Humber College

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Oracle Database 11*g*: PL/SQL Fundamentals

Student Guide

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide

D49990GC20
Edition 2.0
September 2009
D62728

ORACLE

Humber College

Author

Brian Pottle

Technical Contributors and Reviewers

Tom Best
Christoph Burandt
Yanti Chang
Laszlo Czinkoczki
Ashita Dhir
Peter Driver
Gerlinde Frenzen
Nancy Greenberg
Chaitanya Kortamaddi
Tim Leblanc
Bryan Roberts
Abhishek X Singh
Puja Singh
Lex Van Der Werff

Graphic Designer

Satish Bettegowda

Editors

Vijayalakshmi Narasimhan
Daniel Milne

Publisher

Jobi Varghese

Copyright © 2009, Oracle. All rights reserved.

Disclaimer

This document contains proprietary information and is protected by copyright and other intellectual property laws. You may copy and print this document solely for your own use in an Oracle training course. The document may not be modified or altered in any way. Except where your use constitutes "fair use" under copyright law, you may not use, share, download, upload, copy, print, display, perform, reproduce, publish, license, post, transmit, or distribute this document in whole or in part without the express authorization of Oracle.

The information contained in this document is subject to change without notice. If you find any problems in the document, please report them in writing to: Oracle University, 500 Oracle Parkway, Redwood Shores, California 94065 USA. This document is not warranted to be error-free.

Restricted Rights Notice

If this documentation is delivered to the United States Government or anyone using the documentation on behalf of the United States Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS
The U.S. Government's rights to use, modify, reproduce, release, perform, display, or disclose these training materials are restricted by the terms of the applicable Oracle license agreement and/or the applicable U.S. Government contract.

Trademark Notice

Oracle is a registered trademark of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Contents

I Introduction
Lesson Objectives I-2
Course Objectives I-3
Human Resources (HR) Schema for This Course I-4
Course Agenda I-5
Class Account Information I-6
Appendixes Used in This Course I-7
PL/SQL Development Environments I-8
What Is Oracle SQL Developer? I-9
Coding PL/SQL in SQL*Plus I-10
Coding PL/SQL in Oracle JDeveloper I-11
Oracle 11*g* SQL and PL/SQL Documentation I-12
Summary I-13
Practice I Overview: Getting Started I-14

1 Introduction to PL/SQL
Objectives 1-2
Agenda 1-3
About PL/SQL 1-4
PL/SQL Run-Time Architecture 1-6
Benefits of PL/SQL 1-7
PL/SQL Block Structure 1-10
Agenda 1-12
Block Types 1-13
Program Constructs 1-15
Examining an Anonymous Block 1-17
Executing an Anonymous Block 1-18
Agenda 1-19
Enabling Output of a PL/SQL Block 1-20
Viewing the Output of a PL/SQL Block 1-21
Quiz 1-22
Summary 1-23
Practice 1: Overview 1-24

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

2 Declaring PL/SQL Variables
Objectives 2-2
Agenda 2-3
Use of Variables 2-4
Requirements for Variable Names 2-5
Handling Variables in PL/SQL 2-6
Declaring and Initializing PL/SQL Variables 2-7
Delimiters in String Literals 2-9
Agenda 2-10
Types of Variables 2-11
Guidelines for Declaring and Initializing PL/SQL Variables 2-13
Guidelines for Declaring PL/SQL Variables 2-14
Naming Conventions of PL/SQL Structures Used in This Course 2-15
Scalar Data Types 2-16
Base Scalar Data Types 2-17
Declaring Scalar Variables 2-21
`%TYPE` Attribute 2-22
Declaring Variables with the `%TYPE` Attribute 2-24
Declaring Boolean Variables 2-25
`LOB` Data Type Variables 2-26
Composite Data Types: Records and Collections 2-27
Agenda 2-28
Bind Variables 2-29
Referencing Bind Variables 2-31
Using `AUTOPRINT` with Bind Variables 2-32
Quiz 2-33
Summary 2-34
Practice 2: Overview 2-35

3 Writing Executable Statements
Objectives 3-2
Agenda 3-3
Lexical Units in a PL/SQL Block 3-4
PL/SQL Block Syntax and Guidelines 3-6
Commenting Code 3-7
SQL Functions in PL/SQL 3-8
SQL Functions in PL/SQL: Examples 3-9
Using Sequences in PL/SQL Expressions 3-10
Data Type Conversion 3-11
Agenda 3-14

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Nested Blocks 3-15
Nested Blocks: Example 3-16
Variable Scope and Visibility 3-17
Using a Qualifier with Nested Blocks 3-19
Challenge: Determining Variable Scope 3-20
Agenda 3-22
Operators in PL/SQL 3-23
Operators in PL/SQL: Examples 3-24
Programming Guidelines 3-25
Indenting Code 3-26
Quiz 3-27
Summary 3-28
Practice 3: Overview 3-29

4 Interacting with Oracle Database Server: SQL Statements in PL/SQL Programs
Objectives 4-2
Agenda 4-3
SQL Statements in PL/SQL 4-4
`SELECT` Statements in PL/SQL 4-5
Retrieving Data in PL/SQL: Example 4-9
Retrieving Data in PL/SQL 4-10
Naming Ambiguities 4-11
Naming Conventions 4-12
Agenda 4-13
Using PL/SQL to Manipulate Data 4-14
Inserting Data: Example 4-15
Updating Data: Example 4-16
Deleting Data: Example 4-17
Merging Rows 4-18
Agenda 4-20
SQL Cursor 4-21
SQL Cursor Attributes for Implicit Cursors 4-23
Quiz 4-25
Summary 4-26
Practice 4: Overview 4-27

5 Writing Control Structures
Objectives 5-2
Controlling Flow of Execution 5-3
Agenda 5-4
`IF` Statement 5-5

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Simple IF Statement 5-7
IF THEN ELSE Statement 5-8
IF ELSIF ELSE Clause 5-9
NULL Value in IF Statement 5-10
Agenda 5-11
CASE Expressions 5-12
CASE Expressions: Example 5-13
Searched CASE Expressions 5-14
CASE Statement 5-15
Handling Nulls 5-16
Logic Tables 5-17
Boolean Expressions or Logical Expression? 5-18
Agenda 5-19
Iterative Control: LOOP Statements 5-20
Basic Loops 5-21
Basic Loop: Example 5-22
WHILE Loops 5-23
WHILE Loops: Example 5-24
FOR Loops 5-25
FOR Loops: Example 5-27
FOR Loop Rules 5-28
Suggested Use of Loops 5-29
Nested Loops and Labels 5-30
Nested Loops and Labels: Example 5-31
PL/SQL CONTINUE Statement 5-32
PL/SQL CONTINUE Statement: Example 1 5-33
PL/SQL CONTINUE Statement: Example 2 5-34
Quiz 5-35
Summary 5-36
Practice 5: Overview 5-37

6 Working with Composite Data Types
Objectives 6-2
Agenda 6-3
Composite Data Types 6-4
PL/SQL Records or Collections? 6-5
Agenda 6-6
PL/SQL Records 6-7
Creating a PL/SQL Record 6-8
PL/SQL Record Structure 6-9

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

%ROWTYPE Attribute 6-10
Creating a PL/SQL Record: Example 6-12
Advantages of Using the %ROWTYPE Attribute 6-13
Another %ROWTYPE Attribute Example 6-14
Inserting a Record by Using %ROWTYPE 6-15
Updating a Row in a Table by Using a Record 6-16
Agenda 6-17
Associative Arrays (INDEX BY Tables) 6-18
Associative Array Structure 6-19
Steps to Create an Associative Array 6-20
Creating and Accessing Associative Arrays 6-21
Using INDEX BY Table Methods 6-22
INDEX BY Table of Records Option 6-23
INDEX BY Table of Records Option: Example 2 6-24
Nested Tables 6-25
VARRAY 6-27
Summary of Collection Types 6-28
Quiz 6-29
Summary 6-30
Practice 6: Overview 6-31

7 Using Explicit Cursors
Objectives 7-2
Agenda 7-3
Cursors 7-4
Explicit Cursor Operations 7-5
Controlling Explicit Cursors 7-6
Agenda 7-8
Declaring the Cursor 7-9
Opening the Cursor 7-11
Fetching Data from the Cursor 7-12
Closing the Cursor 7-15
Cursors and Records 7-16
Cursor FOR Loops 7-17
Explicit Cursor Attributes 7-19
%ISOPEN Attribute 7-20
%ROWCOUNT and %NOTFOUND: Example 7-21
Cursor FOR Loops Using Subqueries 7-22
Agenda 7-23
Cursors with Parameters 7-24

