME                     NOT NULL VARCHAR2(30)
SEARCH_CONDITION                        LONG
R_OWNER                                 VARCHAR2(30)
R_CONSTRAINT_NAME                       VARCHAR2(30)
DELETE_RULE                             VARCHAR2(9)
```

```
STATUS                                    VARCHAR2(8)
DEFERRABLE                                VARCHAR2(14)
DEFERRED                                  VARCHAR2(9)
VALIDATED                                 VARCHAR2(13)
GENERATED                                 VARCHAR2(14)
BAD                                       VARCHAR2(3)
RELY                                      VARCHAR2(4)
LAST_CHANGE                               DATE
```

USER_CONS_COLUMNS

```
Column Name                    Null?    Type
------------------------------ -------- ----
OWNER                          NOT NULL VARCHAR2(30)
CONSTRAINT_NAME                NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
POSITION                                NUMBER
```

USER_DEPENDENCIES

```
Column Name                    Null?    Type
------------------------------ -------- ----
NAME                           NOT NULL VARCHAR2(30)
TYPE                                    VARCHAR2(12)
REFERENCED_OWNER                        VARCHAR2(30)
REFERENCED_NAME                         VARCHAR2(64)
REFERENCED_TYPE                         VARCHAR2(12)
REFERENCED_LINK_NAME                    VARCHAR2(128)
SCHEMAID                                NUMBER
DEPENDENCY_TYPE                         VARCHAR2(4)
```

USER_ERRORS

```
Column Name                    Null?    Type
------------------------------ -------- ----
NAME                           NOT NULL VARCHAR2(30)
TYPE                                    VARCHAR2(12)
SEQUENCE                       NOT NULL NUMBER
LINE                           NOT NULL NUMBER
POSITION                       NOT NULL NUMBER
TEXT                           NOT NULL VARCHAR2(4000)
```

USER_EXTENTS

```
Column Name                    Null?    Type
------------------------------ -------- ----
SEGMENT_NAME                            VARCHAR2(81)
PARTITION_NAME                          VARCHAR2(30)
SEGMENT_TYPE                            VARCHAR2(18)
TABLESPACE_NAME                         VARCHAR2(30)
EXTENT_ID                               NUMBER
BYTES                                   NUMBER
BLOCKS                                  NUMBER
```

USER_FREE_SPACE

Column Name	Null?	Type
TABLESPACE_NAME		VARCHAR2(30)
FILE_ID		NUMBER
BLOCK_ID		NUMBER
BYTES		NUMBER
BLOCKS		NUMBER
RELATIVE_FNO		NUMBER

USER_INDEXES

Column Name	Null?	Type
INDEX_NAME	NOT NULL	VARCHAR2(30)
INDEX_TYPE		VARCHAR2(27)
TABLE_OWNER	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
TABLE_TYPE		VARCHAR2(11)
UNIQUENESS		VARCHAR2(9)
COMPRESSION		VARCHAR2(8)
PREFIX_LENGTH		NUMBER
TABLESPACE_NAME		VARCHAR2(30)
INI_TRANS		NUMBER
MAX_TRANS		NUMBER
INITIAL_EXTENT		NUMBER
NEXT_EXTENT		NUMBER
MIN_EXTENTS		NUMBER
MAX_EXTENTS		NUMBER
PCT_INCREASE		NUMBER
PCT_THRESHOLD		NUMBER
INCLUDE_COLUMN		NUMBER
FREELISTS		NUMBER
FREELIST_GROUPS		NUMBER
PCT_FREE		NUMBER
LOGGING		VARCHAR2(3)
BLEVEL		NUMBER
LEAF_BLOCKS		NUMBER
DISTINCT_KEYS		NUMBER
AVG_LEAF_BLOCKS_PER_KEY		NUMBER
AVG_DATA_BLOCKS_PER_KEY		NUMBER
CLUSTERING_FACTOR		NUMBER
STATUS		VARCHAR2(8)
NUM_ROWS		NUMBER
SAMPLE_SIZE		NUMBER
LAST_ANALYZED		DATE
DEGREE		VARCHAR2(40)
INSTANCES		VARCHAR2(40)
PARTITIONED		VARCHAR2(3)
TEMPORARY		VARCHAR2(1)

GENERATED	VARCHAR2(1)
SECONDARY	VARCHAR2(1)
BUFFER_POOL	VARCHAR2(7)
USER_STATS	VARCHAR2(3)
DURATION	VARCHAR2(15)
PCT_DIRECT_ACCESS	NUMBER
ITYP_OWNER	VARCHAR2(30)
ITYP_NAME	VARCHAR2(30)
PARAMETERS	VARCHAR2(1000)
GLOBAL_STATS	VARCHAR2(3)
DOMIDX_STATUS	VARCHAR2(12)
DOMIDX_OPSTATUS	VARCHAR2(6)
FUNCIDX_STATUS	VARCHAR2(8)

USER_IND_COLUMNS

Column Name	Null?	Type
INDEX_NAME		VARCHAR2(30)
TABLE_NAME		VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
COLUMN_POSITION		NUMBER
COLUMN_LENGTH		NUMBER
DESCEND		VARCHAR2(4)

USER_IND_EXPRESSIONS

Column Name	Null?	Type
INDEX_NAME		VARCHAR2(30)
TABLE_NAME		VARCHAR2(30)
COLUMN_EXPRESSION		LONG
COLUMN_POSITION		NUMBER

USER_IND_PARTITIONS

Column Name	Null?	Type
INDEX_NAME		VARCHAR2(30)
COMPOSITE		VARCHAR2(3)
PARTITION_NAME		VARCHAR2(30)
SUBPARTITION_COUNT		NUMBER
HIGH_VALUE		LONG
HIGH_VALUE_LENGTH		NUMBER
PARTITION_POSITION		NUMBER
STATUS		VARCHAR2(8)
TABLESPACE_NAME		VARCHAR2(30)
PCT_FREE		NUMBER
INI_TRANS		NUMBER
MAX_TRANS		NUMBER
INITIAL_EXTENT		NUMBER
NEXT_EXTENT		NUMBER
MIN_EXTENT		NUMBER

```
MAX_EXTENT                                NUMBER
PCT_INCREASE                              NUMBER
FREELISTS                                 NUMBER
FREELIST_GROUPS                           NUMBER
LOGGING                                   VARCHAR2(7)
COMPRESSION                               VARCHAR2(8)
BLEVEL                                    NUMBER
LEAF_BLOCKS                               NUMBER
DISTINCT_KEYS                             NUMBER
AVG_LEAF_BLOCKS_PER_KEY                   NUMBER
AVG_DATA_BLOCKS_PER_KEY                   NUMBER
CLUSTERING_FACTOR                         NUMBER
NUM_ROWS                                  NUMBER
SAMPLE_SIZE                               NUMBER
LAST_ANALYZED                             DATE
BUFFER_POOL                               VARCHAR2(7)
USER_STATS                                VARCHAR2(3)
PCT_DIRECT_ACCESS                         NUMBER
GLOBAL_STATS                              VARCHAR2(3)
```

USER_IND_SUBPARTITIONS

```
Column Name                    Null?    Type
------------------------------ -------- ----
INDEX_NAME                     NOT NULL VARCHAR2(30)
PARTITION_NAME                          VARCHAR2(30)
SUBPARTITION_NAME                       VARCHAR2(30)
SUBPARTITION_POSITION          NOT NULL NUMBER
STATUS                                  VARCHAR2(8)
TABLESPACE_NAME                NOT NULL VARCHAR2(30)
PCT_FREE                       NOT NULL NUMBER
INI_TRANS                      NOT NULL NUMBER
MAX_TRANS                      NOT NULL NUMBER
INITIAL_EXTENT                          NUMBER
NEXT_EXTENT                             NUMBER
MIN_EXTENT                     NOT NULL NUMBER
MAX_EXTENT                     NOT NULL NUMBER
PCT_INCREASE                            NUMBER
FREELISTS                               NUMBER
FREELIST_GROUPS                         NUMBER
LOGGING                                 VARCHAR2(3)
BLEVEL                                  NUMBER
LEAF_BLOCKS                             NUMBER
DISTINCT_KEYS                           NUMBER
AVG_LEAF_BLOCKS_PER_KEY                 NUMBER
AVG_DATA_BLOCKS_PER_KEY                 NUMBER
CLUSTERING_FACTOR                       NUMBER
NUM_ROWS                                NUMBER
SAMPLE_SIZE                             NUMBER
LAST_ANALYZED                           DATE
BUFFER_POOL                             VARCHAR2(7)
USER_STATS                              VARCHAR2(3)
GLOBAL_STATS                            VARCHAR2(3)
```

USER_LIBRARIES

Column Name	Null?	Type
LIBRARY_NAME	NOT NULL	VARCHAR2(30)
FILE_SPEC		VARCHAR2(2000)
DYNAMIC		VARCHAR2(1)
STATUS		VARCHAR2(7)

USER_LOBS

Column Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
SEGMENT_NAME	NOT NULL	VARCHAR2(30)
INDEX_NAME	NOT NULL	VARCHAR2(30)
CHUNK		NUMBER
PCTVERSION	NOT NULL	NUMBER
CACHE		VARCHAR2(10)
LOGGING		VARCHAR2(3)
IN_ROW		VARCHAR2(3)

USER_LOB_PARTITIONS

Column Name	Null?	Type
TABLE_NAME		VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
LOB_NAME		VARCHAR2(30)
PARTITION_NAME		VARCHAR2(30)
LOB_PARTITION_NAME		VARCHAR2(30)
LOB_INDPART_NAME		VARCHAR2(30)
PARTITION_POSITION		NUMBER
COMPOSITE		VARCHAR2(3)
CHUNK		NUMBER
PCTVERSION		NUMBER
CACHE		VARCHAR2(10)
IN_ROW		VARCHAR2(3)
TABLESPACE_NAME		VARCHAR2(30)
INITIAL_EXTENT		VARCHAR2(40)
NEXT_EXTENT		VARCHAR2(40)
MIN_EXTENTS		VARCHAR2(40)
MAX_EXTENTS		VARCHAR2(40)
PCT_INCREASE		VARCHAR2(40)
FREELISTS		VARCHAR2(40)
FREELIST_GROUPS		VARCHAR2(40)
LOGGING		VARCHAR2(7)
BUFFER_POOL		VARCHAR2(7)

USER_LOB_SUBPARTITIONS

Column Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
LOB_NAME	NOT NULL	VARCHAR2(30)
LOB_PARTITION_NAME		VARCHAR2(30)
SUBPARTITION_NAME		VARCHAR2(30)
LOB_SUBPARTITION_NAME		VARCHAR2(30)
LOB_INDSUBPART_NAME		VARCHAR2(30)
SUBPARTITION_POSITION	NOT NULL	NUMBER
CHUNK		NUMBER
PCTVERSION	NOT NULL	NUMBER
CACHE		VARCHAR2(10)
IN_ROW		VARCHAR2(3)
TABLESPACE_NAME	NOT NULL	VARCHAR2(30)
INITIAL_EXTENT		NUMBER
NEXT_EXTENT		NUMBER
MIN_EXTENTS	NOT NULL	NUMBER
MAX_EXTENTS	NOT NULL	NUMBER
PCT_INCREASE		NUMBER
FREELISTS		NUMBER
FREELIST_GROUPS		NUMBER
LOGGING		VARCHAR2(3)
BUFFER_POOL		VARCHAR2(7)

USER_METHOD_PARAMS

Column Name	Null?	Type
TYPE_NAME	NOT NULL	VARCHAR2(30)
METHOD_NAME	NOT NULL	VARCHAR2(30)
METHOD_NO	NOT NULL	NUMBER
PARAM_NAME	NOT NULL	VARCHAR2(30)
PARAM_NO	NOT NULL	NUMBER
PARAM_MODE		VARCHAR2(6)
PARAM_TYPE_MOD		VARCHAR2(7)
PARAM_TYPE_OWNER		VARCHAR2(30)
PARAM_TYPE_NAME		VARCHAR2(30)
CHARACTER_SET_NAME		VARCHAR2(44)

USER_METHOD_RESULTS

Column Name	Null?	Type
TYPE_NAME	NOT NULL	VARCHAR2(30)
METHOD_NAME	NOT NULL	VARCHAR2(30)
METHOD_NO	NOT NULL	NUMBER
RESULT_TYPE_MOD		VARCHAR2(7)
RESULT_TYPE_OWNER		VARCHAR2(30)
RESULT_TYPE_NAME		VARCHAR2(30)
CHARACTER_SET_NAME		VARCHAR2(44)

USER_NESTED_TABLES

Column Name	Null?	Type
TABLE_NAME		VARCHAR2(30)
TABLE_TYPE_OWNER		VARCHAR2(30)
TABLE_TYPE_NAME		VARCHAR2(30)
PARENT_TABLE_NAME		VARCHAR2(30)
PARENT_TABLE_COLUMN		VARCHAR2(4000)
STORAGE_SPEC		VARCHAR2(30)
RETURN_TYPE		VARCHAR2(20)

USER_OBJECTS

Column Name	Null?	Type
OBJECT_NAME		VARCHAR2(128)
SUBOBJECT_NAME		VARCHAR2(30)
OBJECT_ID		NUMBER
DATA_OBJECT_ID		NUMBER
OBJECT_TYPE		VARCHAR2(18)
CREATED		DATE
LAST_DDL_TIME		DATE
TIMESTAMP		VARCHAR2(19)
STATUS		VARCHAR2(7)
TEMPORARY		VARCHAR2(1)
GENERATED		VARCHAR2(1)
SECONDARY		VARCHAR2(1)

USER_OBJECT_SIZE

Column Name	Null?	Type
NAME	NOT NULL	VARCHAR2(30)
TYPE		VARCHAR2(13)
SOURCE_SIZE		NUMBER
PARSED_SIZE		NUMBER
CODE_SIZE		NUMBER
ERROR_SIZE		NUMBER

USER_OBJECT_TABLES

Column Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
TABLESPACE_NAME		VARCHAR2(30)
CLUSTER_NAME		VARCHAR2(30)
IOT_NAME		VARCHAR2(30)
PCT_FREE		NUMBER
PCT_USED		NUMBER
INI_TRANS		NUMBER
MAX_TRANS		NUMBER

```
INITIAL_EXTENT                                NUMBER
NEXT_EXTENT                                   NUMBER
MIN_EXTENTS                                   NUMBER
MAX_EXTENTS                                   NUMBER
PCT_INCREASE                                  NUMBER
FREELISTS                                     NUMBER
FREELIST_GROUPS                               NUMBER
LOGGING                                       VARCHAR2(3)
BACKED_UP                                     VARCHAR2(1)
NUM_ROWS                                      NUMBER
BLOCKS                                        NUMBER
EMPTY_BLOCKS                                  NUMBER
AVG_SPACE                                     NUMBER
CHAIN_CNT                                     NUMBER
AVG_ROW_LEN                                   NUMBER
AVG_SPACE_FREELIST_BLOCKS                     NUMBER
NUM_FREELIST_BLOCKS                           NUMBER
DEGREE                                        VARCHAR2(10)
INSTANCES                                     VARCHAR2(10)
CACHE                                         VARCHAR2(5)
TABLE_LOCK                                    VARCHAR2(8)
SAMPLE_SIZE                                   NUMBER
LAST_ANALYZED                                 DATE
PARTITIONED                                   VARCHAR2(3)
IOT_TYPE                                      VARCHAR2(12)
OBJECT_ID_TYPE                                VARCHAR2(16)
TABLE_TYPE_OWNER                    NOT NULL  VARCHAR2(30)
TABLE_TYPE                          NOT NULL  VARCHAR2(30)
TEMPORARY                                     VARCHAR2(1)
SECONDARY                                     VARCHAR2(1)
NESTED                                        VARCHAR2(3)
BUFFER_POOL                                   VARCHAR2(7)
ROW_MOVEMENT                                  VARCHAR2(8)
GLOBAL_STATS                                  VARCHAR2(3)
USER_STATS                                    VARCHAR2(3)
DURATION                                      VARCHAR2(15)
SKIP_CORRUPT                                  VARCHAR2(8)
MONITORING                                    VARCHAR2(3)
CLUSTER_OWNER                                 VARCHAR2(30)
```

USER_PART_COL_STATISTICS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                     NOT NULL VARCHAR2(30)
PARTITION_NAME                          VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
NUM_DISTINCT                            NUMBER
LOW_VALUE                               RAW(32)
HIGH_VALUE                              RAW(32)
DENSITY                                 NUMBER
NUM_NULLS                               NUMBER
```

```
NUM_BUCKETS                                NUMBER
SAMPLE_SIZE                                NUMBER
LAST_ANALYZED                              DATE
GLOBAL_STATS                               VARCHAR2(3)
USER_STATS                                 VARCHAR2(3)
AVG_COL_LEN                                NUMBER
```