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Agenda 7-26
`FOR UPDATE` Clause 7-27
`WHERE CURRENT OF` Clause 7-29
Quiz 7-30
Summary 7-31
Practice 7: Overview 7-32

8 Handling Exceptions
Objectives 8-2
Agenda 8-3
What Is an Exception? 8-4
Handling the Exception: An Example 8-5
Understanding Exceptions with PL/SQL 8-6
Handling Exceptions 8-7
Exception Types 8-8
Agenda 8-9
Syntax to Trap Exceptions 8-10
Guidelines for Trapping Exceptions 8-12
Trapping Predefined Oracle Server Errors 8-13
Trapping Non-Predefined Oracle Server Errors 8-16
Non-Predefined Error Trapping: Example 8-17
Functions for Trapping Exceptions 8-18
Trapping User-Defined Exceptions 8-20
Propagating Exceptions in a Subblock 8-22
`RAISE_APPLICATION_ERROR` Procedure 8-23
Quiz 8-26
Summary 8-27
Practice 8: Overview 8-28

9 Introducing Stored Procedures and Functions
Objectives 9-2
Agenda 9-3
Procedures and Functions 9-4
Differences Between Anonymous Blocks and Subprograms 9-5
Agenda 9-6
Procedure: Syntax 9-7
Creating a Procedure 9-8
Invoking a Procedure 9-10
Agenda 9-11
Function: Syntax 9-12

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapatiankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating a Function 9-13
Invoking a Function 9-14
Passing a Parameter to the Function 9-15
Invoking the Function with a Parameter 9-16
Quiz 9-17
Summary 9-18
Practice 9: Overview 9-19

A Practices and Solutions

B Table Descriptions and Data

C Using SQL Developer
Objectives C-2
What Is Oracle SQL Developer? C-3
Specifications of SQL Developer C-4
SQL Developer 1.5 Interface C-5
Creating a Database Connection C-7
Browsing Database Objects C-10
Displaying the Table Structure C-11
Browsing Files C-12
Creating a Schema Object C-13
Creating a New Table: Example C-14
Using the SQL Worksheet C-15
Executing SQL Statements C-18
Saving SQL Scripts C-19
Executing Saved Script Files: Method 1 C-20
Executing Saved Script Files: Method 2 C-21
Formatting the SQL Code C-22
Using Snippets C-23
Using Snippets: Example C-24
Debugging Procedures and Functions C-25
Database Reporting C-26
Creating a User-Defined Report C-27
Search Engines and External Tools C-28
Setting Preferences C-29
Resetting the SQL Developer Layout C-30
Summary C-31

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

D Using SQL*Plus
Objectives D-2
SQL and SQL*Plus Interaction D-3
SQL Statements Versus SQL*Plus Commands D-4
SQL*Plus: Overview D-5
Logging In to SQL*Plus D-6
Displaying the Table Structure D-7
SQL*Plus Editing Commands D-9
Using `LIST`, `n`, and `APPEND` D-11
Using the `CHANGE` Command D-12
SQL*Plus File Commands D-13
Using the `SAVE` and `START` Commands D-14
`SERVEROUTPUT` Command D-15
Using the SQL*Plus `SPOOL` Command D-16
Using the `AUTOTRACE` Command D-17
Summary D-18

E Using JDeveloper
Oracle JDeveloper E-2
Database Navigator E-3
Creating a Connection E-4
Browsing Database Objects E-5
Executing SQL Statements E-6
Creating Program Units E-7
Compiling E-8
Running a Program Unit E-9
Dropping a Program Unit E-10
Structure Window E-11
Editor Window E-12
Application Navigator E-13
Deploying Java Stored Procedures E-14
Publishing Java to PL/SQL E-15
How Can I Learn More About JDeveloper 11*g*? E-16

F `REF` Cursors
Cursor Variables F-2
Using Cursor Variables F-3
Defining `REF CURSOR` Types F-4
Using the `OPEN-FOR`, `FETCH`, and `CLOSE` Statements F-7
Example of Fetching F-10

Additional Practices and Solutions

Ankur N. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Humber College

Introduction

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Lesson Objectives

After completing this lesson, you should be able to do the following:

- Discuss the goals of the course
- Describe the HR database schema that is used in the course
- Identify the available user interface environments that can be used in this course
- Reference the available appendixes, documentation, and other resources

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Lesson Objectives

This lesson gives you a high-level overview of the course and its flow. You learn about the database schema and the tables that the course uses. You are also introduced to different products in the Oracle 11*g* grid infrastructure.

Course Objectives

After completing this course, you should be able to do the following:

- Identify the programming extensions that PL/SQL provides to SQL
- Write PL/SQL code to interface with the database
- Design PL/SQL anonymous blocks that execute efficiently
- Use PL/SQL programming constructs and conditional control statements
- Handle run-time errors
- Describe stored procedures and functions

ORACLE

Copyright © 2009, Oracle. All rights reserved.

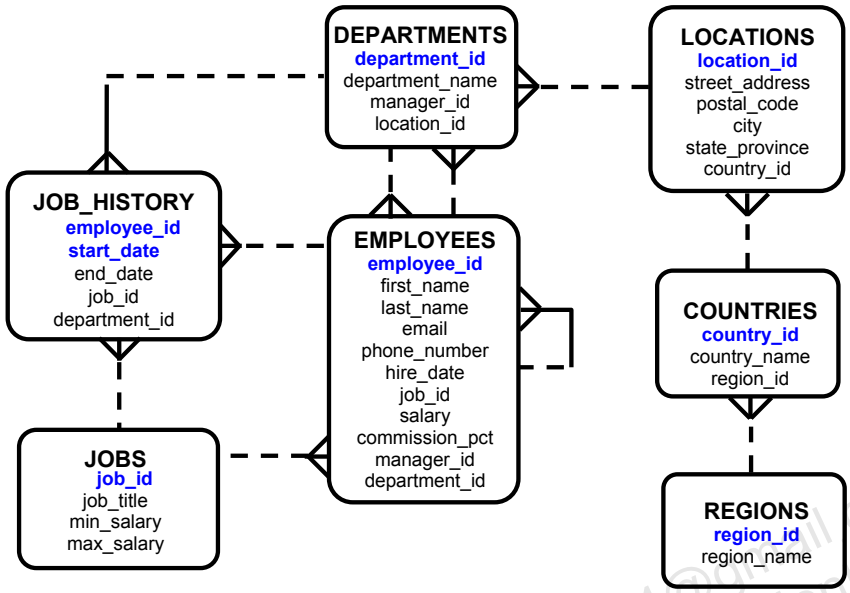
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Course Objectives

This course presents the basics of PL/SQL. You learn about PL/SQL syntax, blocks, and programming constructs and also about the advantages of integrating SQL with those constructs. You learn how to write PL/SQL program units and execute them efficiently. In addition, you learn how to use SQL Developer as a development environment for PL/SQL. You also learn how to design reusable program units such as procedures and functions.

Human Resources (HR) Schema for This Course



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Human Resources (HR) Schema for This Course

The Human Resources (HR) schema is part of the Oracle Sample Schemas that can be installed in an Oracle database. The practice sessions in this course use data from the HR schema.

Table Descriptions

- REGIONS contains rows that represent a region such as the Americas or Asia.
- COUNTRIES contains rows for countries, each of which is associated with a region.
- LOCATIONS contains the specific address of a specific office, warehouse, or production site of a company in a particular country.
- DEPARTMENTS shows details about the departments in which employees work. Each department may have a relationship representing the department manager in the EMPLOYEES table.
- EMPLOYEES contains details about each employee working for a department. Some employees may not be assigned to any department.
- JOBS contains the job types that can be held by each employee.
- JOB_HISTORY contains the job history of the employees. If an employee changes departments within a job or changes jobs within a department, a new row is inserted into this table with the old job information of the employee.

Course Agenda

Day 1:

I. Introduction

1. Introduction to PL/SQL
2. Declaring PL/SQL Variables
3. Writing Executable Statements
4. Interacting with Oracle Database Server: SQL Statements in PL/SQL Programs
5. Writing Control Structures

Day 2:

6. Working with Composite Data Types
7. Using Explicit Cursors
8. Handling Exceptions
9. Introducing Stored Procedures and Functions

ORACLE

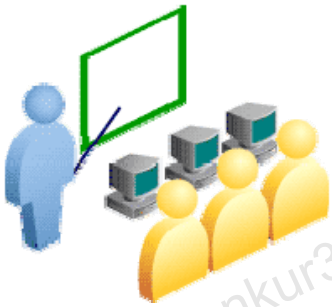
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Class Account Information

- A cloned `HR` account ID is set up for you.
- Your account ID is `ora41`.
- The password matches your account ID.
- Each machine has its own complete environment, and is assigned the same account.
- The instructor has a separate ID.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Appendixes Used in This Course

- Appendix A: Practices and Solutions
- Appendix B: Table Descriptions and Data
- Appendix C: Using SQL Developer
- Appendix D: Using SQL*Plus
- Appendix E: Using JDeveloper
- Appendix F: REF Cursors
- Appendix AP: Additional Practices and Solutions

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

PL/SQL Development Environments

This course setup provides the following tools for developing PL/SQL code:

- Oracle SQL Developer (used in this course)
- Oracle SQL*Plus
- Oracle JDeveloper IDE

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

PL/SQL Development Environments

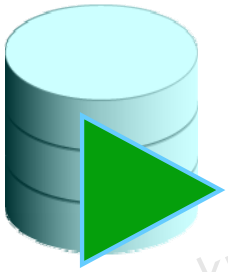
Oracle provides several tools that can be used to write PL/SQL code. Some of the development tools that are available for use in this course:

- **Oracle SQL Developer:** A graphical tool
- **Oracle SQL*Plus:** A window or command-line application
- **Oracle JDeveloper:** A window-based integrated development environment (IDE)

Note: The code and screen examples presented in the course notes were generated from output in the SQL Developer environment.

What Is Oracle SQL Developer?

- Oracle SQL Developer is a free graphical tool that enhances productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema using standard Oracle database authentication.
- You will use SQL Developer in this course.
- Appendix C contains details on using SQL Developer.



SQL Developer

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

What Is Oracle SQL Developer?

Oracle SQL Developer is a free graphical tool designed to improve your productivity and simplify the development of everyday database tasks. With just a few clicks, you can easily create and maintain stored procedures, test SQL statements, and view optimizer plans.

SQL Developer, the visual tool for database development, simplifies the following tasks:

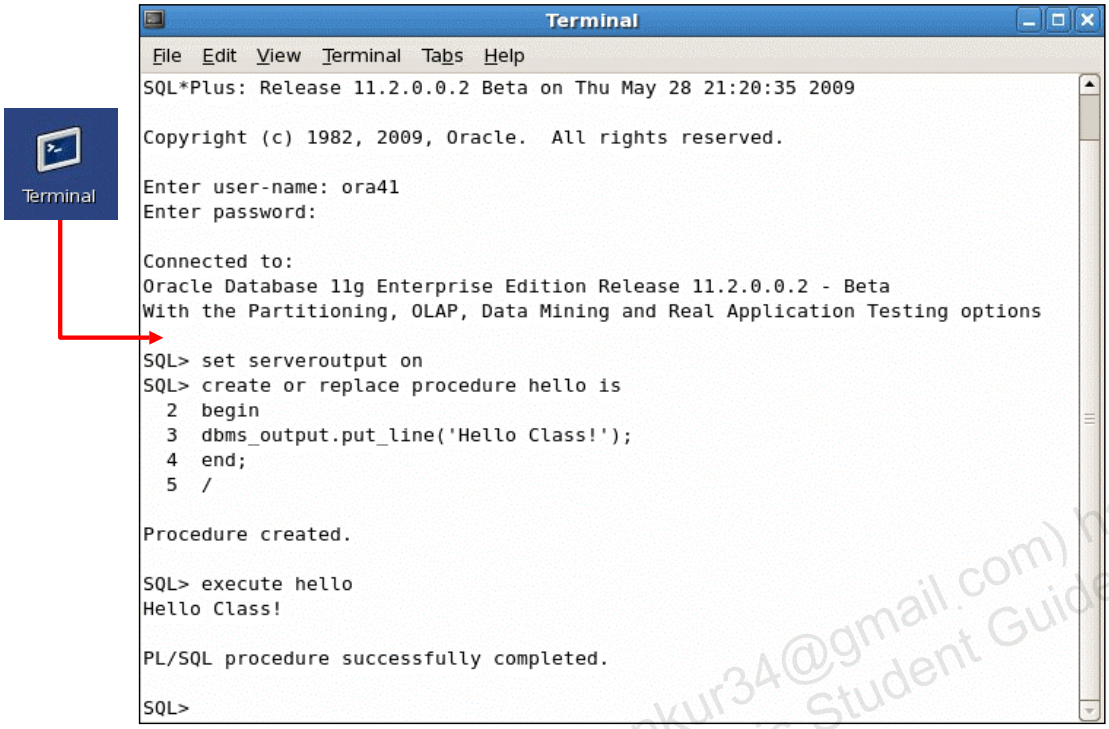
- Browsing and managing database objects
- Executing SQL statements and scripts
- Editing and debugging PL/SQL statements
- Creating reports

You can connect to any target Oracle database schema by using standard Oracle database authentication. When you are connected, you can perform operations on objects in the database.

Appendix C

Appendix C of this course provides an introduction on using the SQL Developer interface. Refer to the appendix for information about creating a database connection, interacting with data using SQL and PL/SQL, and more.

Coding PL/SQL in SQL*Plus



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Coding PL/SQL in SQL*Plus

Oracle SQL*Plus is a command-line interface that enables you to submit SQL statements and PL/SQL blocks for execution and receive the results in an application or a command window.

SQL*Plus is:

- Shipped with the database
- Installed on a client and on the database server system
- Accessed using an icon or the command line

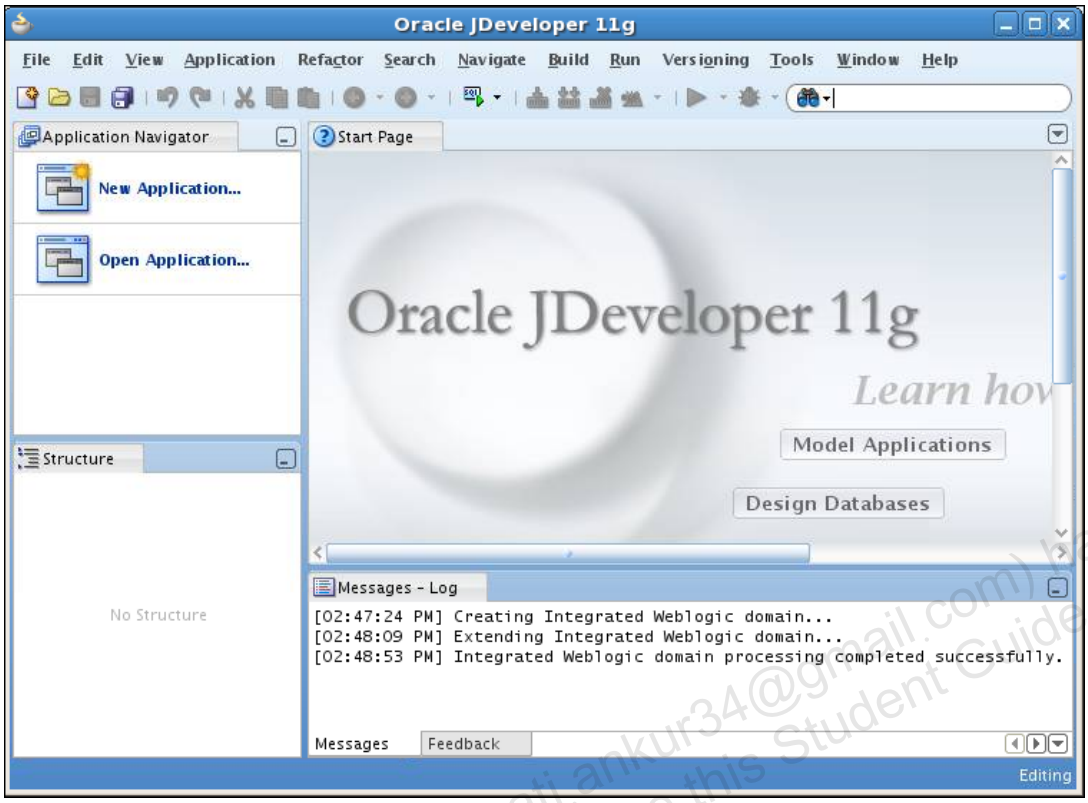
When you code PL/SQL subprograms using SQL*Plus, remember the following:

- You create subprograms by using the `CREATE SQL` statement.
- You execute subprograms by using either an anonymous PL/SQL block or the `EXECUTE` command.
- If you use the `DBMS_OUTPUT` package procedures to print text to the screen, you must first execute the `SET SERVEROUTPUT ON` command in your session.