USER_PART_HISTOGRAMS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                              VARCHAR2(30)
PARTITION_NAME                          VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
BUCKET_NUMBER                           NUMBER
ENDPOINT_VALUE                          NUMBER
ENDPOINT_ACTUAL_VALUE                   VARCHAR2(1000)
```

USER_PART_INDEXES

```
Column Name                    Null?    Type
------------------------------ -------- ----
INDEX_NAME                     NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
PARTITIONING_TYPE                       VARCHAR2(7)
SUBPARTITIONING_TYPE                    VARCHAR2(7)
PARTITION_COUNT                NOT NULL NUMBER
DEF_SUBPARTITION_COUNT                  NUMBER
PARTITIONING_KEY_COUNT         NOT NULL NUMBER
SUBPARTITIONING_KEY_COUNT               NUMBER
LOCALITY                                VARCHAR2(6)
ALIGNMENT                               VARCHAR2(12)
DEF_TABLESPACE_NAME                     VARCHAR2(30)
DEF_PCT_FREE                   NOT NULL NUMBER
DEF_INI_TRANS                  NOT NULL NUMBER
DEF_MAX_TRANS                  NOT NULL NUMBER
DEF_INITIAL_EXTENT                      VARCHAR2(40)
DEF_NEXT_EXTENT                         VARCHAR2(40)
DEF_MIN_EXTENTS                         VARCHAR2(40)
DEF_MAX_EXTENTS                         VARCHAR2(40)
DEF_PCT_INCREASE                        VARCHAR2(40)
DEF_FREELISTS                  NOT NULL NUMBER
DEF_FREELIST_GROUPS            NOT NULL NUMBER
DEF_LOGGING                             VARCHAR2(7)
DEF_BUFFER_POOL                         VARCHAR2(7)
```

USER_PART_KEY_COLUMNS

```
Column Name                    Null?    Type
------------------------------ -------- ----
NAME                                    VARCHAR2(30)
OBJECT_TYPE                             VARCHAR2(11)
COLUMN_NAME                             VARCHAR2(4000)
COLUMN_POSITION                         NUMBER
```

USER_PART_LOBS

Column Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
LOB_NAME	NOT NULL	VARCHAR2(30)
LOB_INDEX_NAME	NOT NULL	VARCHAR2(30)
DEF_CHUNK	NOT NULL	NUMBER
DEF_PCTVERSION	NOT NULL	NUMBER
DEF_CACHE		VARCHAR2(10)
DEF_IN_ROW		VARCHAR2(3)
DEF_TABLESPACE_NAME		VARCHAR2(30)
DEF_INITIAL_EXTENT		VARCHAR2(40)
DEF_NEXT_EXTENT		VARCHAR2(40)
DEF_MIN_EXTENTS		VARCHAR2(40)
DEF_MAX_EXTENTS		VARCHAR2(40)
DEF_PCT_INCREASE		VARCHAR2(40)
DEF_FREELISTS		VARCHAR2(40)
DEF_FREELIST_GROUPS		VARCHAR2(40)
DEF_LOGGING		VARCHAR2(7)
DEF_BUFFER_POOL		VARCHAR2(7)

USER_PART_TABLES

Column Name	Null?	Type
TABLE_NAME		VARCHAR2(30)
PARTITIONING_TYPE		VARCHAR2(7)
SUBPARTITIONING_TYPE		VARCHAR2(7)
PARTITION_COUNT		NUMBER
DEF_SUBPARTITION_COUNT		NUMBER
PARTITIONING_KEY_COUNT		NUMBER
SUBPARTITIONING_KEY_COUNT		NUMBER
DEF_TABLESPACE_NAME		VARCHAR2(30)
DEF_PCT_FREE		NUMBER
DEF_PCT_USED		NUMBER
DEF_INI_TRANS		NUMBER
DEF_MAX_TRANS		NUMBER
DEF_INITIAL_EXTENT		VARCHAR2(40)
DEF_NEXT_EXTENT		VARCHAR2(40)
DEF_MIN_EXTENTS		VARCHAR2(40)
DEF_MAX_EXTENTS		VARCHAR2(40)
DEF_PCT_INCREASE		VARCHAR2(40)
DEF_FREELISTS		NUMBER
DEF_FREELIST_GROUPS		NUMBER
DEF_LOGGING		VARCHAR2(7)
DEF_BUFFER_POOL		VARCHAR2(7)

USER_PASSWORD_LIMITS

```
Column Name                    Null?    Type
------------------------------ -------- ----
RESOURCE_NAME                  NOT NULL VARCHAR2(32)
LIMIT                                   VARCHAR2(40)
```

USER_RESOURCE_LIMITS

```
Column Name                    Null?    Type
------------------------------ -------- ----
RESOURCE_NAME                  NOT NULL VARCHAR2(32)
LIMIT                                   VARCHAR2(40)
```

USER_ROLE_PRIVS

```
Column Name                    Null?    Type
------------------------------ -------- ----
USERNAME                                VARCHAR2(30)
GRANTED_ROLE                            VARCHAR2(30)
ADMIN_OPTION                            VARCHAR2(3)
DEFAULT_ROLE                            VARCHAR2(3)
OS_GRANTED                              VARCHAR2(3)
```

USER_SEGMENTS

```
Column Name                    Null?    Type
------------------------------ -------- ----
SEGMENT_NAME                            VARCHAR2(81)
PARTITION_NAME                          VARCHAR2(30)
SEGMENT_TYPE                            VARCHAR2(18)
TABLESPACE_NAME                         VARCHAR2(30)
BYTES                                   NUMBER
BLOCKS                                  NUMBER
EXTENTS                                 NUMBER
INITIAL_EXTENT                          NUMBER
NEXT_EXTENT                             NUMBER
MIN_EXTENTS                             NUMBER
MAX_EXTENTS                             NUMBER
PCT_INCREASE                            NUMBER
FREELISTS                               NUMBER
FREELIST_GROUPS                         NUMBER
BUFFER_POOL                             VARCHAR2(7)
```

USER_SEQUENCES

```
Column Name                    Null?    Type
------------------------------ -------- ----
SEQUENCE_NAME                  NOT NULL VARCHAR2(30)
MIN_VALUE                               NUMBER
MAX_VALUE                               NUMBER
INCREMENT_BY                   NOT NULL NUMBER
```

```
CYCLE_FLAG                                VARCHAR2(1)
ORDER_FLAG                                VARCHAR2(1)
CACHE_SIZE                        NOT NULL NUMBER
LAST_NUMBER                       NOT NULL NUMBER
```

USER_SOURCE

```
Column Name                    Null?    Type
------------------------------ -------- ----
NAME                                    VARCHAR2(30)
TYPE                                    VARCHAR2(12)
LINE                                    NUMBER
TEXT                                    VARCHAR2(4000)
```

USER_SUBPART_COL_STATISTICS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                     NOT NULL VARCHAR2(30)
SUBPARTITION_NAME                       VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
NUM_DISTINCT                            NUMBER
LOW_VALUE                               RAW(32)
HIGH_VALUE                              RAW(32)
DENSITY                                 NUMBER
NUM_NULLS                               NUMBER
NUM_BUCKETS                             NUMBER
SAMPLE_SIZE                             NUMBER
LAST_ANALYZED                           DATE
GLOBAL_STATS                            VARCHAR2(3)
USER_STATS                              VARCHAR2(3)
AVG_COL_LEN                             NUMBER
```

USER_SUBPART_HISTOGRAMS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                              VARCHAR2(30)
SUBPARTITION_NAME                       VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
BUCKET_NUMBER                           NUMBER
ENDPOINT_VALUE                          NUMBER
ENDPOINT_ACTUAL_VALUE                   VARCHAR2(1000)
```

USER_SUBPART_KEY_COLUMNS

```
Column Name                    Null?    Type
------------------------------ -------- ----
NAME                                    VARCHAR2(30)
OBJECT_TYPE                             VARCHAR2(11)
COLUMN_NAME                             VARCHAR2(4000)
COLUMN_POSITION                         NUMBER
```

USER_SYNONYMS

Column Name	Null?	Type
SYNONYM_NAME	NOT NULL	VARCHAR2(30)
TABLE_OWNER		VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
DB_LINK		VARCHAR2(128)

USER_SYS_PRIVS

Column Name	Null?	Type
USERNAME		VARCHAR2(30)
PRIVILEGE	NOT NULL	VARCHAR2(40)
ADMIN_OPTION		VARCHAR2(3)

USER_TABLES

Column Name	Null?	Type
TABLE_NAME	NOT NULL	VARCHAR2(30)
TABLESPACE_NAME		VARCHAR2(30)
CLUSTER_NAME		VARCHAR2(30)
IOT_NAME		VARCHAR2(30)
PCT_FREE		NUMBER
PCT_USED		NUMBER
INI_TRANS		NUMBER
MAX_TRANS		NUMBER
INITIAL_EXTENT		NUMBER
NEXT_EXTENT		NUMBER
MIN_EXTENTS		NUMBER
MAX_EXTENTS		NUMBER
PCT_INCREASE		NUMBER
FREELISTS		NUMBER
FREELIST_GROUPS		NUMBER
LOGGING		VARCHAR2(3)
BACKED_UP		VARCHAR2(1)
NUM_ROWS		NUMBER
BLOCKS		NUMBER
EMPTY_BLOCKS		NUMBER
AVG_SPACE		NUMBER
CHAIN_CNT		NUMBER
AVG_ROW_LEN		NUMBER
AVG_SPACE_FREELIST_BLOCKS		NUMBER
NUM_FREELIST_BLOCKS		NUMBER
DEGREE		VARCHAR2(10)
INSTANCES		VARCHAR2(10)
CACHE		VARCHAR2(5)
TABLE_LOCK		VARCHAR2(8)
SAMPLE_SIZE		NUMBER
LAST_ANALYZED		DATE
PARTITIONED		VARCHAR2(3)
IOT_TYPE		VARCHAR2(12)
TEMPORARY		VARCHAR2(1)

```
SECONDARY                                 VARCHAR2(1)
NESTED                                    VARCHAR2(3)
BUFFER_POOL                               VARCHAR2(7)
ROW_MOVEMENT                              VARCHAR2(8)
GLOBAL_STATS                              VARCHAR2(3)
USER_STATS                                VARCHAR2(3)
DURATION                                  VARCHAR2(15)
SKIP_CORRUPT                              VARCHAR2(8)
MONITORING                                VARCHAR2(3)
CLUSTER_OWNER                             VARCHAR2(30)
```

USER_TABLESPACES

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLESPACE_NAME                NOT NULL VARCHAR2(30)
INITIAL_EXTENT                          NUMBER
NEXT_EXTENT                             NUMBER
MIN_EXTENTS                    NOT NULL NUMBER
MAX_EXTENTS                             NUMBER
PCT_INCREASE                            NUMBER
MIN_EXTLEN                              NUMBER
STATUS                                  VARCHAR2(9)
CONTENTS                                VARCHAR2(9)
LOGGING                                 VARCHAR2(9)
EXTENT_MANAGEMENT                       VARCHAR2(10)
ALLOCATION_TYPE                         VARCHAR2(9)
```

USER_TAB_COLUMNS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                     NOT NULL VARCHAR2(30)
COLUMN_NAME                    NOT NULL VARCHAR2(30)
DATA_TYPE                               VARCHAR2(106)
DATA_TYPE_MOD                           VARCHAR2(3)
DATA_TYPE_OWNER                         VARCHAR2(30)
DATA_LENGTH                    NOT NULL NUMBER
DATA_PRECISION                          NUMBER
DATA_SCALE                              NUMBER
NULLABLE                                VARCHAR2(1)
COLUMN_ID                      NOT NULL NUMBER
DEFAULT_LENGTH                          NUMBER
DATA_DEFAULT                            LONG
NUM_DISTINCT                            NUMBER
LOW_VALUE                               RAW(32)
HIGH_VALUE                              RAW(32)
DENSITY                                 NUMBER
NUM_NULLS                               NUMBER
NUM_BUCKETS                             NUMBER
LAST_ANALYZED                           DATE
SAMPLE_SIZE                             NUMBER
CHARACTER_SET_NAME                      VARCHAR2(44)
CHAR_COL_DECL_LENGTH                    NUMBER
```

```
GLOBAL_STATS                              VARCHAR2(3)
USER_STATS                                VARCHAR2(3)
AVG_COL_LEN                               NUMBER
```

USER_TAB_COL_STATISTICS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                     NOT NULL VARCHAR2(30)
COLUMN_NAME                    NOT NULL VARCHAR2(30)
NUM_DISTINCT                            NUMBER
LOW_VALUE                               RAW(32)
HIGH_VALUE                              RAW(32)
DENSITY                                 NUMBER
NUM_NULLS                               NUMBER
NUM_BUCKETS                             NUMBER
LAST_ANALYZED                           DATE
SAMPLE_SIZE                             NUMBER
GLOBAL_STATS                            VARCHAR2(3)
USER_STATS                              VARCHAR2(3)
AVG_COL_LEN                             NUMBER
```

USER_TAB_COMMENTS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                     NOT NULL VARCHAR2(30)
TABLE_TYPE                              VARCHAR2(11)
COMMENTS                                VARCHAR2(4000)
```

USER_TAB_HISTOGRAMS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                              VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
ENDPOINT_NUMBER                         NUMBER
ENDPOINT_VALUE                          NUMBER
ENDPOINT_ACTUAL_VALUE                   VARCHAR2(1000)
```

USER_TAB_MODIFICATIONS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                              VARCHAR2(30)
PARTITION_NAME                          VARCHAR2(30)
SUBPARTITION_NAME                       VARCHAR2(30)
INSERTS                                 NUMBER
UPDATES                                 NUMBER
DELETES                                 NUMBER
TIMESTAMP                               DATE
TRUNCATED                               VARCHAR2(3)
```

USER_TAB_PARTITIONS

Column Name	Null?	Type
TABLE_NAME		VARCHAR2(30)
COMPOSITE		VARCHAR2(3)
PARTITION_NAME		VARCHAR2(30)
SUBPARTITION_COUNT		NUMBER
HIGH_VALUE		LONG
HIGH_VALUE_LENGTH		NUMBER
PARTITION_POSITION		NUMBER
TABLESPACE_NAME		VARCHAR2(30)
PCT_FREE		NUMBER
PCT_USED		NUMBER
INI_TRANS		NUMBER
MAX_TRANS		NUMBER
INITIAL_EXTENT		NUMBER
NEXT_EXTENT		NUMBER
MIN_EXTENT		NUMBER
MAX_EXTENT		NUMBER
PCT_INCREASE		NUMBER
FREELISTS		NUMBER
FREELIST_GROUPS		NUMBER
LOGGING		VARCHAR2(7)
NUM_ROWS		NUMBER
BLOCKS		NUMBER
EMPTY_BLOCKS		NUMBER
AVG_SPACE		NUMBER
CHAIN_CNT		NUMBER
AVG_ROW_LEN		NUMBER
SAMPLE_SIZE		NUMBER
LAST_ANALYZED		DATE
BUFFER_POOL		VARCHAR2(7)
GLOBAL_STATS		VARCHAR2(3)
USER_STATS		VARCHAR2(3)

USER_TAB_PRIVS

Column Name	Null?	Type
GRANTEE	NOT NULL	VARCHAR2(30)
OWNER	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)
GRANTOR	NOT NULL	VARCHAR2(30)
PRIVILEGE	NOT NULL	VARCHAR2(40)
GRANTABLE		VARCHAR2(3)

USER_TAB_PRIVS_MADE

Column Name	Null?	Type
GRANTEE	NOT NULL	VARCHAR2(30)
TABLE_NAME	NOT NULL	VARCHAR2(30)

```
GRANTOR                        NOT NULL VARCHAR2(30)
PRIVILEGE                      NOT NULL VARCHAR2(40)
GRANTABLE                               VARCHAR2(3)
```

USER_TAB_PRIVS_RECD

```
Column Name                    Null?    Type
------------------------------ -------- ----
OWNER                          NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
GRANTOR                        NOT NULL VARCHAR2(30)
PRIVILEGE                      NOT NULL VARCHAR2(40)
GRANTABLE                               VARCHAR2(3)
```

USER_TAB_SUBPARTITIONS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                     NOT NULL VARCHAR2(30)
PARTITION_NAME                          VARCHAR2(30)
SUBPARTITION_NAME                       VARCHAR2(30)
SUBPARTITION_POSITION          NOT NULL NUMBER
TABLESPACE_NAME                NOT NULL VARCHAR2(30)
PCT_FREE                       NOT NULL NUMBER
PCT_USED                       NOT NULL NUMBER
INI_TRANS                      NOT NULL NUMBER
MAX_TRANS                      NOT NULL NUMBER
INITIAL_EXTENT                          NUMBER
NEXT_EXTENT                             NUMBER
MIN_EXTENT                     NOT NULL NUMBER
MAX_EXTENT                     NOT NULL NUMBER
PCT_INCREASE                            NUMBER
FREELISTS                               NUMBER
FREELIST_GROUPS                         NUMBER
LOGGING                                 VARCHAR2(3)
NUM_ROWS                                NUMBER
BLOCKS                                  NUMBER
EMPTY_BLOCKS                            NUMBER
AVG_SPACE                               NUMBER
CHAIN_CNT                               NUMBER
AVG_ROW_LEN                             NUMBER
SAMPLE_SIZE                             NUMBER
LAST_ANALYZED                           DATE
BUFFER_POOL                             VARCHAR2(7)
GLOBAL_STATS                            VARCHAR2(3)
USER_STATS                              VARCHAR2(3)
```

USER_TRIGGERS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TRIGGER_NAME                            VARCHAR2(30)
TRIGGER_TYPE                            VARCHAR2(16)
TRIGGERING_EVENT                        VARCHAR2(216)
```

```
TABLE_OWNER                                VARCHAR2(30)
BASE_OBJECT_TYPE                           VARCHAR2(16)
TABLE_NAME                                 VARCHAR2(30)
COLUMN_NAME                                VARCHAR2(4000)
REFERENCING_NAMES                          VARCHAR2(128)
WHEN_CLAUSE                                VARCHAR2(4000)
STATUS                                     VARCHAR2(8)
DESCRIPTION                                VARCHAR2(4000)
ACTION_TYPE                                VARCHAR2(11)
TRIGGER_BODY                               LONG
```

USER_TRIGGER_COLS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TRIGGER_OWNER                           VARCHAR2(30)
TRIGGER_NAME                            VARCHAR2(30)
TABLE_OWNER                             VARCHAR2(30)
TABLE_NAME                              VARCHAR2(30)
COLUMN_NAME                             VARCHAR2(4000)
COLUMN_LIST                             VARCHAR2(3)
COLUMN_USAGE                            VARCHAR2(17)
```

USER_TS_QUOTAS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLESPACE_NAME                NOT NULL VARCHAR2(30)
BYTES                                   NUMBER
MAX_BYTES                               NUMBER
BLOCKS                         NOT NULL NUMBER
MAX_BLOCKS                              NUMBER
```

USER_TYPES

```
Column Name                    Null?    Type
------------------------------ -------- ----
TYPE_NAME                      NOT NULL VARCHAR2(30)
TYPE_OID                       NOT NULL RAW(16)
TYPECODE                                VARCHAR2(30)
ATTRIBUTES                              NUMBER
METHODS                                 NUMBER
PREDEFINED                              VARCHAR2(3)
INCOMPLETE                              VARCHAR2(3)
```

USER_TYPE_ATTRS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TYPE_NAME                      NOT NULL VARCHAR2(30)
ATTR_NAME                      NOT NULL VARCHAR2(30)
ATTR_TYPE_MOD                           VARCHAR2(7)
ATTR_TYPE_OWNER                         VARCHAR2(30)
ATTR_TYPE_NAME                          VARCHAR2(30)
```

```
LENGTH                                    NUMBER
PRECISION                                 NUMBER
SCALE                                     NUMBER
CHARACTER_SET_NAME                        VARCHAR2(44)
ATTR_NO                          NOT NULL NUMBER
```

USER_TYPE_METHODS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TYPE_NAME                      NOT NULL VARCHAR2(30)
METHOD_NAME                    NOT NULL VARCHAR2(30)
METHOD_NO                      NOT NULL NUMBER
METHOD_TYPE                             VARCHAR2(6)
PARAMETERS                     NOT NULL NUMBER
RESULTS                        NOT NULL NUMBER
```

USER_UNUSED_COL_TABS

```
Column Name                    Null?    Type
------------------------------ -------- ----
TABLE_NAME                     NOT NULL VARCHAR2(30)
COUNT                                   NUMBER
```

USER_UPDATABLE_COLUMNS

```
Column Name                    Null?    Type
------------------------------ -------- ----
OWNER                          NOT NULL VARCHAR2(30)
TABLE_NAME                     NOT NULL VARCHAR2(30)
COLUMN_NAME                    NOT NULL VARCHAR2(30)
UPDATABLE                               VARCHAR2(3)
INSERTABLE                              VARCHAR2(3)
DELETABLE                               VARCHAR2(3)
```

USER_USERS