Note

- To launch SQL*Plus in Linux environment, open a Terminal window and enter the command: `sqlplus`.
- For more information about using SQL*Plus, see Appendix D.

Coding PL/SQL in Oracle JDeveloper



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Coding PL/SQL in Oracle JDeveloper

Oracle JDeveloper allows developers to create, edit, test, and debug PL/SQL code by using a sophisticated GUI. Oracle JDeveloper is a part of Oracle Developer Suite and is also available as a separate product.

When you code PL/SQL in JDeveloper, consider the following:

- You first create a database connection to enable JDeveloper to access a database schema owner for the subprograms.
- You can then use the JDeveloper context menus on the Database connection to create a new subprogram construct using the built-in JDeveloper Code Editor.
- You invoke a subprogram by using a `Run` command on the context menu for the named subprogram. The output appears in the JDeveloper Log Message window, as shown in the lower portion of the screenshot.

Note

- JDeveloper provides color-coding syntax in the JDeveloper Code Editor and is sensitive to PL/SQL language constructs and statements.
- For more information about using JDeveloper, see Appendix E.

Oracle 11*g* SQL and PL/SQL Documentation

- *Oracle Database New Features Guide 11g Release 2 (11.2)*
- *Oracle Database Advanced Application Developer's Guide 11g Release 2 (11.2)*
- *Oracle Database PL/SQL Language Reference 11g Release 2 (11.2)*
- *Oracle Database Reference 11g Release 2 (11.2)*
- *Oracle Database SQL Language Reference 11g Release 2 (11.2)*
- *Oracle Database Concepts 11g Release 2 (11.2)*
- *Oracle Database PL/SQL Packages and Types Reference 11g Release 2 (11.2)*
- *Oracle Database SQL Developer User's Guide Release 1.5*

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide

Summary

In this lesson, you should have learned how to:

- Discuss the goals of the course
- Describe the `HR` database schema that is used in the course
- Identify the available user interface environments that can be used in this course
- Reference the available appendixes, documentation, and other resources

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Practice I Overview: Getting Started

This practice covers the following topics:

- Starting SQL Developer
- Creating a new database connection
- Browsing the HR schema tables
- Setting a SQL Developer preference

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a nontransferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Practice I: Overview

In this practice, you use SQL Developer to execute SQL statements to examine data in the HR schema. You also create a simple anonymous block.

Note: All written practices use SQL Developer as the development environment. Although it is recommended that you use SQL Developer, you can also use the SQL*Plus or JDeveloper environments that are available in this course.

1

Introduction to PL/SQL

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (rajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Objectives

After completing this lesson, you should be able to do the following:

- Explain the need for PL/SQL
- Explain the benefits of PL/SQL
- Identify the different types of PL/SQL blocks
- Output messages in PL/SQL

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

This lesson introduces PL/SQL and the PL/SQL programming constructs. You also learn about the benefits of PL/SQL.

Agenda

- Understanding the benefits and structure of PL/SQL
- Examining PL/SQL blocks
- Generating output messages in PL/SQL

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

About PL/SQL

PL/SQL:

- Stands for “Procedural Language extension to SQL”
- Is Oracle Corporation’s standard data access language for relational databases
- Seamlessly integrates procedural constructs with SQL

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

About PL/SQL

Structured Query Language (SQL) is the primary language used to access and modify data in relational databases. There are only a few SQL commands, so you can easily learn and use them.

Consider an example:

```
SELECT first_name, department_id, salary FROM employees;
```

The preceding SQL statement is simple and straightforward. However, if you want to alter any data that is retrieved in a conditional manner, you soon encounter the limitations of SQL.

Consider a slightly modified problem statement: For every employee retrieved, check the department ID and salary. Depending on the department’s performance and also the employee’s salary, you may want to provide varying bonuses to the employees.

Looking at the problem, you know that you have to execute the preceding SQL statement, collect the data, and apply logic to the data.

- One solution is to write a SQL statement for each department to give bonuses to the employees in that department. Remember that you also have to check the salary component before deciding the bonus amount. This makes it a little complicated.
- A more effective solution might include conditional statements. PL/SQL is designed to meet such requirements. It provides a programming extension to the already-existing SQL.

About PL/SQL

PL/SQL:

- Provides a block structure for executable units of code. Maintenance of code is made easier with such a well-defined structure.
- Provides procedural constructs such as:
  - Variables, constants, and data types
  - Control structures such as conditional statements and loops
  - Reusable program units that are written once and executed many times

ORACLE

Copyright © 2009, Oracle. All rights reserved.

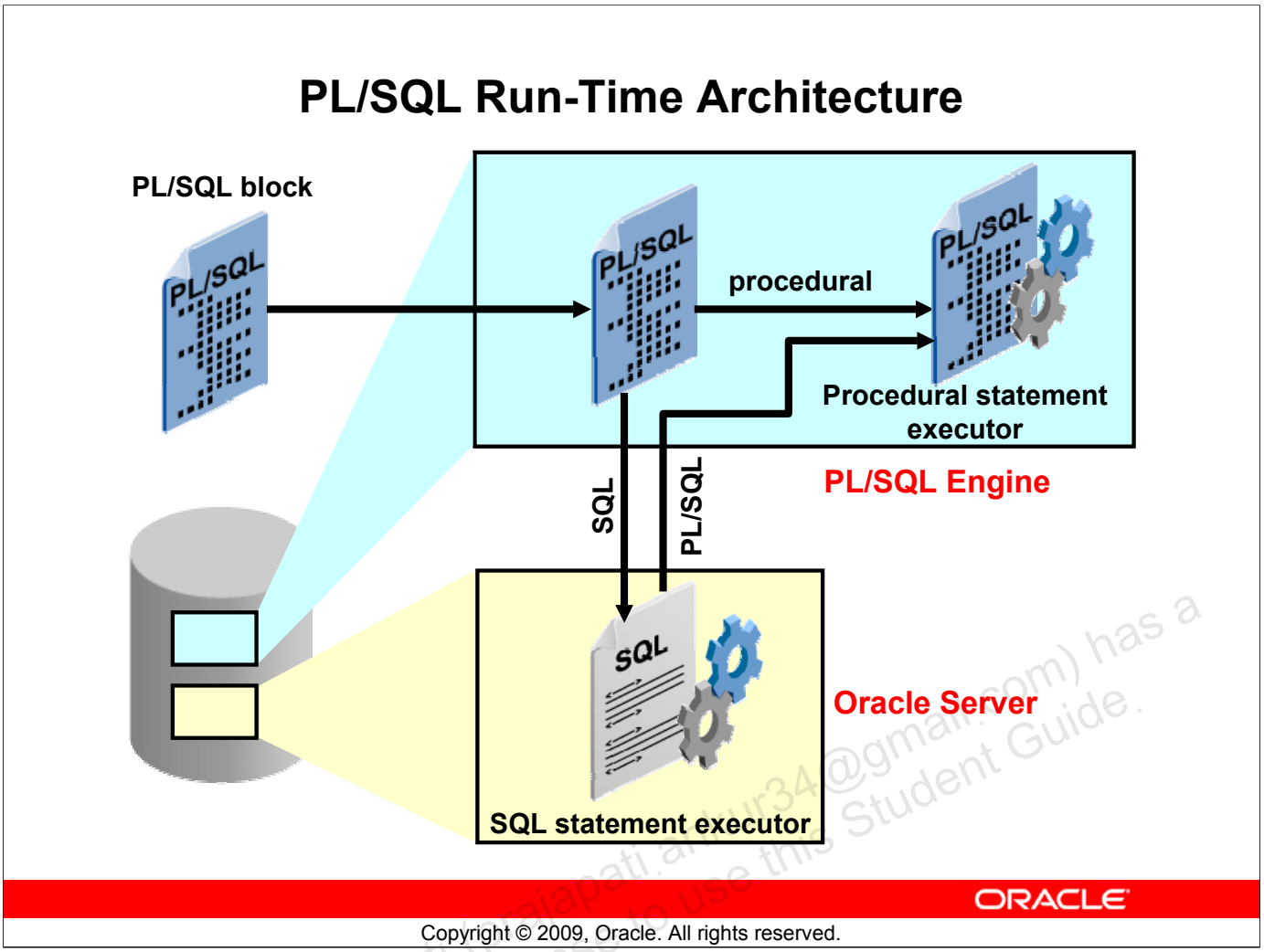
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

About PL/SQL (continued)

PL/SQL defines a block structure for writing code. Maintaining and debugging code is made easier with such a structure because you can easily understand the flow and execution of the program unit.

PL/SQL offers modern software engineering features such as data encapsulation, exception handling, information hiding, and object orientation. It brings state-of-the-art programming to the Oracle Server and toolset. PL/SQL provides all the procedural constructs that are available in any third-generation language (3GL).



PL/SQL Run-Time Architecture

The diagram in the slide shows a PL/SQL block being executed by the PL/SQL engine. The PL/SQL engine resides in:

- The Oracle database for executing stored subprograms
- The Oracle Forms client when you run client/server applications, or in the Oracle Application Server when you use Oracle Forms Services to run Forms on the Web

Irrespective of the PL/SQL run-time environment, the basic architecture remains the same. Therefore, all PL/SQL statements are processed in the Procedural Statement Executor, and all SQL statements must be sent to the SQL Statement Executor for processing by the Oracle Server processes. The SQL environment may also invoke the PL/SQL environment. For example, the PL/SQL environment is invoked when a PL/SQL function is used in a `SELECT` statement.

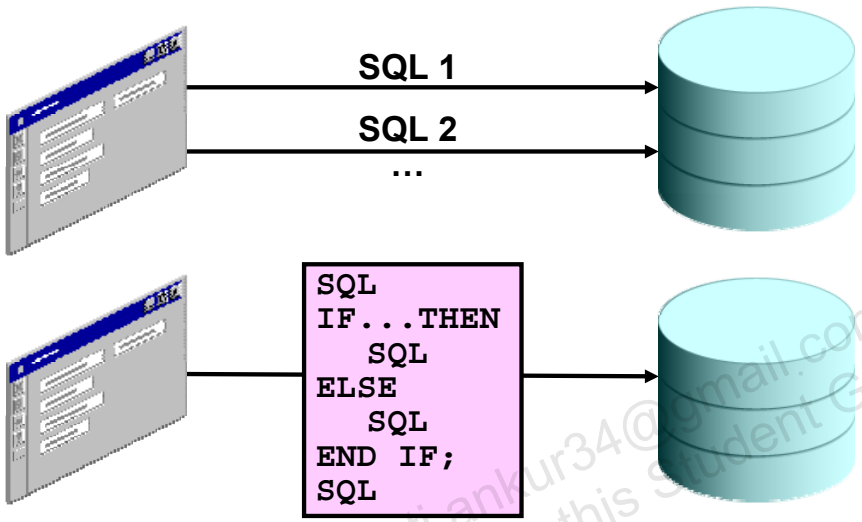
The PL/SQL engine is a virtual machine that resides in memory and processes the PL/SQL m-code instructions. When the PL/SQL engine encounters a SQL statement, a context switch is made to pass the SQL statement to the Oracle Server processes. The PL/SQL engine waits for the SQL statement to complete and for the results to be returned before it continues to process subsequent statements in the PL/SQL block. The Oracle Forms PL/SQL engine runs in the client for the client/server implementation, and in the application server for the Forms Services implementation. In either case, SQL statements are typically sent over a network to an Oracle Server for processing.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Rakesh M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Benefits of PL/SQL

- Integration of procedural constructs with SQL
- Improved performance



ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Benefits of PL/SQL

Integration of procedural constructs with SQL: The most important advantage of PL/SQL is the integration of procedural constructs with SQL. SQL is a nonprocedural language. When you issue a SQL command, your command tells the database server *what* to do. However, you cannot specify *how* to do it. PL/SQL integrates control statements and conditional statements with SQL, giving you better control of your SQL statements and their execution. Earlier in this lesson, you saw an example of the need for such integration.

Improved performance: Without PL/SQL, you would not be able to logically combine SQL statements as one unit. If you have designed an application that contains forms, you may have many different forms with fields in each form. When a form submits data, you may have to execute a number of SQL statements. SQL statements are sent to the database one at a time. This results in many network trips and one call to the database for each SQL statement, thereby increasing network traffic and reducing performance (especially in a client/server model).

With PL/SQL, you can combine all these SQL statements into a single program unit. The application can send the entire block to the database instead of sending the SQL statements one at a time. This significantly reduces the number of database calls. As the slide illustrates, if the application is SQL intensive, you can use PL/SQL blocks to group SQL statements before sending them to the Oracle database server for execution.

Benefits of PL/SQL

- Modularized program development
- Integration with Oracle tools
- Portability
- Exception handling

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Benefits of PL/SQL (continued)

Modularized program development: The basic unit in all PL/SQL programs is the block. Blocks can be in a sequence or they can be nested in other blocks. Modularized program development has the following advantages:

- You can group logically related statements within blocks.
- You can nest blocks inside larger blocks to build powerful programs.
- You can break your application into smaller modules. If you are designing a complex application, PL/SQL allows you to break down the application into smaller, manageable, and logically related modules.
- You can easily maintain and debug code.

In PL/SQL, modularization is implemented using procedures, functions, and packages, which are discussed in the lesson titled “Introducing Stored Procedures and Functions.”

Integration with tools: The PL/SQL engine is integrated in Oracle tools such as Oracle Forms and Oracle Reports. When you use these tools, the locally available PL/SQL engine processes the procedural statements; only the SQL statements are passed to the database.

Benefits of PL/SQL (continued)

Portability: PL/SQL programs can run anywhere an Oracle Server runs, irrespective of the operating system and platform. You do not need to customize them to each new environment. You can write portable program packages and create libraries that can be reused in different environments.

Exception handling: PL/SQL enables you to handle exceptions efficiently. You can define separate blocks for dealing with exceptions. You learn more about exception handling in the lesson titled "Handling Exceptions."

PL/SQL shares the same data type system as SQL (with some extensions) and uses the same expression syntax.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

PL/SQL Block Structure

- DECLARE (optional)
  - Variables, cursors, user-defined exceptions
- BEGIN (mandatory)
  - SQL statements
  - PL/SQL statements
- EXCEPTION (optional)
  - Actions to perform when exceptions occur
- END; (mandatory)

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

PL/SQL Block Structure

The slide shows a basic PL/SQL block. A PL/SQL block consists of four sections:

- **Declarative (optional):** The declarative section begins with the keyword DECLARE and ends when the executable section starts.
- **Begin (required):** The executable section begins with the keyword BEGIN. This section needs to have at least one statement. However, the executable section of a PL/SQL block can include any number of PL/SQL blocks.
- **Exception handling (optional):** The exception section is nested within the executable section. This section begins with the keyword EXCEPTION.
- **End (required)**: All PL/SQL blocks must conclude with an END statement. Observe that END is terminated with a semicolon.

PL/SQL Block Structure (continued)

In a PL/SQL block, the keywords `DECLARE`, `BEGIN`, and `EXCEPTION` are not terminated by a semicolon. However, the keyword `END`, all SQL statements, and PL/SQL statements must be terminated with a semicolon.