```
Column Name                    Null?    Type
------------------------------ -------- ----
USERNAME                       NOT NULL VARCHAR2(30)
USER_ID                        NOT NULL NUMBER
ACCOUNT_STATUS                 NOT NULL VARCHAR2(32)
LOCK_DATE                               DATE
EXPIRY_DATE                             DATE
DEFAULT_TABLESPACE             NOT NULL VARCHAR2(30)
TEMPORARY_TABLESPACE           NOT NULL VARCHAR2(30)
CREATED                        NOT NULL DATE
INITIAL_RSRC_CONSUMER_GROUP             VARCHAR2(30)
EXTERNAL_NAME                           VARCHAR2(4000)
```

USER_VARRAYS

Column Name	Null?	Type
PARENT_TABLE_NAME		VARCHAR2(30)
PARENT_TABLE_COLUMN		VARCHAR2(4000)
TYPE_OWNER		VARCHAR2(30)
TYPE_NAME		VARCHAR2(30)
LOB_NAME		VARCHAR2(30)
STORAGE_SPEC		VARCHAR2(30)
RETURN_TYPE		VARCHAR2(20)

USER_VIEWS

Column Name	Null?	Type
VIEW_NAME	NOT NULL	VARCHAR2(30)
TEXT_LENGTH		NUMBER
TEXT		LONG
TYPE_TEXT_LENGTH		NUMBER
TYPE_TEXT		VARCHAR2(4000)
OID_TEXT_LENGTH		NUMBER
OID_TEXT		VARCHAR2(4000)
VIEW_TYPE_OWNER		VARCHAR2(30)
VIEW_TYPE		VARCHAR2(30)

✦ ✦ ✦

Suggested Readings, Web Sites, and Other Resources

This appendix provides information on books, Web sites, and other resources, such as periodicals, that can be used to better prepare yourself for the "Introduction to Oracle: SQL & PL/SQL" exam. Although you are not expected or required to read all the books and visit every Web site, working with Oracle and being completely familiar with the contents of this book are necessary to prepare you for taking the exam.

Suggested Readings

Additional research and preparation before taking the "Introduction to Oracle: SQL & PL/SQL" exam can always be beneficial. The books listed here provide additional information on relational database management systems in general, as well as Oracle and PL/SQL.

Books

Christopher Allen, *Oracle PL/SQL 101* (McGraw-Hill, 2000).

E.F. Codd, *The Relational Model for Database Management Version 2* (Addison-Wesley, 1999).

C.J. Date, *The Database Relational Model: A Retrospective Review and Analysis* (Addison-Wesley, 2001).

C.J. Date, *An Introduction to Database Systems, 7th Edition* (Addison-Wesley, 1999).

Gary Dodge et al., *Essential Oracle8i Data Warehousing: Designing, Building, and Managing Oracle Data Warehouses* (John Wiley & Sons, 2000).

David Ensor and Tim Stevenson, *Oracle Design* (O'Reilly & Associates, 1997).

Steven Feuerstein, *Oracle PL/SQL Programming, 2nd Edition* (O'Reilly & Associates, 1997).

Jonathan Gennick, et al., *Oracle SQL*Plus: The Definitive Guide* (O'Reilly & Associates, 1999).

Michael J. Hernandez, *Database Design for Mere Mortals : A Hands-On Guide to Relational Database Design* (Addison-Wesley, 1997).

Ralph Kimball et al., *The Data Warehouse Lifecycle Toolkit : Expert Methods for Designing, Developing, and Deploying Data Warehouses* (John Wiley & Sons, 1998).

David C. Kreines and Ken Jacobs, *Oracle SQL: The Essential Reference* (O'Reilly & Associates, 2000).

Oracle documentation manuals

Oracle8i Administrators Guide

Oracle8i Concepts

Oracle8i SQL Reference

*SQL*Plus User's Guide and Reference*

Web Sites

The Web sites in Table G-1 are useful in acquiring Oracle software and getting information about Oracle and the Oracle Certified Professional (OCP) program.

Table G-1
Suggested Sites

URL	***Description***
http://www.oracle.com	Oracle Corporation home page.
http://education.oracle.com	Oracle Education home page. Includes information on the Oracle Certified Professional program, as well as Oracle course offerings.
http://technet.oracle.com	Oracle Technology Network home page. Provides information on Oracle products, as well as copies of Oracle database and other tools. Membership is free.
http://technet.oracle.com/docs/content.html	Oracle documentation on TechNet. This URL provides links to all the Oracle documentation sets for Oracle 8*i*—in case you don't have the hard copies or CD handy.
http://www.certcities.com	CertCities home page. This site offers information on certifications, as well as columns and feature articles on Oracle and the Oracle Certified Professional (OCP) program.
http://www.orafaq.com/faq.htm	Underground Oracle FAQ. This site provides links and information on Oracle products and features.

✦ ✦ ✦

Index

Numbers & Symbols

& (ampersand), 172–175, 231, 233–235
* (asterisk), 41, 413, 540
\ (backslash), 290
: (colon), 422, 430, 593
, (comma), 41, 49, 50, 242, 248, 274
- (dash), 413
. (decimal point), 93, 106
$ (dollar sign), 94, 95, 248, 270, 414
“ (double quotes), 50, 414
/ (forward slash), 230, 253, 413, 576
- (hyphen), 236
( ) (parentheses), 44, 45, 47, 298, 202, 512
% (percent sign), 69–70
. (period), 248, 505
| (pipe symbol), 45, 249
(pound sign), 270, 414
; (semicolon), 253, 409, 410, 446, 453, 536
‘ (single quote), 46, 59, 69, 173, 193, 196, 290, 411
_ (underscore), 69, 270, 414
@ command, 239, 254

A

ACCEPT command, 175, 235–236
ACCESS_INTO_NULL exception, 542
ACID test, 206–207
ACROBAT folder, 618
Acrobat Reader, 615, 618
active sets, 489, 491–493
add (+) operator, 42, 43, 44–45
ADD_MONTHS function, 108
Address1 column, 198, 670, 671
Address2 column, 670, 671
AddressType datatype, 19
ADMIN directory, 306
Adobe Acrobat Reader, 615, 618
aggregate functions, 482
  Aggregate BY clause and, 119–122
  assessment questions, 126–128, 132–133
  basic description of, 114–119
  DISTINCT keyword and, 118
  HAVING clause and, 122–123
  identifying available, 119
  lab exercises, 129–131, 135–137
  pre-test, 90, 131–132
  scenarios, 129, 133–135
  using the WHERE clause with, 119
alias(es)
  adding, 47–48
  basic description of, 48
  column, 47–48
  comparison operators and, 62–63
  errors and, 62–63
  joins and, 144, 150
  naming, 50
  in the ORDER BY statement, 63
  PL/SQL and, 427
ALL keyword, 372
ALL operator, 153–155
Allen, Christopher, 717
ALL_views, 22, 280–283. *See also* ALL_views (listed by name)
  comments and, 290
  querying, 290
  synonyms and, 335–336
ALL_views (listed by name). *See also* ALL_views
  ALL_CATALOG view, 283
  ALL_COL_COMMENTS view, 290
  ALL_CONSTRAINTS view, 307–308
  ALL_SEQUENCES view, 331–332
  ALL_SYNONYMS view, 336
  ALL_TAB_COMMENTS view, 290
ALTER ANY TABLE privilege, 367
ALTER privilege, 373
ALTER SEQUENCE command, 332–333
ALTER TABLE command, 210, 283–287, 293, 301, 303–304, 306–307
ALTER TRIGGER command, 599–600
ALTER USER command, 365–366
ALTER USER privilege, 367
ALTER VIEW command, 373
American National Standards Institute (ANSI), 11, 37, 58, 162, 282
American Standard Code for Information Interchange (ASCII), 117, 238, 674
ampersand (&), 172–175, 231, 233–235
AND operator, 63–64, 143–144, 147, 172, 418–420
annual_sales column, 147
anonymous blocks. *See* blocks
ANSI. *See* American National Standards Institute (ANSI)
anti-virus software, 620
ANY operator, 153–155
APPEND option, 238
Aria ZIM, 615, 618
ARIAZIM folder, 618
arithmetic operations, 41–52, 411
arrays, 500, 508
ARRAYSIZE option, 241
AS keyword, 49
AS operator, 49
AS SELECT clause, 312

ASCII. *See* American Standard Code for Information Interchange (ASCII)
assignment operators, 411, 414, 415
asterisk (*), 41, 413, 540
@ command, 239, 254
atomicity, 206
ATTRIBUTE column, 255
attributes. *See also* attributes (listed by name)
  basic description of, 426
  cursor, 487–488
  %TYPE, 421–422
attributes (listed by name). *See also* attributes
  %FOUND attribute, 487, 493, 497
  %ISOPEN attribute, 487, 493, 497
  %NOTFOUND attribute, 487, 488, 493, 497, 498
  %ROWCOUNT attribute, 463, 487, 493, 497
  %ROWTYPE attribute, 424, 494, 504–505, 511
  %TYPE attribute, 421–424, 482, 504, 507, 513
auditing, 599
AUTOPRINT environment variable, 430
AVG function, 114–115, 120, 161, 251, 582

B

backslash (\), 290
BACKUP ANY TABLE privilege, 367
backup copies, of databases, 365
base
  datatypes, 416
  tables, 22–23, 309–310
Batch Jobs table, 673
BEGIN keyword, 411
BEGIN section, 512, 514
BETWEEN operator, 147
BETWEEN...AND operator, 66, 67–68
BFILE datatype, 15, 275
binary format, 5
BINARY_INTEGER datatype, 416, 426, 506
bind variables, 236, 414, 422, 430, 482, 577
BirthDate column, 284
bitmap indexes, 319. *See also* index(es)
BLOB datatype, 10, 15, 116, 275, 417
block(s)
  basic description of, 407, 573
  basic loops and, 445
  bind variables and, 236, 422
  declare section of, 410–411
  error handling and, 535, 539, 540, 542, 548–549
  exception section of, 412
  executable section of, 411
  executing, 411, 429–421
  loading, into the Database Buffer Cache, 204
  named, 408–409
  nested, 459–461
  parsing, 429
  stored programs and, 576
  structure of, 409–412
  testing, 429–421
Bonus table, 146
book_class procedure, 429
Boolean datatype, 415, 418–420, 456–458, 555
BREAK command, 250–252
BTITLE command, 244, 246
buffer(s), 229–230
  caches, 204
  commands stored in, executing, 230
  contents, saving, 237–239
  editing, 230, 237–239, 241
  redo, 204–205
  writing lines of text to, 430–431
bugs, basic description of, 537. *See also* error(s); exception(s)
business rules, 292

C

C (high-level language), 577
C++ (high-level language), 69, 71
CACHE parameter, 328
caches, 203–204, 328, 331, 332. *See also* memory
capitalization, 50–51, 97, 675. *See also* case-sensitivity
cardinality columns, high, 320–321
carriage returns, 39
Cartesian Product. *See* cross-joins
CASCADE clause, 305
CASCADE CONSTRAINTS option, 288, 377
CASCADE keyword, 303
CASCADE option, 364–365
CASE statement, 457
case-conversion functions, 99–101
case-sensitivity, 39–40, 59, 70–71, 105, 412. *See also* capitalization
CD-ROM (*Oracle SQL and PS/SQL Certification Bible*)
  applications on, list of, 618–620
  contents of, basic description of, 615–620
  copying scripts from, 674, 675–676
  installing, 617
  problems with, troubleshooting, 620
  system requirements for, 616
  technical support for, 620
CellPhone column, 670
CE[NTER] option, 246
CertCities Web site, 719
CERTDB folder, 674, 676
CERTDB tablespace, 674
CERTDBOBJ.SQL, 674, 676–681
Change column, 674
ChangedBy column, 674, 301–303
“change_on_install” password, 361, 362
ChangePrice script, 464
CHAR datatype, 235, 285
character(s). *See also* character datatype; character functions
  controlling the number of, 243–244
  leading/trailing, removing, 103–104
  -manipulation functions, 101–105
  padding, 93
  values, how Oracle determines, 117
  wildcard, 69–70, 255

character datatype, 46–47, 249. *See also* character(s)
PL/SQL and, 415, 417–418, 445
using comparison operators with, 59–62
character functions. *See also* character(s)
basic description of, 99–110
case-conversion functions, 99–101
character-manipulation functions, 101–102
CHAR(n) datatype, 417
CHAR(size) datatype, 14, 15, 275
CHAR_VALUE column, 255
CHECK constraint, 16, 292, 299–308
Checkpoint process, 203
City column, 670, 671
ClassEnrollment table, 164, 196, 277, 294–295, 300, 335, 374–375, 379, 502, 546, 673
Classes table, 291
ClassID column, 295, 297, 672, 673
classid_seq sequence, 485–486
class_maximum variable, 584, 587
ClassRoomNumber column, 672
CLEAR BREAKS command, 251
CLEAR COMPUTES command, 252
CLE[AR] option, 248–249
client(s). *See also* server(s)
programs, abnormal termination of, 210
-server environments, 406–407, 619
-side procedures, 575
CLOB datatype, 15, 116, 275, 417
CLOSE command, 491, 498
COBOL, 577
Codd, E. F., 6, 7, 717
collection datatypes, 504–518
collection methods, 512, 515–518
COLLECTION_IS_NULL exception, 542
colon (:), 422, 430, 593
COLSEP option, 241
column(s). *See also* pseudo-columns
adding, 284–285
aliases, 47–48
basic description of, 13–14
cardinality, 320–321
concatenating, 45–46
constraints (in-line constraints), 293–294
definitions, 274
deleting, 286–287
documenting, 289–291
dropping, 286–287
formatting, 247–249
level, granting object privileges at, 374
lists, 195–196, 249, 485
lowest and highest values in, returning, 116–117
modifying, 285–286
names, 278, 279, 483
ownership of, specifying, 272
in the PRODUCT_USER_PROFILE table, 255
properties of, 13–14
UNUSED, marking, 286, 287
used in join conditions, 320
viewing all, 41
COLUMN command, 247–249, 253
Colx variable, 174
comma (,), 41, 49, 50, 242, 248, 274
comma-delimited files, 242
command(s). *See also specific commands*
disabling, 255
files, executing the contents of, 239
lists, displaying, 241
rules/conventions for, 39–40
stored in buffers, re-executing, 230
terminators, 253
comments, 69, 289–290, 413
Comments column, 672, 673
COMMIT statement, 205, 207–210, 462, 484, 502
sequences and, 331
triggers and, 595
comparison operators, 66–72, 146, 147
subqueries and, 152, 155
used with character and date data, 59–62
used with expressions, 62–63
WHERE clause and, 58–63
compiler(s). *See also* compiling
comments and, 413
directives, 545
PL/SQL and, 407, 413
processing of blocks by, 407
compiling. *See also* compiler(s)
basic description of, 407
errors, 535–537, 545, 574
subprograms and, 408
composite datatypes, 423–426, 504–518
COMPUTE command, 251–252
CONCAT function, 101, 102–103
concatenation (||) operator, 91–92, 102, 411
basic description of, 45–46
NULL values and, 52–53
conditional processing, 455–458
CONNECT BY PRIOR clause, 172
CONNECT command, 209, 363
CONNECT PRIOR statement, 169
connect time, maximum, 365
consistency, 206
constraint(s)
adding, to tables, 301–302
basic description of, 15–17, 291–292
conditions, finding rows not meeting, 306
creating, 373
data dictionary views and, 22
data integrity using, 291–308
defining, 293–294
disabling, 304–306
dropping, 302–303
enabling, 304–307
exceptions and, 546
indexes and, 319, 322, 324
in-line, 293–294
managing, 300–308
naming, 293
out-of-line, 293–294

Continued

constraint(s) *(continued)*
referential integrity, 546
sequences and, 325, 330–331
types of, 16–17, 295–296
using, instead of repeating values, 415
viewing information about, 307–308
CONSTRAINT clause, 312
CONSTRAINT keyword, 294
constructor methods, 512, 514
Contact column, 671
control structures, 406
conventions
for indexes, 320–321
for keywords, 39–40
for objects, 269–272
for SQL statements, 39–40
conversion functions, 91–99, 197
correlated subqueries, 159–161
correlation names, 591–592
COU[NT] function, 251
COUNT(*) function, 115–116, 117
COUNT(column) function, 114, 115–116
COUNT method, 515, 516
counters, 448
Country column, 144, 670, 671, 672
CourseAudit table, 301–302, 674
CourseEnrollment table, 200
CourseID column, 6–9
CourseName column, 374, 670
CourseNumber column, 295, 299–300, 331, 670, 672, 674
Courses table, 6–8, 9, 13, 41, 199, 204, 290, 295, 372, 374, 424, 598, 670
crashes, 205, 209, 326, 331
CREATE ANY INDEX privilege, 323
CREATE INDEX command, 319
CREATE option, 238
CREATE PACKAGE BODY command, 584
CREATE PROCEDURE privilege, 367
CREATE PUBLIC SYNONYM permission, 335, 367
CREATE ROLL command, 378–379
CREATE SEQUENCE command, 326–328
CREATE SEQUENCE privilege, 367
CREATE SESSION privilege, 364, 367–369
CREATE statement, 269
CREATE SYNONYM privilege, 367
CREATE TABLE privilege, 367
CREATE TABLE statement, 273–283
CREATE TRIGGER command, 588
CREATE USER command, 363–364
CREATE USER privilege, 367
CREATE VIEW privilege, 367
CREATE VIEW statement, 157, 310–311
CREATED column, 603
CREATEUSER.SQL, 674, 676–677
cross-joins, 145–146. *See also* join(s)
currency settings, 94
CURRVAL pseudo-column, 300, 329–330
cursor(s), 427–428, 463
attributes, 487–488, 493
basic description of, 487
closing, 491, 498
declaring, 489–490
disabling, 491
fetching from, 492–494
FOR loop, 495–497, 499–500
FOR UPDATE option and, 501–503
opening, 490–491, 501
parameters, 500–501
state information, storage of, in the PGA, 203
subqueries and, 499–500
values returned by, declaring records that will hold, 494
variables, 497–499
CURSOR_ALREADY_OPEN exception, 543
CustomerID column, 271
Customers table, 271
CYCLE option, 325–326
CYCLE/NOCYCLE parameter, 328

D

dash (-), 413
Data Control Language (DCL), 407, 462
basic description of, 9, 37–38
INSERT statement and, 484
transactions and, 209
triggers and, 589, 599
Data Definition Language (DDL), 407, 462, 481
basic description of, 8, 37, 38
creating objects and, 269
DROP TABLE command and, 289
DROP USER command and, 364–365
INSERT statement and, 484
security and, 364–365
transactions and, 209
triggers and, 589, 599
TRUNCATE TABLE command and, 289
data dictionaries. *See also* data dictionary views
basic description of, 21–23
caches for, 203–204
getting information on tables from, 279–283
reconciling object names in, 144
removing table definitions from, 288–289
data dictionary views, 22, 600–603, 687–715. *See also* views (listed by name)
describing roles, 380–381
for object privileges, 375–376
Data Manipulation Language (DML), 406–408, 481
adding data with, 193–198
basic description of, 8, 37, 38–40
controlling transactions with, 205–211
FEEDBACK option and, 242–243
indexes and, 322
modifying existing data with, 198–199
statements, how Oracle processes, 203–205
triggers and, 596

Data Segment, 204
Data Warehouse Lifecycle Toolkit, The: Expert Methods for Designing, Developing, and Deploying Data Warehouses (Kimball, et al.), 273, 718
Database Buffer Cache, 204
Database Design for Mere Mortals: A Hands-On Guide to Relational Database Design (Hernandez), 273, 718
database management system (DBMS), 6
database objects. *See* object(s)
Database Relational Model, The: A Retrospective Review and Analysis (Date), 7, 717
datatype(s). *See also specific datatypes*
  base, 416
  basic description of, 13–14, 274–276
  conversion, 420–421, 445
  PL/SQL and, 418–427
  sub-, 416
date(s). *See also* date formats
  determining the number of months between, 108–109
  of errors, 556
  expiration, for passwords, 23
  functions, 107–110
  returning current, from the system, 112
  returning new, 108
  returning the last, of a month, 109–110
  returning the next occurrence of, 109
  rounding to the nearest, 110
  truncating to the nearest, 110
  values, adding a number of months to, 108
Date, C. J., 7, 717–718
DATE datatype, 14, 235, 415, 417–418
date formats, 96–99, 194–197, 420–421. *See also* date(s)
  ACCEPT command and, 235–236
  arithmetic operations and, 43–44
  NLS (National Language Support), 61, 96–98, 194, 197
  PL/SQL and, 411, 415, 417–418
  TO_CHAR function and, 96–98
  using comparison operators with, 59–62
DateChanged column, 674
DATE_VALUE column, 255
DaysDuration column, 43, 672
DB Writer (DBWR), 203
DB2 (IBM), 619
DBA_views, 22, 280, 335–336
DBMS. *See* database management system (DBMS)
DBMS_OUTPUT package, 430–431, 446
DBMS_OUTPUT.PUT_LINE procedure, 445, 537
DBMS_STANDARDS package, 553
DBSETUP folder, 674, 676
DBWR. *See* DB Writer (DBWR)
DCL. *See* Data Control Language (DCL)
DDL. *See* Data Definition Language (DDL)
debugging. *See also* error(s)
  basic description of, 537
  modularity and, 406
  program logic errors, 537
  stored programs and, 575
DEC datatype, 416
DECIMAL datatype, 416
decimal point (.), 93, 106
DECLARE keyword, 409–411, 540
DECLARE section, 512, 514
DECODE function, 112–113
DEFAULT clause, 277–278, 283, 315
  adding, to a column, 285
  sequences and, 330
DEFAULT keyword, 414, 415, 423
DEFINE command, 234–235
DELETE ANY TABLE privilege, 368
DELETE method, 516
DELETE privilege, 373
DELETE statement, 199–205, 208–209, 487–488
  database objects and, 289, 310, 314, 317, 318
  preventing the execution of, 255
  server processes and, 204
  subqueries in, 202–203
  triggers and, 588, 589, 590, 594, 595
  views and, 310, 314, 317
  WHERE CURRENT OF clause and, 503
DELETE(row) method, 516
DELETE(start_row,end_row) method, 516
deleting. *See also* DELETE statement
  columns, 286–287
  with DML statements, 199–205
  procedures, 580
  rows, 199–205, 289, 487
  triggers, 600
  using subqueries, 200–203
  variables, 234, 235
delimited identifiers, 50
demo (demonstration) software, 620
DESC[RIBE] command, 99, 194, 237, 282, 284, 287, 313, 588
Description column, 670, 671
DISCONNECT command, 209
DISTINCT keyword, 53–55, 118, 315
divide (/) operator, 42, 43, 44–45
DML. *See* Data Manipulation Language (DML)
Dodge, Gary, 718
dollar sign ($), 94, 95, 248, 270, 414
dot notation, 426, 453
double quotes (“), 50, 414
DROP ANY TABLE privilege, 288, 367
DROP COLUMN command, 286, 287
DROP INDEX command, 325
DROP PACKAGE command, 587
DROP PROCEDURE command, 580
DROP PUBLIC SYNONYM permission, 336–337

DROP SEQUENCE command, 333
DROP statement, 269
DROP SYNONYM command, 336–337
DROP TABLE command, 210, 288–289
DROP TRIGGER command, 600
DROP USER command, 364–365
DROP VIEW command, 318
DUAL table, 99
DUPLICATES option, 251
DUP_VAL_ON_INDEX exception, 543
durability, 206–207
dynamic cursors, 497. *See also* cursor(s)

E

e_child_found exception, 544
EDIT command, 40, 230, 238, 252–254
ELSE clause, 456–457, 542
ELSIF conditions, 457–458
EMail column, 302, 670, 672
Embarcadero Technologies, 615, 618
Employees table, 18, 309
END IF statement, 457, 458
END keyword, 536
END LOOP statement, 446, 453
EnrollmentDate column, 277, 673
enrollmenthistory.sql, 164
Ensor, David, 273, 718
entity sets, 13
environment
  settings, 252
  variables, 430–431, 445
= (equal sign), 146, 202, 411
equal to (=) operator, 58–59
equijoins. *See also* join(s)
  basic description of, 142–143
  non-, 146–148
  working with, 142–145
error(s). *See also* debugging; error handling; exception(s)
  ARRAYSIZE option and, 241
  coding conventions and, 555–556
  compile, 535–537, 545, 574
  constraints and, 306
  database objects and, 270–272, 306, 316–318, 329–330, 335
  date of, 556
  DML and, 193, 195–196, 201–202, 208
  forward slashes and, 253
  INSERT statement and, 195–196, 201
  joins and, 143
  logging, 552
  messages, 536, 556
  missing semicolons and, 536
  multi-row functions and, 119–122
  numbers, 542–543, 545, 547, 548, 555
  PL/SQL and, 406, 412, 450–451
  program logic, 537–539
  propagation, 549–555
  security and, 364, 369, 377
  sequences and, 329–330
  single-row functions and, 93, 98, 99, 107, 111, 112
  SQL and, 39, 41, 50, 59, 61, 62
  stored programs and, 574, 589–591, 598
  subqueries and, 155, 159, 202
  substitution variables and, 173
  synonyms and, 335
  syntax, 39
  tables, 548–549, 556
  trapping, 545, 548–549, 550
  types of, 535–538
  views and, 316, 317
error handling, 406, 409, 412. *See also* error(s); exception(s)
  assessment questions, 558–563, 566–567
  lab exercises, 563–565, 567–570
  PL/SQL and, 533–750
  pre-test, 534, 565–566
  scenarios, 563, 567
Errors table, 549
ER/Studio, 615, 618–619
ERSTUDIO folder, 618
Essential Oracle8i Data Warehousing: Designing, Building, and Managing Oracle Data Warehouses (Dodge, et al.), 718
evaluation versions, of software, 620
event triggers, 588, 589, 598–599. *See also* trigger(s)
exams
  practice, 615, 618, 622–652
  preparing for, 665–667
  registering for, 666
  retaking, 667
  tips for taking, 666–667
Excel (Microsoft), 242
exception(s), 414, 489. *See also* debugging; error(s)
  assessment questions, 558–563, 566–567
  basic description of, 539
  coding conventions and, 555–556
  cursors and, 491
  declaring, 544
  handling, 412, 539–547
  implicit cursors and, 481–483
  lab exercises, 563–565, 567–570
  names, 546
  non-predefined, 540, 544–546
  NOTWAIT option and, 502
  predefined, 540, 541–544
  pre-test, 534, 565–566
  propagation, in nested blocks, 549–555
  scenarios, 563, 567
  unhandled, 539–540, 542, 553–554
  user-defined, 540, 546–547
  variable names and, 483
EXCEPTION keyword, 409, 539
Exceptions table, 306
EXECUTE command, 576
EXECUTE privilege, 373

execution
of blocks, 429–421
of commands stored in buffers, 230
of the contents of command files, 239
controlling, 443–447
halting, at a specified line, 537
order of statements, 123
phase, 204
privileges, 373
of the UPDATE statement, preventing, 255
EXISTS method, 515
EXISTS operator, 160–161
EXIT statement, 446, 447, 454–455
EXIT WHEN statement, 449
explicit cursors. *See also* cursor(s)
basic description of, 489–503
closing, 491
declaring, 489–490
fetching from, 492–493
opening, 490–491
using, to populate arrays, 508
expressions
PL/SQL and, 411–412, 456
using comparison operators with, 62–63
EXTEND method, 425, 512, 514, 516
EXTEND(n) method, 516
EXTEND(n,m) method, 516

F

Fax column, 671
FEEDBACK option, 242–243
FETCH command, 489, 491, 492–493
fetch phase, 205
Feuerstein, Steven, 718
file(s). *See also* .sql files
cabinets, as simple databases, 5–6
comma-delimited, 242
command, executing the contents of, 239
-name extensions, 237–238, 240, 253
saving buffer contents in, 237–239
sending screen output to, 239–240
synchronization of, 203
FIRST method, 515
FirstName column, 59, 199, 296, 670–671
fixed-length character datatype, 93
FLOAT datatype, 416
FLOATING POINT datatype, 416
folders
adding/creating, 245–247, 674
naming, 675
FOR loop, 447–451, 454, 495–497, 499–500
FOR UPDATE option, 501–503
FORCE clause, 311
foreign key(s). *See also* FOREIGN KEY constraint
basic description of, 8
DELETE command and, 200
dropping tables and, 288
self-referential keys and, 149
TRUNCATE TABLE command and, 289
FOREIGN KEY constraint, 16, 288–303, 373, 377, 546. *See also* foreign key(s)
dropping, 302–303
disabling, 305
indexes and, 320, 322
viewing information about, 307–308
foreign languages. *See* multilingual environments
FOR[MAT] option, 246, 248
formatting
columns, 247–249
footers, 245–247
headers, 245–247
reports, 245–252
forward slash (/), 230, 253, 413, 576
%FOUND attribute, 487, 493, 497
freeware programs, 620
FROM clause, 40–41, 57, 59, 99
advanced SELECT statements and, 141, 143–145, 157, 158, 159
implicit cursors and, 481
positioning of the INTO clause relative to, 482
SET operators and, 162
subqueries and, 157, 158, 159, 499–500
full_name function, 582
fully qualified object names, 270–272, 362
function(s). *See also* functions (listed by name)
-based indexes, 322–323
basic description of, 408
calling, 581–582
case-conversion, 99–101
character-manipulation, 101–102
creating, 581
date, 107–110
deleting, 582
nested, 113–114
functions (listed by name). *See also* function(s)
ADD_MONTHS function, 108
AVG function, 114–115, 120, 161, 251, 582
CONCAT function, 101, 102–103
COU[NT] function, 251
COUNT(*) function, 115–116, 117
COUNT(column) function, 114, 115–116
DECODE function, 112–113
full_name function, 582
INITCAP function, 100
INSTR function, 101, 105
LAST_DAY function, 108, 109–110
LENGTH function, 101, 105
LOWER function, 100
LPAD function, 101, 103, 170–171
LTRIM function, 104
MAX[IMUM] function, 115, 116–117, 157–158, 251, 582
MIN function, 115, 116–117, 251, 582
MOD function, 106
MONTHS_BETWEEN function, 108–109
NEXT_DAY function, 108, 109
NVL function, 112–113, 118, 151
ROUND function, 106, 108, 110

Continued

functions (listed by name) *(continued)*
RPAD function, 103
RTRIM function, 104
SQLCODE function, 548, 549, 556
SQLERRM function, 548, 549, 556
STD function, 251
SUBSTR function, 101–102, 114
SUM function, 114, 251, 279
SYSDATE function, 112, 277, 300
TO_CHAR function, 93–96, 98–99, 248, 420–421, 445
TO_DATE function, 194, 197–198, 420–421
TO_NUMBER function, 92–93, 420–421
TRIM function, 101, 103–104, 516
TRUNC function, 106, 108, 110
UID function, 300
UPPER function, 100, 199
USER function, 300
USERENV function, 300
VAR[IANCE] function, 251

G

genealogical databases, 5
Gennick, Johnathan, 718
GET command, 238–239
GLOBAL QUERY REWRITE privilege, 323
GLOBAL TEMPORARY clause, 276
global variables, 422. *See also* variable(s)
GNU software, 620
GOTO statement, 454, 455, 541
GRANT command, 38, 372, 373–374
Gray, Jim, 206
greater than (>) operator, 58–59
greater than or equal to (>=) operator, 58–59
GROUP BY clause, 119–122, 310, 315
indexes and, 321, 322
sequences and, 330
subqueries and, 157
groups. *See also* GROUP BY clause
creating, with the BREAK command, 250–251
performing calculations on, 251–252

H

hard disk space requirements, for the CD, 616
HAVING clause, 122–123, 330, 499, 582
headers
adding, 245–247
display of, controlling, 243
PAGESIZE option and, 244
HEA[DING] option, 243, 248
"Hello World" program, 574–576
Hernandez, Michael J., 718, 273
HIDE option, 236
hierarchical queries. *See also* hierarchy
basic description of, 166–172
LEVEL pseudo-column and, 170–171
hierarchy. *See also* hierarchical queries
basic description of, 6
limiting rows in, 171–172
re-creating, 167
starting queries in, establishing a location for, 167
tree, "walking," 167
HomePhone column, 670, 672
HOST command, 255
host variables (bind variables), 236, 414, 422, 430, 482, 577
HTML. *See* HyperText Markup Language (HTML)
Hungry Minds Worldwide Customer Service, 620
HUNGRYMINDS folder, 619
HyperText Markup Language (HTML)
ER/Studio and, 610
Rapid SQL and, 619
hyphen (-), 236

I

IBM (International Business Machines), 619
identifiers
basic description of, 410
exceptions as, 539, 541
declaration of, 410–411
IF statement, 459, 488–489, 542
IF...THEN statement, 455–456
implicit cursors, 481–484, 487–489. *See also* cursor(s)
IN operator, 66, 68–69
IN OUT parameters, 20, 576, 578, 579
IN parameters, 20, 576–577
INCREMENT BY parameter, 332, 327
indents, 39
index(es). *See also* counters
basic description of, 18
bitmap, 319
B-tree, 319, 320–322
-by tables (PL/SQL tables), 424–426, 506–509, 515
creating, 318–325
data dictionary views and, 22, 23
dropping, 288, 304, 325
function-based, 322–323
having too many, problems with, 18
information in, viewing, 324
names, 319
negative conditions and, 65
rules for, 320–321
security and, 323, 368–369
INDEX BY BINARY_INTEGER clause, 511
INDEX privilege, 373
INITCAP function, 100
initialization, of rows, 512
inline views
basic description of, 157
correlated subqueries and, 161
SELECT statement and, 157–159
input/output (I/O) operations, 321
INSERT ANY TABLE privilege, 367
INSERT_DATA.SQL, 674, 676, 681–686
INSERT privilege, 373

INSERT statement, 462, 484–486
DML and, 193–194, 196, 201, 204
preventing the execution of, 255
sequences and, 329–330, 333
server processes and, 204
subqueries in, 201
triggers and, 588, 589, 590, 592, 593, 595
UNUSED columns and, 286
variables in, 485
views and, 310, 314, 317
installing
the CD, 616–617
Oracle databases, 28–29
Oracle8*i*, 616
instances, connecting to, 363
Instant Message for Oracle v2.5, 615
INSTEAD OF triggers, 312–313, 315, 316, 597–598
INSTR function, 101, 105
InstructorClasses view, 317
InstructorCost index, 325
InstructorID column, 113–114, 142–143, 145–146, 160, 301, 303, 331, 670, 672
Instructors table, 113–114, 116, 143, 198, 296, 670
InstructorType column, 670
INT datatype, 416
INTEGER datatype, 416
integrity, of data, 254, 291–308, 595. *See also* referential integrity constraint
INTERSECT operator, 165–166
INTO clause, 482, 489–490, 492–493
Introduction to Database Systems, An, 7th Edition (Date), 718
INVALID_CURSOR exception, 491, 543
INVALID_NUMBER exception, 543, 550
IS NULL operator, 66, 71–72
isolation, 206, 210
%ISOPEN attribute, 487, 493, 497

J

Jacobs, Ken, 718
Java, 20, 21
JobID column, 673
JobName column, 673
join(s)
condition, basic description of, 9
correlated subqueries and, 161
cross-, 145–146
indexes and, 320
nested subqueries and, 152
outer, 148–149
self-, 149–151, 167
using table aliases in, 144
working with, 141–151
writing SELECT statements using, 482
join (+) operator, 147
JUS[TIFY] option, 248

K

keys. *See also* foreign key(s); primary keys
indexes and, 18
self-referential, 149
use of the term, 318
keywords. *See also* keywords (listed by name)
including, as alias names, 50
rules for, 39–40
keywords (listed by name). *See also* keywords
ALL keyword, 372
AS keyword, 49
BEGIN keyword, 411
CASCADE keyword, 303
CONSTRAINT keyword, 294
DECLARE keyword, 409–411, 540