Section	Description	Inclusion
Declarative (`DECLARE`)	Contains declarations of all variables, constants, cursors, and user-defined exceptions that are referenced in the executable and exception sections	Optional
Executable (`BEGIN` … `END`)	Contains SQL statements to retrieve data from the database; contains PL/SQL statements to manipulate data in the block	Mandatory
Exception (`EXCEPTION`)	Specifies the actions to perform when errors and abnormal conditions arise in the executable section	Optional

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Agenda

- Understanding the benefits and structure of PL/SQL
- Examining PL/SQL blocks
- Generating output messages in PL/SQL

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Block Types

Procedure	Function	Anonymous

`PROCEDURE name`
`IS`
 	
`BEGIN`
`  --statements`
 	
`[EXCEPTION]`
 	
`END;`
	`FUNCTION name`
	`RETURN datatype`
	`IS`
	`BEGIN`
	`  --statements`
	`  RETURN value;`
	`[EXCEPTION]`
	
	`END;`
		`[DECLARE]`
		
		`BEGIN`
		`  --statements`
		
		`[EXCEPTION]`
		
		`END;`

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Block Types

A PL/SQL program comprises one or more blocks. These blocks can be entirely separate or nested within another block.

There are three types of blocks that make up a PL/SQL program:

- Procedures
- Functions
- Anonymous blocks

Procedures: Procedures are named objects that contain SQL and/or PL/SQL statements.

Functions: Functions are named objects that contain SQL and/or PL/SQL statements. Unlike a procedure, a function returns a value of a specified data type.

Anonymous blocks

Anonymous blocks are unnamed blocks. They are declared inline at the point in an application where they are to be executed and are compiled each time the application is executed. These blocks are not stored in the database. They are passed to the PL/SQL engine for execution at run time. Triggers in Oracle Developer components consist of such blocks.

If you want to execute the same block again, you have to rewrite the block. You cannot invoke or call the block that you wrote earlier because blocks are anonymous and do not exist after they are executed.

Block Types (continued)

Subprograms

Subprograms are complementary to anonymous blocks. They are named PL/SQL blocks that are stored in the database. Because they are named and stored, you can invoke them whenever you want (depending on your application). You can declare them either as procedures or as functions. You typically use a procedure to perform an action and a function to compute and return a value.

Subprograms can be stored at the server or application level. Using Oracle Developer components (Forms, Reports), you can declare procedures and functions as part of the application (a form or report) and call them from other procedures, functions, and triggers within the same application, whenever necessary.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Program Constructs

Tools Constructs
Anonymous blocks
Application procedures or functions
Application packages
Application triggers
Object types

Database Server Constructs
Anonymous blocks
Stored procedures or functions
Stored packages
Database triggers
Object types

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur54@gmail.com) has a non-transferable license to use this Student Guide.

Program Constructs

The following table outlines a variety of PL/SQL program constructs that use the basic PL/SQL block. The program constructs are available based on the environment in which they are executed.

Program Construct	Description	Availability
Anonymous blocks	Unnamed PL/SQL blocks that are embedded within an application or are issued interactively	All PL/SQL environments
Application procedures or functions	Named PL/SQL blocks that are stored in an Oracle Forms Developer application or a shared library; can accept parameters and can be invoked repeatedly by name	Oracle Developer tools components (for example, Oracle Forms Developer, Oracle Reports)
Stored procedures or functions	Named PL/SQL blocks that are stored in the Oracle server; can accept parameters and can be invoked repeatedly by name	Oracle server or Oracle Developer tools
Packages (application or stored)	Named PL/SQL modules that group related procedures, functions, and identifiers	Oracle server and Oracle Developer tools components (for example, Oracle Forms Developer)

Program Constructs (continued)

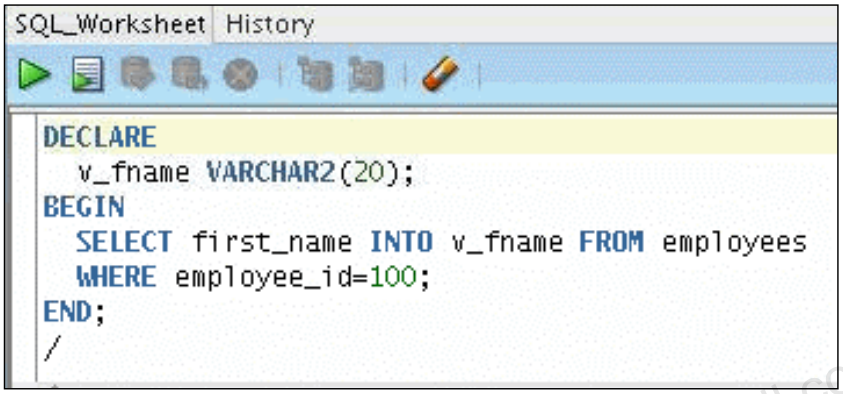
Program Construct	Description	Availability
Database triggers	PL/SQL blocks that are associated with a database table and are fired automatically when triggered by various events	Oracle server or any Oracle tool that issues the DML
Application triggers	PL/SQL blocks that are associated either with a database table or system events. They are fired automatically when triggered by a DML or a system event respectively.	Oracle Developer tools components (for example, Oracle Forms Developer)
Object types	User-defined composite data types that encapsulate a data structure along with the functions and procedures needed to manipulate data	Oracle server and Oracle Developer tools

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Examining an Anonymous Block

An anonymous block in the SQL Developer workspace:



Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Examining an Anonymous Block

To create an anonymous block by using SQL Developer, enter the block in the workspace (as shown in the slide).

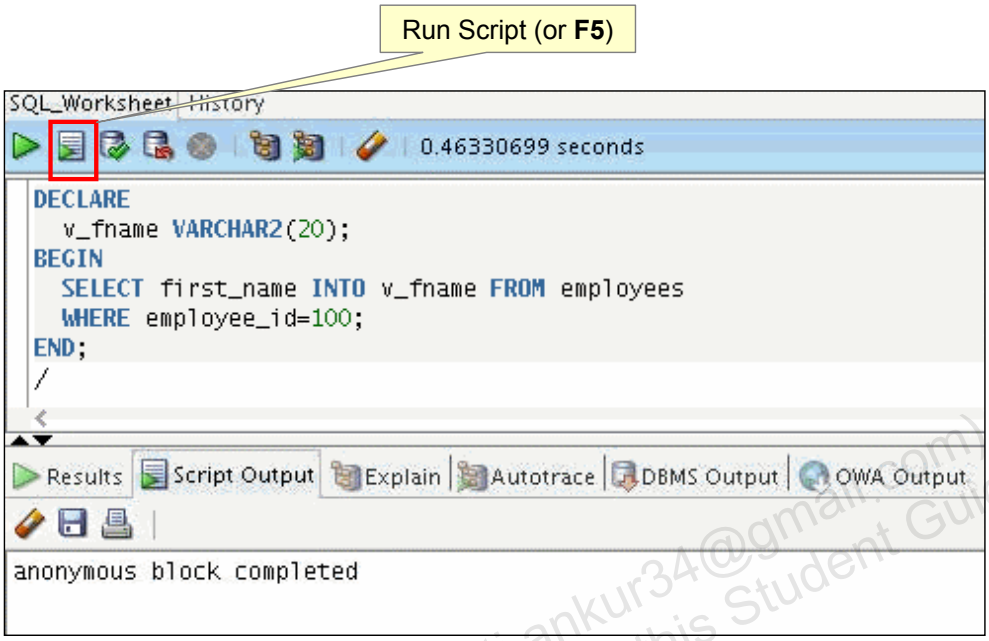
Example

The example block has the declarative section and the executable section. You need not pay attention to the syntax of statements in the block; you learn the syntax later in the course.

The anonymous block gets the `first_name` of the employee whose `employee_id` is `100`, and stores it in a variable called `v_fname`.

Executing an Anonymous Block

Click the Run Script button to execute the anonymous block:



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Executing an Anonymous Block

To execute an anonymous block, click the Run Script button (or press F5).