DEFAULT keyword, 414, 415, 423
DISTINCT keyword, 53–55, 118, 315
END keyword, 536
EXCEPTION keyword, 409, 539
ON keyword, 372
PUBLIC keyword, 372
SELECT keyword, 54
VALUES keyword, 193, 201
Kimball, Ralph, 273, 718
Knowledge Base for Active SQL*Plus, 615, 619
Kreines, David C., 718

L

labels, nested, 451–455
LABS folder, 617
languages, foreign. *See* multilingual environments
LAST_DAY function, 108, 109–110
LAST_DDL_TIME column, 603
LastName column, 59, 296, 320, 670, 671
LastUpdated column, 673
LE[FT] option, 246
LENGTH function, 101, 105
less than (<) operator, 58–59
less than or equal to (<=) operator, 58–59
LEVEL pseudo-column, 170–171, 300
LGWR (Log Writer), 203, 205
Library Cache, 203
LIKE operator, 66, 69–71
lines, controlling the number of, on a single page, 244
LINESIZE option, 243–244, 253
Linux, 616, 617, 675
lira symbol (L), 94
LIST command, 230
LOB datatype, 12, 23
local variables, 556. *See also* variable(s)
LocationID column, 271, 288, 290, 298, 320, 331, 671, 672
LocationName column, 671
Locations table, 144, 149, 194, 202–203, 288, 298, 320, 671
locks, 205, 206, 210–211
logic. *See also* logical operators
conditional, 112
controlling transactions and, 205–206
DECODE function and, 112–113
errors, 537–539

logical operators
AND operator, 63–64, 143–144, 147, 172, 418–420
NOT operator, 65, 418–420
OR operator, 64, 418–420
login. *See* logon
LOGIN_DENIED exception, 543
login.sql, 252
logoff, 599
logon, 252, 365, 543, 599
logs
error, 552
redo, 203–204, 206
triggers and, 599
LONG datatype, 14, 116, 274, 417, 549
LONG option, 244
LONG RAW datatype, 14, 116, 275, 417
LONG_VALUE column, 255
loop(s)
basic, 445–446
control structures and, 406
nested, 451–455
overview of, 445–455
LOOP statement, 446
LOWER function, 100
LPAD function, 101, 103, 170–171
LTRIM function, 104

M

maintenance tasks, 203
Management table, 149–151, 167, 168
Manager column, 149–151, 167–168
MAX[IMUM] function, 115, 116–117, 157–158, 251, 582
MAXVALUE parameter, 327, 332
memory. *See also* memory buffer(s)
areas in, referred to as cursors, 427–428
caches, 203–204, 331, 332
packages and, 21, 583
pointers, 428, 497–499
requirements for the CD, 616
resources, preserving, 416
sequences and, 326, 331
shared pool, 583
storage of cursor state information in, 203
variables and, 410, 416, 427
memory buffer(s). *See also* memory
commands stored in, executing, 230
contents, saving, 237–239
editing, 230, 237–239, 241
redo long, 204–205
writing lines of text to, 430–431
metadata, basic description of, 21
Mgr.ID column, 150
Microsoft Excel, 242
Microsoft SQL Server, 610
Microsoft Windows 9*x*, 616, 675
Microsoft Windows 2000, 616, 675
Microsoft Windows ME, 616
Microsoft Windows NT, 616, 675
MiddleInitial column, 670, 671
MIN function, 115, 116–117, 251, 582
MINUS operator, 166
MINVALUE parameter, 327, 332
MOD function, 106
modularity, 405–406
MONTHS_BETWEEN function, 108–109
multi-line comments, 413. *See also* comments
multilingual environments, 94, 417
multimedia, 10
multiply (*) operator, 42, 43, 44–45
multi-row functions, 411, 482
assessment questions, 126–128, 132–133
basic description of, 114–119
DISTINCT keyword and, 118
GROUP BY clause and, 119–122
HAVING clause and, 122–123
identifying available, 119
lab exercises, 129–131, 135–137
pre-test, 90, 131–132
scenarios, 129, 133–135
using the WHERE clause with, 119
mutating tables, 596

N

Name column, 150
National Language Support (NLS)
character sets, 15, 417
currency settings, 94
date format, 61, 96–98, 194, 197
NATURAL datatype, 416
NATURALN datatype, 416
NCHAR(n) datatype, 417
NCHAR(size) datatype, 15, 275
NCLOB datatype, 15, 275, 417
negative conditions, avoiding, 65
nested
blocks, 459–461, 549–555
functions, 113–114
labels, 451–455
loops, 451–455
tables, 424–426, 511–513, 515–516
network model, for databases, 6
NEXT method, 516
NEXT_DAY function, 108, 109
NEXTVAL pseudo-column, 300, 329–331
NLS. *See* National Language Support (NLS)
NLS_currency parameter, 94
“no rows selected” message, 59
NOCACHE parameter, 328
NO_DATA_FOUND exception, 483, 543, 544, 547, 549, 551
NO_DISTINCT column, 118
NODUP[LICATES] option, 250
NOFORCE clause, 311
NOMAXVALUE parameter, 327

NOMINVALUE parameter, 327
nonequijoins, 146–148. *See also* join(s)
nonpairwise condition, 156
nonschema users, 361, 362
NOPRI[NT] option, 248
not equal to (<> or !=) operator, 58–59, 156
NOT EXISTS operator, 160–161
NOT NULL constraint, 16, 195, 292, 295, 300–301, 521
  CREATE TABLE statement and, 274, 277
  dropping, 302–303
  views and, 308, 315
NOT operator, 65, 418–420
Notepad, 238, 252
%NOTFOUND attribute, 487, 488, 493, 497, 498
NOVALIDATE constraint, 304
NOWAIT option, 502–503
NULL constraint, 274, 277, 295–296, 321
NULL operator, 196
NUMBER datatype, 235–236, 248, 416, 426
number functions, 105–107, 251
NUMBER(p,s) datatype, 14, 274
NUMERIC datatype, 115, 415–416, 445
NUMERIC_VALUE column, 255
NVARCHAR2(n) datatype, 417
NVARCHAR2(size) datatype, 14, 15, 274
NVL function, 112–113, 118, 151

O

object(s). *See also* object-oriented programming (OOP)
  assessment questions, 339–342, 347–349
  basic description of, 11–23, 267–358
  compile errors and, 535, 536
  creating, ground rules for, 269–272
  extensions, 10
  hierarchical queries and, 167
  lab exercises, 343–345, 351–358
  naming, 270–271
  orphan, 364
  pre-test, 268, 345–346
  privileges, 366, 372–377, 379
  scenarios, 342–343, 349–350
  scripts used to create, text of, 676–686
  security and, 361, 364, 366, 372–377, 379
  types, 407, 426
Object Relational Database Management System (ORDBMS), 9–10
OBJECT_ID column, 603
OBJECT_NAME column, 603
object-oriented programming (OOP), 407, 424, 426. *See also* object(s)
OBJECT_TYPE column, 603
OfficePhone column, 296, 670
ON keyword, 372
OOP. *See* object-oriented programming (OOP)
OPEN command, 489–491
operators. *See also* operators (listed by name)
  available in PL/SQL expressions, 411–412
  basic description of, 47
  order of precedence for, 44–45, 65–66
operators (listed by name). *See also* operators
  add (+) operator, 42, 43, 44–45
  ALL operator, 153–155
  AND operator, 63–64, 143–144, 147, 172, 418–419
  ANY operator, 153–155
  AS operator, 49
  BETWEEN operator, 147
  BETWEEN...AND operator, 66, 67–68
  concatenation (||) operator, 45–46, 52–53, 91–92, 102, 411
  divide (/) operator, 42, 43, 44–45
  equal to (=) operator, 58–59
  EXISTS operator, 160–161
  greater than (>) operator, 58–59
  greater than or equal to (>=) operator, 58–59
  IN operator, 66, 68–69
  INTERSECT operator, 165–166
  IS NULL operator, 66, 71–72
  join (+) operator, 147
  less than (<) operator, 58–59
  less than or equal to (<=) operator, 58–59
  LIKE operator, 66, 69–71
  MINUS operator, 166
  multiply (*) operator, 42, 43, 44–45
  NOPRI[NT] option, 248
  not equal to (<> or !=) operator, 58–59, 156
  NOT EXISTS operator, 160–161
  NOT operator, 65, 418–420
  NULL operator, 196
  OR operator, 64, 418–420
  subtract (-) operator, 42, 43, 44–45
  UNION ALL operator, 165
  UNION operator, 162–164, 166
OR operator, 64, 418–420
Oracle Certified Professional program, 66, 719
Oracle Certified Professional Program Candidate Guide, 666
Oracle Design (Ensor and Stevenson), 273, 718
Oracle Developer, 407, 575, 589
Oracle Forms, 20, 574, 575
Oracle PL/SQL 101 (Allen), 717
Oracle PL/SQL Programming, 2nd Edition (Feuerstein), 718
Oracle SAM, 615, 618
*Oracle SQL*Plus: The Definitive Guide* (Gennick), 718
Oracle Technical Network, 28, 616
Oracle Web site, 28, 616, 719
Oracle8*i*
  Administrators Companion, 369
  Administrators Guide, 365, 366, 718
  Data Warehousing Guide, 323
  Enterprise Edition, 616
  SQL Reference, 270, 363, 365, 718
  trial copy of, downloading, 616

Oracle9*i*, 11
ORDBMS. *See* Object Relational Database Management System (ORDBMS)
ORDER BY clause, 55–57, 63, 74, 159, 233
  cursor declarations and, 490
  indexes and, 321, 322
  PL/SQL functions and, 582
  SELECT statements and, 482
  sequences and, 330
  SET operators and, 162, 163
  subqueries and, 152
order of precedence, 44–45, 65–66
OrderDetails table, 276
OrderEntry role, 379
Orders table, 276, 362
orphan objects, 364
OUT parameters, 20, 577–579

P

package(s)
  accessing programs and variables in, 586–587
  basic description of, 21, 408, 583–588
  body of, 584–586
  contents, listing, 588
  data dictionary views and, 22
  names, 584
  removing, 587
  specification, 583–584
padding characters, 93
PAGESIZE option, 244
pair-wise operations, 202
parameter(s)
  cursor, 500–501
  declaring, 576–577
  names, 576–577
  passing, 576
parent/child relationships, 167, 299
parentheses, 44, 45, 47, 202, 298, 512
parse phase, 204
passwords. *See also* privileges; security
  changing, 365–366
  connecting to instances and, 363
  expiration dates, 23
  scripts and, 674
  SYS users and, 361, 362
  unencrypted, specification of, 363
PAUSE option, 244–245
Pentium II processors, 616
percent sign (%), 69–70
PerDiemCost column, 199, 202, 670
PerDiemExpenses column, 52, 72, 111, 116, 202, 670
PerDiemExpenses field, 111
performance issues, 161, 406, 420–421
period (.), 248, 505
permissions, 9, 23, 271. *See also* privileges
  CREATE PUBLIC SYNONYM permission, 335, 367
  deleting, 376–377
  DROP PUBLIC SYNONYM permission, 336–337
  granting, 362–366, 378–380
  packages and, 583
  synonyms and, 335, 336
PGA. *See* Private Global Area (PGA)
PhoneNumber datatype, 10
pipe symbol (|), 45, 249
PL/Formatter, 615, 619
portability, 407
POSITIVE datatype, 416
POSITIVEN datatype, 416
PostalCode column, 670, 671, 672
pound sign (#), 270, 414
pragma, 545
Price column, 673, 674
PRIMARY KEY constraint, 16–18, 288–297, 301–305. *See also* primary keys
  disabling/enabling, 304–305, 307
  dropping, 302–303
  indexes and, 319, 322, 324
  sequences and, 325, 329, 330–331
  TRUNCATE TABLE command and, 289
  viewing information about, 308
primary keys. *See also* PRIMARY KEY constraint
  basic description of, 7–8
  dropping tables and, 288
  implicit cursors and, 481
PRINT command, 236, 422, 430
printspec clause, 246–247
PRIOR command, 169
PRIOR method, 516
Private Global Area (PGA), 203, 204
private programs, 584
private variables, 584. *See also* variable(s)
privileges. *See also* permissions; security
  for creating indexes, 323
  dropping tables and, 288
  granted, determining, 371–372, 375–377, 380–381
  granting/administering, 366–372
  object, 372–377, 379
  revoking, 370–371, 376–377
  system, 366–372, 379
procedure(s)
  basic description of, 408, 574–576
  client-side, 575
  data dictionary views and, 22
  debugging and, 537
  deleting, 580
  error handling and, 553–555
  executing blocks and, 429
  nesting, 579–580
  parameters and, 576–578
  server-side, 574, 575–576
  writing lines of text to buffers with, 430–431
Procedure Builder, 20, 537, 574
Process Monitor, 203, 210
processors, 616
PRODUCT column, 255

PRODUCT_USER_PROFILE table, 254–255
program units, 20–21
PROGRAM_ERROR exception, 543
pseudo-columns
  CURRVAL pseudo-column, 300, 329–330
  LEVEL pseudo-column, 170–171, 300
  NEXTVAL pseudo-column, 300, 329–331
  ROWID pseudo-column, 176
  ROWNUM pseudo-column, 300, 315
PUBLIC keyword, 372
public programs, 583
public variables, 584. *See also* variable(s)
PUPBLD.SQL, 254
PUT_LINE procedure, 430

Q

QUERY REWRITE privilege, 323

R

RAISE statement, 539, 546–547, 552, 553
RAISE_APPLICATION_ERROR procedure, 553–555
RAM (random-access memory). *See* memory
Rapid/SQL, 615, 619
RAPIDSQL folder, 619
RAW datatype, 116, 417
RAW(size) datatype, 14, 275
RDBMS. *See* relational database management system (RDBMS)
read consistency, 205
READ ONLY option, 316, 317–318
readability, of code, 39–40, 409, 414, 619
REAL datatype, 416
record(s)
  basic description of, 423
  composite datatypes and, 423–324, 504–506
  creating, 504–506
  cursor FOR loop and, 495–497
  declaring, that will hold the values returned by a cursor, 494
  fitting, on one line, 243–244
  locking, 503
  populating, with a SELECT statement, 505
  returned by cursors, fetching, 498–499
recoverability, 205
Redo Log Buffer, 204–205
Redo Log Files, 205
REF CURSOR variable, 428, 497–498
reference types, 427–428
REFERENCES privilege, 373, 377
REFERENCING clause, 592
REFERENCING_NAMES column, 602
referential integrity constraint, 297, 546. *See also* FOREIGN KEY constraint
REHEADER command, 246
relational database management system (RDBMS)
  advanced SELECT statements and, 141
  basic description of, 6–9, 37
  characteristics of, 6–9
  database objects and, 11–23
  modifying data and, 193
  Oracle as a, 6–7
  ORDBMS and, 10
Relational Model for Database Management, Version 2 (Codd), 7, 717
"Relational Model of Data for Large Shared Data Banks, A" (Codd), 6
relational operators, 411
REPFOOTER command, 246–247
REPHEADER command, 246
REPLACE option, 238
ReplacesCourse column, 196, 670
report(s)
  formatting, 245–252
  titles, 246
  triggers and, 575
result sets
  eliminating duplication in, 53–55
  joins and, 143, 147
  MINUS operator and, 166
RetailPrice column, 286, 670
RETURN statement, 580
Reuter, Andreas, 206
RevealNet, 615, 619
REVOKE command, 38, 370–371, 377
R[IGHT] option, 246
role(s)
  basic description of, 377–382
  creating/granting, 378–380
  granted, determining, 380–381
  revoking, 381–382
ROLE_SYS_PRIVS view, 380
ROLE_TAB_PRIVS view, 380
Rollback Segment, 204–206, 210
ROLLBACK statement, 462, 484, 502
  DML and, 205, 207–209
  sequences and, 331
  triggers and, 595
ROUND function, 106, 108, 110
row(s)
  adding, 193–198, 484–486
  deleting, 199–205, 289, 487
  fetching, 492–493
  initializing, 512
  -level triggers, 588, 591–594, 596, 601
  limiting, using the WHERE clause, 57–75
  not meeting constraint conditions, 306
  number of, passed into functions, 115–116
  populating, in index-by tables, 508
  searching for specific, 306
  sorting, 55–57
  subqueries that return multiple, 152–155
  updating, 198–199, 486–487
%ROWCOUNT attribute, 463, 487, 493, 497
ROWID datatype, 14, 18, 275, 277, 318, 417–418
ROWID pseudo-column, 176
ROWNUM pseudo-column, 72–75, 300, 315
%ROWTYPE attribute, 424, 494, 504–505, 511

ROWTYPE_MISMATCH exception, 543
RPAD function, 103
RTRIM function, 104
rules
  business, 292
  for indexes, 320–321
  for keywords, 39–40
  for objects, 269–272
  for SQL statements, 39–40
RUN command, 230, 231, 238–239
runtime. *See also* runtime errors (exceptions)
  variables, 172, 173, 175
  writing SQL statements at, 407
runtime errors (exceptions), 414, 489. *See also* error(s)
  assessment questions, 558–563, 566–567
  basic description of, 539
  coding conventions and, 555–556
  cursors and, 491
  declaring, 544
  handling, 412, 539–547
  implicit cursors and, 481–483
  lab exercises, 563–565, 567–570
  names, 546
  non-predefined, 540, 544–546
  NOTWAIT option and, 502
  predefined, 540, 541–544
  pre-test, 534, 565–566
  propagation, in nested blocks, 549–555
  scenarios, 563, 567
  unhandled, 539–540, 542, 553–554
  user-defined, 540, 546–547
  variable names and, 483

S

Sales table, 146–147
Sales_record table, 92
Salutation column, 670, 671
SAVE command, 237–238, 253
SAVEPOINT statement, 331, 462, 465–466
savepoints, 208, 331, 462, 465–466
scalar datatypes, 13–15, 19, 274–275, 413–423, 500
scalar variables, 413–423, 500
ScheduledClasses table, 43, 141–148, 288, 303, 307–308, 320, 334, 545, 596–597, 603, 672
schema(s)
  basic description of, 13
  data dictionary views and, 22, 23
  owners, 361, 362
  security and, 361–366
  users, 13, 361
scientific notation, 411
scope, 460, 539
screen output, sending, to a file, 239–240
script(s). *See also* scripts (listed by name)
  copying, from the CD, 674, 675–676
  creating, 252–254
  executing, 252–254, 675–676
  display options for, 245
  tasks which must be completed before running, 674–675
  titles, 253
  used to create database objects, 674–686
scripts (listed by name). *See also* script(s)