Note: The message "anonymous block completed" is displayed in the Script Output window after the block is executed.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Agenda

- Understanding the benefits and structure of PL/SQL
- Examining PL/SQL blocks
- Generating output messages in PL/SQL

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Enabling Output of a PL/SQL Block

1. To enable output in SQL Developer, execute the following command before running the PL/SQL block:

```
SET SERVEROUTPUT ON
```

2. Use a predefined Oracle package and its procedure in the anonymous block:
   - DBMS_OUTPUT.PUT_LINE

```
DBMS_OUTPUT.PUT_LINE(' The First Name of the
Employee is ' || v_fname);
…
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Enabling Output of a PL/SQL Block

In the example shown in the previous slide, a value is stored in the `v_fname` variable. However, the value has not been printed.

PL/SQL does not have built-in input or output functionality. Therefore, you need to use predefined Oracle packages for input and output. To generate output, you must perform the following:

1. Execute the following command:

   ```
   SET SERVEROUTPUT ON
   ```

   Note: To enable output in SQL*Plus, you must explicitly issue the `SET SERVEROUTPUT ON` command.

2. In the PL/SQL block, use the `PUT_LINE` procedure of the `DBMS_OUTPUT` package to display the output. Pass the value that has to be printed as an argument to this procedure (as shown in the slide). The procedure then outputs the argument.

Viewing the Output of a PL/SQL Block

SQL_Worksheet | History

0.14278294 seconds

```
SET SERVEROUTPUT ON

DECLARE
  v_fname VARCHAR(20);
BEGIN
  SELECT first_name
  INTO v_fname
  FROM employees
  WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE(' The First Name of the Employee is ' || v_fname);
END;
/
```

Press **F5** to execute the command and PL/SQL block.

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
anonymous block completed
 The First Name of the Employee is Steven
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Viewing the Output of a PL/SQL Block

Press F5 (or click the Run Script icon) to view the output for the PL/SQL block. This action:

1. Executes the `SET SERVEROUTPUT ON` command
2. Runs the anonymous PL/SQL block

The output appears on the Script Output tab.

Quiz

A PL/SQL block *must* consist of the following three sections:

- A Declarative section, which begins with the keyword `DECLARE` and ends when the executable section starts.
- An Executable section, which begins with the keyword `BEGIN` and ends with `END`.
- An Exception handling section, which begins with the keyword `EXCEPTION` and is nested within the executable section.

1. True
2. False

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Answer: 2

A PL/SQL block consists of three sections:

- **Declarative (optional):** The optional declarative section begins with the keyword `DECLARE` and ends when the executable section starts.
- **Executable (required):** The required executable section begins with the keyword `BEGIN` and ends with `END`. This section essentially needs to have at least one statement. Observe that `END` is terminated with a semicolon. The executable section of a PL/SQL block can, in turn, include any number of PL/SQL blocks.
- **Exception handling (optional):** The optional exception section is nested within the executable section. This section begins with the keyword `EXCEPTION`.

Summary

In this lesson, you should have learned how to:

- Integrate SQL statements with PL/SQL program constructs
- Describe the benefits of PL/SQL
- Differentiate between PL/SQL block types
- Output messages in PL/SQL

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Summary

PL/SQL is a language that has programming features that serve as extensions to SQL. SQL, which is a nonprocedural language, is made procedural with PL/SQL programming constructs. PL/SQL applications can run on any platform or operating system on which an Oracle Server runs. In this lesson, you learned how to build basic PL/SQL blocks.

Practice 1: Overview

This practice covers the following topics:

- Identifying the PL/SQL blocks that execute successfully
- Creating and executing a simple PL/SQL block

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practice 1: Overview

This practice reinforces the basics of PL/SQL covered in this lesson.

- Exercise 1 is a paper-based exercise in which you identify PL/SQL blocks that execute successfully.
- Exercise 2 involves creating and executing a simple PL/SQL block.

2

Declaring PL/SQL Variables

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

After completing this lesson, you should be able to do the following:

- Recognize valid and invalid identifiers
- List the uses of variables
- Declare and initialize variables
- List and describe various data types
- Identify the benefits of using the `%TYPE` attribute
- Declare, use, and print bind variables

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

You have already learned about basic PL/SQL blocks and their sections. In this lesson, you learn about valid and invalid identifiers. You learn how to declare and initialize variables in the declarative section of a PL/SQL block. The lesson describes the various data types. You also learn about the `%TYPE` attribute and its benefits.

Agenda

- Introducing variables
- Examining variable data types and the `%TYPE` attribute
- Examining bind variables

ORACLE

Copyright © 2009, Oracle. All rights reserved.

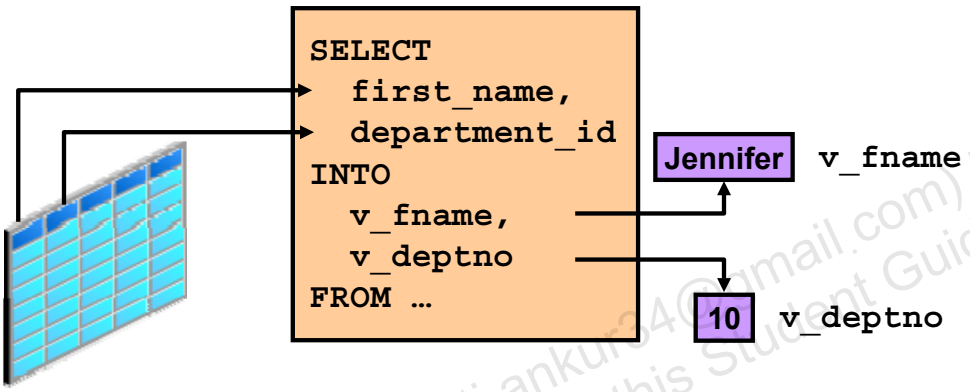
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Use of Variables

Variables can be used for:

- Temporary storage of data
- Manipulation of stored values
- Reusability



Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Use of Variables

With PL/SQL, you can declare variables, and then use them in SQL and procedural statements.

Variables are mainly used for storage of data and manipulation of stored values. Consider the PL/SQL statement in the slide. The statement retrieves `first_name` and `department_id` from the table. If you have to manipulate `first_name` or `department_id`, you have to store the retrieved value. Variables are used to temporarily store the value. You can use the value stored in these variables for processing and manipulating data. Variables can store any PL/SQL object such as variables, types, cursors, and subprograms.

Reusability is another advantage of declaring variables. After the variables are declared, you can use them repeatedly in an application by referring to them multiple times in various statements.

Requirements for Variable Names

A variable name:

- Must start with a letter
- Can include letters or numbers
- Can include special characters (such as $, _, and #)
- Must contain no more than 30 characters
- Must not include reserved words

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Requirements for Variable Names

The rules for naming a variable are listed in the slide.

Handling Variables in PL/SQL

Variables are:

- Declared and (optionally) initialized in the declarative section
- Used and assigned new values in the executable section
- Passed as parameters to PL/SQL subprograms
- Used to hold the output of a PL/SQL subprogram

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Handling Variables in PL/SQL

You can use variables in the following ways:

- **Declare and initialize them in the declaration section:** You can declare variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its data type, and name the storage location so that you can reference it. Declarations can also assign an initial value and impose the `NOT NULL` constraint on the variable. Forward references are not allowed. You must declare a variable before referencing it in other statements, including other declarative statements.
- **Use them and assign new values to them in the executable section:** In the executable section, the existing value of the variable can be replaced with a new value.
- **Pass them as parameters to PL/SQL subprograms:** Subprograms can take parameters. You can pass variables as parameters to subprograms.
- **Use them to hold the output of a PL/SQL subprogram:** Variables can be used to hold the value that is returned by a function.

Declaring and Initializing PL/SQL Variables

Syntax:

```
identifier [CONSTANT] datatype [NOT NULL]
     [:= | DEFAULT expr];
```

Examples:

```
DECLARE
  v_hiredate      DATE;
  v_deptno        NUMBER(2) NOT NULL := 10;
  v_location      VARCHAR2(13) := 'Atlanta';
  c_comm          CONSTANT NUMBER := 1400;
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Declaring and Initializing PL/SQL Variables

You must declare all PL/SQL identifiers in the declaration section before referencing them in the PL/SQL block. You have the option of assigning an initial value to a variable (as shown in the slide). You do not need to assign a value to a variable in order to declare it. If you refer to other variables in a declaration, be sure that they are already declared separately in a previous statement.

In the syntax:

identifier	Is the name of the variable
CONSTANT	Constrains the variable so that its value cannot change (Constants must be initialized.)
data type	Is a scalar, composite, reference, or LOB data type (This course covers only scalar, composite, and LOB data types.)
NOT NULL	Constrains the variable so that it contains a value (NOT NULL variables must be initialized.)
expr	Is any PL/SQL expression that can be a literal expression, another variable, or an expression involving operators and functions

Note: In addition to variables, you can also declare cursors and exceptions in the declarative section. You learn about declaring cursors in the lesson titled “Using Explicit Cursors” and about exceptions in the lesson titled “Handling Exceptions.”