  CERTDBOBJ.SQL, 674, 676–681
  CREATEUSER.SQL, 674, 676–677
  enrollmenthistory.sql, 164
  INSERT_DATA.SQL, 674, 676, 681–686
  login.sql, 252
  PUPBLD.SQL, 254
  selfjoin.sql, 149
  UTLEXCPT.SQL, 306
SCRIPTS folder, 617
scrolling, through data returned by SELECT statements, 244–245
security. *See also* permissions; privileges; user(s)
  assessment questions, 384–387, 390–391
  basic Oracle model for, 361
  lab exercises, 387–389, 392–396
  passwords, 23, 361–363, 365–366, 674
  policy, 365
  pre-test, 360, 389–390
  for the PRODUCT_USER_PROFILE table, 254
  scenarios, 387, 391–392
  schemas and, 361–366
SELECT ANY TABLE privilege, 367
SELECT INTO statement, 489
SELECT keyword, 54
SELECT list
  advanced SELECT statements and, 141, 144, 151, 157
  arithmetic operations and, 41–42
  basic description of, 40–41
  creating an alias for columns in, 51–52
  DISTINCT keyword and, 118
  GROUP BY clause and, 120–121
  INSERT statement and, 201
  joins and, 141
  multi-row functions and, 115
  placing DISTINCT keywords in, 54
  placing string values in, 46
  PL/SQL functions and, 582
  referencing pseudo-columns as part of, 72–75
  subqueries and, 151, 157, 201–202
SELECT privilege, 373, 375, 376
SELECT statement
  advanced, 139–190
  arithmetic operations and, 41–55
  ARRAYSIZE option and, 241
  assessment questions, 179–182, 185–187
  basic description of, 37, 40–57
  calling functions from, 582
  COLSEP option and, 241–242
  column aliases and, 47–52
  concatenating columns and, 45–46
  cursors and, 428, 481–484, 488–490, 496–501
  date values and, 43–44

duplication in result sets and, 53–55
error handling and, 547, 551
fetch phase and, 205
hierarchical queries and, 166–172
indexes and, 320, 323
inline views and, 157–159
joins and, 141–151
lab exercises, 183–184, 188–190
order of precedence and, 44–45
ordering data in, 55–57
PL/SQL records and, 504–506
pre-test, 140, 185
scenarios, 182–183, 187–188
scrolling through data returned by, 244–245
sequences and, 330
server processes and, 204
single-row functions and, 91
SQL buffer and, 229–230
subqueries and, 151–161
substitution variables and, 172–176, 233
UNUSED columns and, 286
views and, 17–18, 309–310, 313–315
Self Test Software, 615, 618
SELF_IS_NULL exception, 543
self-joins, 149–151, 167
selfjoin.sql, 149
self-referential keys, 149–151
SELFTEST folder, 618
semicolon (;), 253, 409, 410, 446, 453, 536
sequence(s)
basic description of, 17
benefits of, 325–326
caching, 326, 331, 332
creating, 325–328
cycle status for, 332
data dictionary views and, 22
dropping, 333
getting information on, 331–332
modifying, 332–333
using, 329–331
values, non-cached, 332
server(s). *See also* client(s)
errors and, 406, 539
internal clocks for, 112
monitoring, 203
parsing of blocks by, 429
PL/SQL and, 406, 407, 412, 429–430
process, 203–205
running scripts and, 674
sending blocks to, 429–430
-side procedures, 574, 575–576
triggers and, 589
SERVEROUTPUT environment variable, 430–431, 445
SET clause, 199, 201–202
SET commands, 240–245
SET operators, 161–166
SET PAUSE OFF option, 245
SET PAUSE ON option, 245
SET VERIFY OFF command, 173
setup.exe, 617
Shared Pool, 203, 583
Shared SQL Areas, 203, 204
shareware programs, 620
SHOW ERRORS command, 536–537
SIGNTYPE datatype, 416
single quote (’), 46, 59, 69, 173, 193, 196, 290, 411
single-line comments, 413. *See also* comments
single-row functions. *See also* single-row functions (listed by name)
assessment questions, 126–128, 132–133
basic description of, 91
character functions, 99–110
conversion functions, 91–98
date functions, 107–110
lab exercises, 129–131, 135–137
nesting functions, 113–114
number functions, 105–107
PL/SQL and, 411
pre-test, 90, 131–132
scenarios, 129, 133–135
using, with date values, 96–98
using, with numeric data, 93–96
writing SELECT statements using, 482
single-row functions (listed by name). *See also* single-row functions
ADD_MONTHS function, 108
CONCAT function, 101, 102–103
DECODE function, 112–113
INITCAP function, 100
INSTR function, 101, 105
LAST_DAY function, 108, 109–110
LENGTH function, 101, 105
LOWER function, 100
LPAD function, 101, 103, 170–171
MOD function, 106
MONTHS_BETWEEN function, 108–109
NEXT_DAY function, 108, 109
NVL function, 112–113, 118, 151
ROUND function, 106, 108, 110
SUBSTR function, 101–102, 114
SYSDATE function, 112, 277, 300
TO_CHAR function, 93–96, 98–99, 248, 420–421, 445
TO_NUMBER function, 92–93, 420–421
TRIM function, 101, 103–104, 516
TRUNC function, 106, 108, 110
UPPER function, 100, 199
SKI[P] option, 246, 250
SMALLINT datatype, 416
SOFTWARE folder, 618
SOLUTIONS folder, 617
SPOOL command, 239–240
SPOOL OFF command, 240
SPOOL OUT command, 240
SQL. *See* Structured Query Language (SQL)

.sql files
CERTDBOBJ.SQL, 674, 676–681
CREATEUSER.SQL, 674, 676–677
enrollmenthistory.sql, 164
INSERT_DATA.SQL, 674, 676, 681–686
login.sql, 252
PUPBLD.SQL, 254
selfjoin.sql, 149
UTLEXCPT.SQL, 306
SQL Programmer 2001, 615, 619
SQL Server (Microsoft), 610
SQL-92 standard, 58
SQL-99 standard, 11, 58
SQLCODE function, 548, 549, 556
SQLERRM function, 548, 549, 556
SQL.LNO variable, 247
SQL*Plus, 426, 429–431, 445, 455, 462–464
assessment questions, 257–259, 262
basic description of, 40
buffer, editing, 237–239, 241
commands, 237–240
creating scripts with, 252–254
customizing, 240–245, 254–255
DELETE statement and, 488
environment, 227–266
error handling and, 535–536, 540, 541
executing blocks/scripts in, 252–254, 429–431
footers and, 245–247
formatting output with, 245–252
headers and, 245–247
lab exercises, 260–261, 263–266
pre-test, 228, 261
repeating values and, 174
runtime variables and, 172–173, 174
saving environment settings with, 252
scenarios, 259–260, 262–263
security and, 363
SQL buffer and, 229–230
substitution variables and, 172–173
user-defined types and, 426
User's Guide and Reference, 718, 254
using, as your SQL editor, 209
variables, printing, 430
SQL.PNP variable, 247
SQLPROG folder, 619
STANDARD package, 541
START command, 239, 254
StartDate column, 43, 672
State column, 316, 670, 671
statement(s). *See also* statements (listed by name)
execution order of, 123
-level triggers, 588–591, 596, 601
preventing the execution of, 255
statements (listed by name). *See also* DELETE statement; INSERT statement; SELECT statement; UPDATE statement
CASE statement, 457
COMMIT statement, 205, 207–210, 331, 462, 484, 502, 595
CONNECT PRIOR statement, 169
CREATE statement, 269
CREATE TABLE statement, 273–283
CREATE VIEW statement, 157, 310–311
DROP statement, 269
END IF statement, 457, 458
END LOOP statement, 446, 453
EXIT statement, 446, 447, 454–455
EXIT WHEN statement, 449
GOTO statement, 454, 455, 541
IF statement, 459, 488–489, 542
IF...THEN statement, 455–456
LOOP statement, 446
RAISE statement, 539, 546–547, 552, 553
RETURN statement, 580
ROLLBACK statement, 205, 207–209, 331, 462, 484, 502, 595
SAVEPOINT statement, 462, 331, 465–466
SELECT INTO statement, 489
STATUS column, 602, 603, 672, 673
STD function, 251
Stevenson, Tim, 273, 718
STORAGE_ERROR exception, 543
stored programs. *See also specific types*
assessment questions, 605–607, 610
basic description of, 571–614
lab exercises, 608–609, 611–614
pre-test, 572, 609–610
scenarios, 607–608, 610–611
STRING datatype, 417
Structured Query Language (SQL)
arithmetic operations and, 41–55
assessment questions, 77–79, 83–85
basic description of, 8–9
buffer, 229–230
column aliases and, 47–52
comparison operators in, 58–63, 66–72
duplication in result sets and, 53–55
lab exercises, 80–82, 86–88
logical operators and, 63–66
nesting functions and, 113–114
overview, 37–40
PL/SQL support for, 406–407
pre-test, 36, 82–83
scenarios, 80, 85–86
standards, 11
statements, basic description of, 481–489
statements, general rules for, 39–40
wild card operators and, 69–70
STUDENT user, 674
student_enrolled cursor, 490–493
StudentID column, 8
StudentNumber column, 295, 297, 329, 331, 671, 673
StudentNumber constraint, 293
Students table, 6–8, 13, 200–201, 271, 293, 320, 333, 421–422, 671

subprograms. *See also specific types*
basic description of, 573
server-side, 575
stored, 575, 408–409
subqueries
basic, 151–157
correlated, 159–161
creating tables using, 278–279
cursors with, 499–500
deleting data with, 200–203
in DML statements, 200–203
in INSERT statements, 201
multi-column, 155–157
nested, 152
performance issues with, 161
that return multiple rows, 152–155
in UPDATE statements, 201–202
using, 200–203
working with, 151–161
writing SELECT statements using, 482
SUBSCRIPT_BEYOND_COUNT exception, 543
SUBSCRIPT_OUTSIDE_LIMIT exception, 544
substitution variables, 172–176, 231–234
ACCEPT command and, 175
repeating values and, 174
SUBSTR function, 101–102, 114
subtract (-) operator, 42, 43, 44–45
SUM function, 114, 251, 279
Sun SPARC, 616
Sybase, 619
Sylvain Faust International, 615, 619
Sylvan Prometric testing centers, 666
synonyms
basic description of, 19
creating, 333–335
dropping, 336–338
getting information on, 335–336
marking definitions as INVALID in, 288
public, 335
SYS user, 22, 270–271, 361–363
SYSDATE command, 196, 197
SYSDATE function, 112, 277, 300
SYSDATE variable, 485
system
crashes, 205, 209, 326, 331
privileges, 366–372, 379
requirements, for the CD, 616
returning the current date from, 112
System Global Area (SGA), 203–204
SYSTEM user, 254, 270–271, 361–363, 674

T

tab characters, 39
table(s). *See also* column(s); row(s)
adding constraints to, 301–302
base, 22–23, 309–310
basic description of, 13
creating, 28–29, 272–291
documenting, 289–291
dropping, 288–289
lists, displaying, 280–281
locking, 299
managing, 272–291
modifying, 283–286
physical location of, on disk, 9
relational databases and, relationship of, 7–9
truncating, 289–291
TABLE_NAME column, 602
TABLE_OWNER column, 602
tablespaces, 369, 674
Technology Track, 616
Telephone column, 671
TERMOUT option, 245
TESTEXAM folder, 618
text editors, 230, 238, 252
thousands separator, 93
time. *See also* date(s)
connect, maximum, 365
-stamps, 205
TIMEOUT_ON_RESOURCE exception, 544
TITLE command, 246
titles
for reports, 246
for scripts, 253
TO_CHAR function, 93–96, 98–99, 248, 420–421, 445
TO_DATE function, 98, 194, 197–198, 420–421
TO_MANY_ROWS exception, 482–483
TO_NUMBER function, 92–93, 420–421
total_sales collection, 425
transactions
ACID test and, 206–207
basic description of, 461–462
controlling, 205–212, 461–466
indexes and, 322
locks and, 210–211
rollback of, 207–210
savepoints and, 208, 462, 465–466
tree, "walking the," 167. *See also* hierarchy
trial software, 620
trigger(s)
basic description of, 21, 408, 588
deleting, 600
enabling/disabling, 599–600, 602
event, 588, 589, 598–599
firing order for, 596–597
INSTEAD OF, 312–313, 315, 316, 597–598
mutating, 596
predicates, 594–595
report, 575
resource manager, 598–599
restrictions on, 595–596
row-level, 588, 591–594, 596, 601
statement-level, 588–591, 596, 601
system event, 598–599
views and, 312–313, 315, 316
WHEN clause and, 593–584

TRIGGER_BODY column, 602
TRIGGERING_EVENT column, 601
TRIGGER_NAME column, 601
TRIGGER_TYPE column, 601
TRIM function, 101, 103–104, 516
troubleshooting, problems with the CD, 620
TRUNC function, 106, 108, 110
TRUNCATE command, 200
TRUNCATE TABLE command, 289–291
TTITLE command, 244, 246, 253
%TYPE attribute, 421–422, 482, 504, 507, 513

U

UID function, 300
UNDEFINE command, 175, 234, 235
underscore (_), 69, 270, 414
unhandled exceptions, 539–540, 542, 553–554
UNION ALL operator, 165
UNION operator, 162–164, 166
UNIQUE constraint, 16–18, 288–289, 292, 295–297
  disabling/enabling, 304–305, 307
  dropping, 302–303
  indexes and, 319, 322, 324
  sequences and, 325
  viewing information about, 308
Unix, 675
UPDATE ANY TABLE privilege, 368
UPDATE privilege, 373
UPDATE statement, 198–199, 205–209, 462. *See also* updating data
  controlling transactions and, 205–206
  DEFAULT clause and, 277
  exceptions and, 547
  fetch phase and, 205
  PL/SQL and, 486–487, 488, 503, 547
  preventing the execution of, 255
  server processes and, 204
  subqueries in, 201–202
  triggers and, 588–590, 592–595
  UNUSED columns and, 286
  views and, 310, 314, 316–317
  WHERE CURRENT OF clause and, 503
updating data. *See also* UPDATE statement
  assessment questions, 213–216, 221–222
  with DML statements, 198–199
  lab exercises, 218, 223–224
  pre-test, 182, 220
  scenarios, 216–217, 222–223
UPPER function, 100, 199
UROWID datatype, 14, 275, 417–418
user(s). *See also* privileges
  basic description of, 13, 361
  creating, 362–366
  -defined datatypes, 10, 12–14, 19, 426–427
  -defined exceptions, 540, 546–547
  -defined functions, 20, 580–582
  managing, 362–366
  privileges, granting/administering, 366–372
  removing, from databases, 364
  schemas and, 13, 361–366
  two types of, 361
USER function, 300
USER variable, 485
USERENV function, 300
USERID column, 255
USER_views. *See also* USER_views (listed by name)
  comments and, 290
  querying, 279–280, 290
  synonyms and, 335–336
USER_views (listed by name). *See also* USER_views
  USER_ALL TABLES view, 688, 692–693
  USER_ARGUMENTS view, 688, 693
  USER_CATALOG view, 22, 282, 688, 694
  USER_CLU_COLUMNS view, 688, 694
  USER_CLUSTER_HASH_EXPRESSIONS view, 688, 694
  USER_CLUSTERS view, 688, 694
  USER_COL_COMMENTS view, 290, 688, 695
  USER_COL_PRIVS view, 688, 695
  USER_COL_PRIVS_MADE view, 375–376, 688, 695
  USER_COL_PRIVS_RECD view, 375–376, 688, 695
  USER_CONS_COLUMNS view, 22, 302, 305, 307–308, 688, 696
  USER_CONSTRAINTS view, 22, 688, 695–696, 302, 305, 307–308
  USER_DEPENDENCIES view, 696
  USER_ERRORS view, 22, 688, 696
  USER_EXTENTS view, 688, 696
  USER_FREE_SPACE view, 688, 697
  USER_IND_COLUMNS view, 23, 324, 688, 698
  USER_INDEXES view, 22, 324, 688, 697–698
  USER_IND_EXPRESSIONS view, 23, 689, 698
  USER_IND_PARTITIONS view, 689, 698–699
  USER_IND_SUBPARTITONS view, 689, 699
  USER_LIBRARIES view, 689, 700
  USER_LOB_PARTITIONS view, 689, 700
  USER_LOBS view, 689, 700
  USER_LOB_SUBPARTITIONS view, 689, 701
  USER_METHOD_PARAMS view, 689, 701
  USER_METHOD_RESULTS view, 689, 701
  USER_NESTED_TABLES view, 689, 702
  USER_OBJECTS view, 23, 602–603, 689, 702
  USER_OBJECT_SIZE view, 689, 702
  USER_OBJECT_TABLES view, 689, 702–703
  USER_PART_COL_STATISTICS view, 689, 703–704
  USER_PART_HISTOGRAMS view, 689, 704
  USER_PART_INDEXES view, 690, 704
  USER_PART_KEY_COLUMNS view, 690, 704
  USER_PART_LOBS view, 690, 705
  USER_PART_TABLES view, 690, 705
  USER_PASSWORD_LIMITS view, 690, 706
  USER_RESOURCE_LIMITS view, 690, 706
  USER_ROLE_PRIVS view, 380, 690, 706
  USER_SEGMENTS view, 23, 690, 706
  USER_SEQUENCES view, 23, 331–332, 690, 706–707
  USER_SOURCE view, 600–601, 690, 707

USER_SUBPART_COL_STATISTICS view, 690, 707
USER_SUBPART_HISTOGRAMS view, 690, 707
USER_SUBPART_KEY_COLUMNS view, 690, 707
USER_SYNONYMS view, 23, 335–336, 690, 708
USER_SYS_PRIVS view, 372, 690, 708
USER_TAB_COLUMNS view, 280–281, 691, 709–710
USER_TAB_COL_STATISTICS view, 691, 710
USER_TAB_COMMENTS view, 290, 691, 710
USER_TAB_HISTOGRAMS view, 691, 710
USER_TABLES view, 23, 690, 708–709
USER_TABLESPACES view, 690, 709
USER_TAB_MODIFICATIONS view, 691, 710
USER_TAB_PARTITIONS view, 691, 711
USER_TAB_PRIVS view, 23, 691, 711
USER_TAB_PRIVS_MADE view, 23, 375–376, 691, 711–723
USER_TAB_PRIVS_RECD view, 23, 375–376, 691, 712
USER_TAB_SUBPARTITIONS view, 691, 712
USER_TRIGGER_COLS view, 691, 713
USER_TRIGGERS view, 601–602, 691, 712–713
USER_TS_QUOTAS view, 691, 713
USER_TYPE_ATTRS view, 691, 713–714
USER_TYPE_METHODS view, 692, 714
USER_TYPES view, 691, 713
USER_UNUSED_COL_TABS view, 692, 714
USER_UPDATABLE_COLUMNS view, 692, 714
USER_USERS view, 23, 692, 714
USER_VARRAYS view, 692, 715
USER_VIEWS view, 23, 692, 715
UTLEXCPT.SQL, 306

V

VALIDATE constraint, 304
VALUE_ERROR exception, 544
VALUES command, 330
VALUES keyword, 193, 201
VALUES list, 194–195
VARCHAR datatype, 417
VARCHAR2 datatype, 285, 416, 426, 582
VARCHAR2(n) datatype, 417
VARCHAR2(size) datatype, 14, 15, 274
variable(s). *See also* variables (listed by name)
bind, 236, 414, 422, 430, 482, 577
character datatype, 92
creating, 581
cursors, 497–499
declaring, 231–236, 411–418, 461, 497–498
deleting, 234, 235
environment, 430–431, 445
errors and, 412, 535, 537
global, 422
index-by tables and, 500
local, 556
names, 173, 414, 422, 483
placing single quotes around, 173
PL/SQL and, 406, 410, 413–428
printing, 430
private, 584
public, 584
runtime, 172, 173, 175
scalar, 413–423, 500
scope, 460, 539
specifying a precision and scale for, 416
substitution, 172–176, 231–234
undefining, 175
VARIABLE command, 236, 422, 430, 577
variables (listed by name). *See also* variable(s)
AUTOPRINT variable, 430
class_maximum variable, 584, 587
Colx variable, 174
REF CURSOR variable, 428, 497–498
SERVEROUTPUT variable, 430–431, 445
SQL.LNO variable, 247
SQL.PNP variable, 247
SYSDATE variable, 485
USER variable, 485
VAR[IANCE] function, 251
VARRAY datatype, 513–516, 424–426, 513–515
vi (text editor), 252. *See also* text editors
view(s). *See also* views (listed by name)
advantages of, 309
base tables and, 309–310
basic description of, 17–18, 157
creating, 309–318
dropping, 318
inline, 157–159
INVALID, 288, 318
views (listed by name)
ALL_CATALOG view, 283
ALL_COL_COMMENTS view, 290
ALL_CONSTRAINTS view, 307–308
ALL_SEQUENCES view, 331–332
ALL_SYNONYMS view, 336
ALL_TAB_COMMENTS view, 290
DBA_views, 22, 280, 335–336
InstructorClasses view, 317
ROLE_SYS_PRIVS view, 380
ROLE_TAB_PRIVS view, 380
USER_ALL TABLES view, 688, 692–693
USER_ARGUMENTS view, 688, 693
USER_CATALOG view, 22, 282, 688, 694
USER_CLU_COLUMNS view, 688, 694
USER_CLUSTER_HASH_EXPRESSIONS view, 688, 694
USER_CLUSTERS view, 688, 694
USER_COL_COMMENTS view, 290, 688, 695
USER_COL_PRIVS view, 688, 695
USER_COL_PRIVS_MADE view, 375–376, 688, 695
USER_COL_PRIVS_RECD view, 375–376, 688, 695
USER_CONS_COLUMNS view, 22, 302, 305, 307–308, 688, 696
USER_CONSTRAINTS view, 22, 688, 695–696, 302, 305, 307–308
USER_DEPENDENCIES view, 696
USER_ERRORS view, 22, 688, 696
USER_EXTENTS view, 688, 696
USER_FREE_SPACE view, 688, 697

Continued

views (listed by name) *(continued)*
USER_IND_COLUMNS view, 23, 324, 688, 698
USER_INDEXES view, 22, 324, 688, 697–698
USER_IND_EXPRESSIONS view, 23, 689, 698
USER_IND_PARTITIONS view, 689, 698–699
USER_IND_SUBPARTITONS view, 689, 699
USER_LIBRARIES view, 689, 700
USER_LOB_PARTITIONS view, 689, 700
USER_LOBS view, 689, 700
USER_LOB_SUBPARTITIONS view, 689, 701
USER_METHOD_PARAMS view, 689, 701
USER_METHOD_RESULTS view, 689, 701
USER_NESTED_TABLES view, 689, 702
USER_OBJECTS view, 23, 602–603, 689, 702
USER_OBJECT_SIZE view, 689, 702
USER_OBJECT_TABLES view, 689, 702–703
USER_PART_COL_STATISTICS view, 689, 703–704
USER_PART_HISTOGRAMS view, 689, 704
USER_PART_INDEXES view, 690, 704
USER_PART_KEY_COLUMNS view, 690, 704
USER_PART_LOBS view, 690, 705
USER_PART_TABLES view, 690, 705
USER_PASSWORD_LIMITS view, 690, 706
USER_RESOURCE_LIMITS view, 690, 706
USER_ROLE_PRIVS view, 380, 690, 706
USER_SEGMENTS view, 23, 690, 706
USER_SEQUENCES view, 23, 331–332, 690, 706–707
USER_SOURCE view, 600–601, 690, 707
USER_SUBPART_COL_STATISTICS view, 690, 707
USER_SUBPART_HISTOGRAMS view, 690, 707
USER_SUBPART_KEY_COLUMNS view, 690, 707
USER_SYNONYMS view, 23, 335–336, 690, 708
USER_SYS_PRIVS view, 372, 690, 708
USER_TAB_COL_STATISTICS view, 691, 710
USER_TAB_COLUMNS view, 280–281, 691, 709–710
USER_TAB_COMMENTS view, 290, 691, 710
USER_TAB_HISTOGRAMS view, 691, 710
USER_TABLES view, 23, 690, 708–709
USER_TABLESPACES view, 690, 709
USER_TAB_MODIFICATIONS view, 691, 710
USER_TAB_PARTITIONS view, 691, 711
USER_TAB_PRIVS view, 23, 691, 711
USER_TAB_PRIVS_MADE view, 23, 375–376, 691, 711–723
USER_TAB_PRIVS_RECD view, 23, 375–376, 691, 712
USER_TAB_SUBPARTITIONS view, 691, 712
USER_TRIGGER_COLS view, 691, 713
USER_TRIGGERS view, 601–602, 691, 712–713
USER_TS_QUOTAS view, 691, 713
USER_TYPE_ATTRS view, 691, 713–714
USER_TYPE_METHODS view, 692, 714
USER_TYPES view, 691, 713
USER_UNUSED_COL_TABS view, 692, 714
USER_UPDATABLE_COLUMNS view, 692, 714
USER_USERS view, 23, 692, 714
USER_VARRAYS view, 692, 715
USER_VIEWS view, 23, 692, 715
viruses, 620

W

Wages table, 117–118
“walking the tree,” 167
Web sites, recommended, 28, 616, 719
WHEN condition, 446, 593
WHEN OTHERS clause, 542, 548–549
WHEN_CLAUSE column, 602
WHERE clause
advanced SELECT statements and, 143, 144, 145
case-conversion functions and, 100
comparison operators and, 66–72
constraints and, 300
DELETE statement and, 200
GROUP BY clause and, 121–122
HAVING clause and, 122–123
hierarchical queries and, 171–172
implicit cursors and, 481
indexes and, 320–322
joins and, 148–149
limiting rows using, 57–75
logical operators in, 63–66
PL/SQL functions and, 582
ROWNUM pseudo-column in, 72–75
SET operators and, 162
specifying values used in, at run time, 500–501
subqueries and, 154–155, 159, 201, 499–501
substitution variables and, 173
TRUNCATE TABLE command and, 289
UPDATE statement and, 198, 199
variable names and, 483
views and, 316
WHERE CURRENT OF option, 501–503
WHILE loop, 447, 450
white space, 412
wild card operators, 69–70, 255
Windows 9x (Microsoft), 616, 675
Windows 2000 (Microsoft), 616, 675
Windows ME (Microsoft), 616
Windows NT (Microsoft), 616, 675
WITH ADMIN OPTION privilege, 369–370, 372, 374, 376
WITH CHECK OPTION clause, 312, 315–316
WITH GRANT OPTION privilege, 374–375, 376
WITH READ ONLY option, 312, 318
WITH_DISTINCT column, 118
WordPad, 238
working directory, 252
WorkPhone column, 672
WRA[PPED] option, 248

Z

ZERO_DIVIDE exception, 541, 544, 549, 555, 556
zeros, leading, 546
ZIM Technologies, 615, 618

Hungry Minds, Inc.
End-User License Agreement

READ THIS. You should carefully read these terms and conditions before opening the software packet(s) included with this book ("Book"). This is a license agreement ("Agreement") between you and Hungry Minds, Inc. ("HMI"). By opening the accompanying software packet(s), you acknowledge that you have read and accept the following terms and conditions. If you do not agree and do not want to be bound by such terms and conditions, promptly return the Book and the unopened software packet(s) to the place you obtained them for a full refund.

1. **License Grant.** HMI grants to you (either an individual or entity) a nonexclusive license to use one copy of the enclosed software program(s) (collectively, the "Software") solely for your own personal or business purposes on a single computer (whether a standard computer or a workstation component of a multi-user network). The Software is in use on a computer when it is loaded into temporary memory (RAM) or installed into permanent memory (hard disk, CD-ROM, or other storage device). HMI reserves all rights not expressly granted herein.
2. **Ownership.** HMI is the owner of all right, title, and interest, including copyright, in and to the compilation of the Software recorded on the disk(s) or CD-ROM ("Software Media"). Copyright to the individual programs recorded on the Software Media is owned by the author or other authorized copyright owner of each program. Ownership of the Software and all proprietary rights relating thereto remain with HMI and its licensers.
3. **Restrictions On Use and Transfer.**
   - **(a)** You may only (i) make one copy of the Software for backup or archival purposes, or (ii) transfer the Software to a single hard disk, provided that you keep the original for backup or archival purposes. You may not (i) rent or lease the Software, (ii) copy or reproduce the Software through a LAN or other network system or through any computer subscriber system or bulletin-board system, or (iii) modify, adapt, or create derivative works based on the Software.
   - **(b)** You may not reverse engineer, decompile, or disassemble the Software. You may transfer the Software and user documentation on a permanent basis, provided that the transferee agrees to accept the terms and conditions of this Agreement and you retain no copies. If the Software is an update or has been updated, any transfer must include the most recent update and all prior versions.

4. **Restrictions on Use of Individual Programs.** You must follow the individual requirements and restrictions detailed for each individual program in Appendix A of this Book. These limitations are also contained in the individual license agreements recorded on the Software Media. These limitations may include a requirement that after using the program for a specified period of time, the user must pay a registration fee or discontinue use. By opening the Software packet(s), you will be agreeing to abide by the licenses and restrictions for these individual programs that are detailed in Appendix A and on the Software Media. None of the material on this Software Media or listed in this Book may ever be redistributed, in original or modified form, for commercial purposes.

5. **Limited Warranty.**

   (a) HMI warrants that the Software and Software Media are free from defects in materials and workmanship under normal use for a period of sixty (60) days from the date of purchase of this Book. If HMI receives notification within the warranty period of defects in materials or workmanship, HMI will replace the defective Software Media.

   (b) **HMI AND THE AUTHOR OF THE BOOK DISCLAIM ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING WITHOUT LIMITATION IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, WITH RESPECT TO THE SOFTWARE, THE PROGRAMS, THE SOURCE CODE CONTAINED THEREIN, AND/OR THE TECHNIQUES DESCRIBED IN THIS BOOK. HMI DOES NOT WARRANT THAT THE FUNCTIONS CONTAINED IN THE SOFTWARE WILL MEET YOUR REQUIREMENTS OR THAT THE OPERATION OF THE SOFTWARE WILL BE ERROR FREE.**

   (c) This limited warranty gives you specific legal rights, and you may have other rights that vary from jurisdiction to jurisdiction.

6. **Remedies.**

   (a) HMI's entire liability and your exclusive remedy for defects in materials and workmanship shall be limited to replacement of the Software Media, which may be returned to HMI with a copy of your receipt at the following address: Software Media Fulfillment Department, Attn.: *Oracle8i™DBA: SQL and PL/SQL Certification Bible,* Hungry Minds, Inc., 10475 Crosspoint Blvd., Indianapolis, IN 46256, or call 1-800-762-2974. Please allow four to six weeks for delivery. This Limited Warranty is void if failure of the Software Media has resulted from accident, abuse, or misapplication. Any replacement Software Media will be warranted for the remainder of the original warranty period or thirty (30) days, whichever is longer.

   (b) In no event shall HMI or the author be liable for any damages whatsoever (including without limitation damages for loss of business profits, business interruption, loss of business information, or any other pecuniary loss) arising from the use of or inability to use the Book or the Software, even if HMI has been advised of the possibility of such damages.

(c) Because some jurisdictions do not allow the exclusion or limitation of liability for consequential or incidental damages, the above limitation or exclusion may not apply to you.

7. **U.S. Government Restricted Rights.** Use, duplication, or disclosure of the Software for or on behalf of the United States of America, its agencies and/or instrumentalities (the "U.S. Government") is subject to restrictions as stated in paragraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause of DFARS 252.227-7013, or subparagraphs (c) (1) and (2) of the Commercial Computer Software - Restricted Rights clause at FAR 52.227-19, and in similar clauses in the NASA FAR supplement, as applicable.

8. **General.** This Agreement constitutes the entire understanding of the parties and revokes and supersedes all prior agreements, oral or written, between them and may not be modified or amended except in a writing signed by both parties hereto that specifically refers to this Agreement. This Agreement shall take precedence over any other documents that may be in conflict herewith. If any one or more provisions contained in this Agreement are held by any court or tribunal to be invalid, illegal, or otherwise unenforceable, each and every other provision shall remain in full force and effect.

CD Installation Instructions

Each software item on the Oracle8i™DBA: SQL and PL/SQL Certification Bible CD-ROM is located in its own folder. To install a particular piece of software, open its folder with My Computer or Internet Explorer. What you do next depends on what you find in the software's folder:

1. First, look for a `ReadMe.txt` file or a `.doc` or `.htm` document. If this is present, it should contain installation instructions and other useful information.
2. If the folder contains an executable (`.exe`) file, this is usually an installation program. Often it will be called `Setup.exe` or `Install.exe`, but in some cases the filename reflects an abbreviated version of the software's name and version number. Run the `.exe` file to start the installation process.

The `ReadMe.txt` file in the CD-ROM's root directory may contain additional installation information, so be sure to check it.

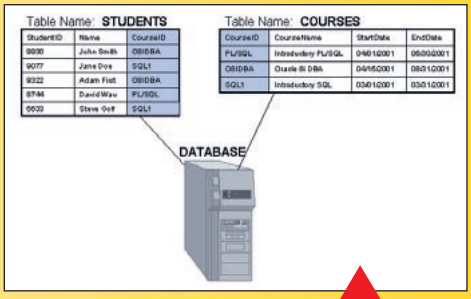
For a listing of the software on the CD-ROM, see Appendix A.

The only guide you need for Oracle8*i* DBA SQL and PL/SQL exam success . . .

You're holding in your hands the most comprehensive and effective guide available for the first exam on the Oracle8*i* DBA track. An outstanding team of database professionals and trainers delivers crystal-clear explanations of every topic covered on the exam, highlighting critical concepts and offering hands-on tips that can help you in your real-world DBA career.

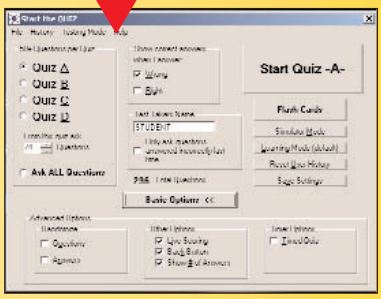
Get complete coverage of SQL and PL/SQL exam objectives

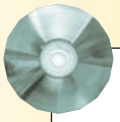
- Get a handle on Oracle database basics
- Retrieve data using basic SQL statements
- Find out how to use single and multi-row functions
- Delve into advanced select statements
- Create and manage Oracle database objects
- Get in-depth explanations of security configurations
- Understand PL/SQL basics
- Take control of program execution in PL/SQL
- Master PL/SQL database interactions
- Handle errors and exceptions in PL/SQL
- Get the scoop on stored programs



Figures and code examples illustrate critical concepts, such as how a join works

Customized test engine provides a different experience each time you take the exam.





Test-Prep Tools on CD-ROM

- Hungry Minds test engine powered by top-rated Boson Software
- Trial versions of Knowledge Base for Active PL/SQL and ER/Studio Rapid SQL and the Introduction to Oracle: SQL and PL/SQL Prep Exam
- Plus an e-version of the book

Boson Software

About the Authors

Damir Bersinic, an Oracle Certified DBA as well as an MCSE and MCDBA, has worked with Oracle and SQL Server for nearly two decades. He is the founder and president of Bradley Systems Inc., a Microsoft certified partner specializing in database, Internet, and system integration consulting and training. **Stephen Giles,** creator of a customized SQL course, **Susan Ibach,** an application developer, and **Myles Brown,** a database programming specialist, all teach the Oracle curriculum at TMI-Lernix, a leading training company based in Canada.

System Requirements: PC running Windows NT Service Pack 4 or later; 64 MB RAM. See the What's on the CD-ROM? Appendix for details and complete system requirements.

$59.99 USA
$89.99 Canada
£44.99 UK incl. VAT

Reader Level:
Beginning to Advanced

Shelving Category:
Certification

www.hungryminds.com

ISBN 0-7645-4832-8