Declaring and Initializing PL/SQL Variables

1

```
DECLARE
  v_myName VARCHAR2(20);
BEGIN
 DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
 v_myName := 'John';
 DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
END;
/
```

2

```
DECLARE
 v_myName VARCHAR2(20):= 'John';
BEGIN
 v_myName := 'Steven';
 DBMS_OUTPUT.PUT_LINE('My name is: '|| v_myName);
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

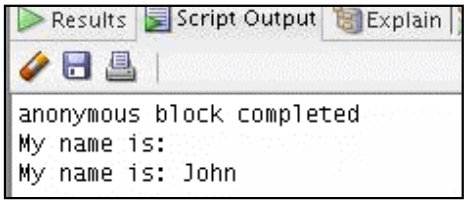
Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

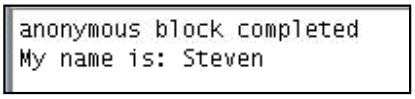
Declaring and Initializing PL/SQL Variables (continued)

Examine the two code blocks in the slide.

1. In the first block, the `v_myName` variable is declared but not initialized. A value `John` is assigned to the variable in the executable section.
   - String literals must be enclosed in single quotation marks. If your string has a quotation mark as in "Today's Date," the string would be `'Today''s Date'`.
   - The assignment operator is: "`:=`".
   - The `PUT_LINE` procedure is invoked by passing the `v_myName` variable. The value of the variable is concatenated with the string `'My name is:'`.
   - Output of this anonymous block is:



2. In the second block, the `v_myName` variable is declared and initialized in the declarative section. `v_myName` holds the value `John` after initialization. This value is manipulated in the executable section of the block. The output of this anonymous block is:



Delimiters in String Literals

```
DECLARE
    v_event VARCHAR2(15);
BEGIN
   v_event  := q'!Father's day!';
  DBMS_OUTPUT.PUT_LINE('3rd Sunday in June is :
   '|| v_event );
   v_event  := q'[Mother's day]';
  DBMS_OUTPUT.PUT_LINE('2nd Sunday in May is :
   '|| v_event );
END;
/
```

Resulting output

```
anonymous block completed
3rd Sunday in June is : Father's day
2nd Sunday in May is : Mother's day
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur.34@gmail.com) has a non-transferable license to use this Student Guide.

Delimiters in String Literals

If your string contains an apostrophe (identical to a single quotation mark), you must double the quotation mark, as in the following example:

```
v_event  VARCHAR2(15):='Father''s day';
```

The first quotation mark acts as the escape character. This makes your string complicated, especially if you have SQL statements as strings. You can specify any character that is not present in the string as a delimiter. The slide shows how to use the `q'` notation to specify the delimiter. The example uses `!` and `[` as delimiters. Consider the following example:

```
v_event  := q'!Father's day!';
```

You can compare this with the first example on this page. You start the string with `q'` if you want to use a delimiter. The character following the notation is the delimiter used. Enter your string after specifying the delimiter, close the delimiter, and close the notation with a single quotation mark. The following example shows how to use `[` as a delimiter:

```
v_event  := q'[Mother's day]';
```

Agenda

- Introducing variables
- Examining variable data types and the `%TYPE` attribute
- Examining bind variables

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Types of Variables

- PL/SQL variables:
  - Scalar
  - Reference
  - Large object (LOB)
  - Composite
- Non-PL/SQL variables: Bind variables

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Types of Variables

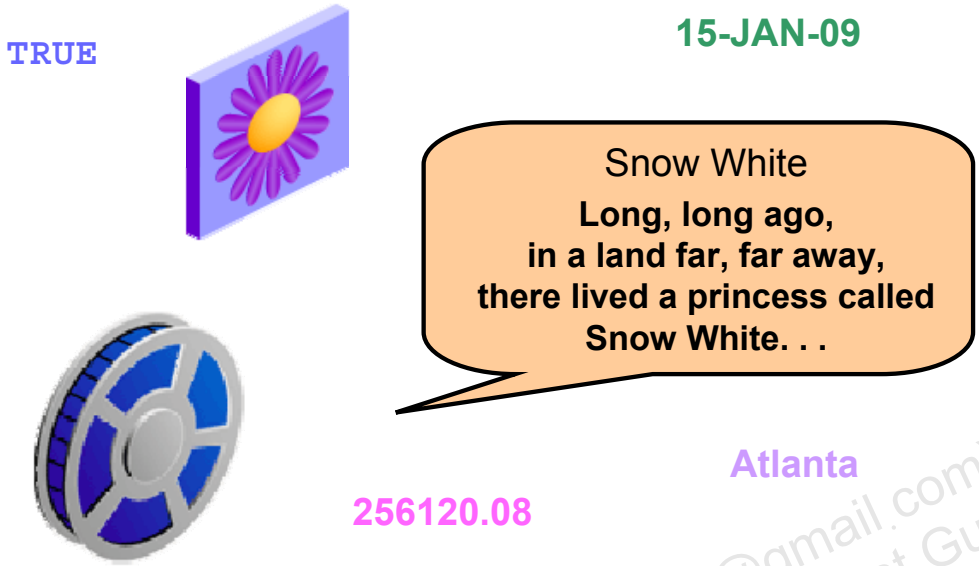
Every PL/SQL variable has a data type, which specifies a storage format, constraints, and a valid range of values. PL/SQL supports several data type categories, including scalar, reference, large object (LOB), and composite.

- **Scalar data types:** Scalar data types hold a single value. The value depends on the data type of the variable. For example, the `v_myName` variable in the example in the section "Declaring and Initializing PL/SQL Variables" (in this lesson) is of type `VARCHAR2`. Therefore, `v_myName` can hold a string value. PL/SQL also supports Boolean variables.
- **Reference data types:** Reference data types hold values, called *pointers*, which point to a storage location.
- `LOB` **data types:** `LOB` data types hold values, called *locators*, which specify the location of large objects (such as graphic images) that are stored outside the table.
- **Composite data types:** Composite data types are available by using PL/SQL *collection* and *record* variables. PL/SQL collections and records contain internal elements that you can treat as individual variables.

Non-PL/SQL variables include host language variables declared in precompiler programs, screen fields in Forms applications, and host variables. You learn about host variables later in this lesson.

For more information about `LOB`s, see the *PL/SQL User's Guide and Reference*.

Types of Variables



Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Arvind M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Types of Variables (continued)

The slide illustrates the following data types:

- `TRUE` represents a Boolean value.
- 15-JAN-09 represents a `DATE`.
- The image represents a `BLOB`.
- The text in the callout can represent a `VARCHAR2` data type or a `CLOB`.
- 256120.08 represents a `NUMBER` data type with precision and scale.
- The film reel represents a `BFILE`.
- The city name *Atlanta* represents a `VARCHAR2` data type.

Guidelines for Declaring and Initializing PL/SQL Variables

- Follow consistent naming conventions.
- Use meaningful identifiers for variables.
- Initialize variables that are designated as `NOT NULL` and `CONSTANT`.
- Initialize variables with the assignment operator (`:=`) or the `DEFAULT` keyword:

```
v_myName VARCHAR2(20):='John';
```

```
v_myName VARCHAR2(20) DEFAULT 'John';
```

- Declare one identifier per line for better readability and code maintenance.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Guidelines for Declaring and Initializing PL/SQL Variables

Here are some guidelines to follow when you declare PL/SQL variables.

- Follow consistent naming conventions—for example, you might use `name` to represent a variable and `c_name` to represent a constant. Similarly, to name a variable, you can use `v_fname`. The key is to apply your naming convention consistently for easier identification.
- Use meaningful and appropriate identifiers for variables. For example, consider using `salary` and `sal_with_commission` instead of `salary1` and `salary2`.
- If you use the `NOT NULL` constraint, you must assign a value when you declare the variable.
- In constant declarations, the `CONSTANT` keyword must precede the type specifier. The following declaration names a constant of `NUMBER` type and assigns the value of 50,000 to the constant. A constant must be initialized in its declaration; otherwise, you get a compilation error. After initializing a constant, you cannot change its value.

```
sal CONSTANT NUMBER := 50000.00;
```

Guidelines for Declaring PL/SQL Variables

- Avoid using column names as identifiers.

```
DECLARE
  employee_id  NUMBER(6);
BEGIN
  SELECT     employee_id
  INTO       employee_id
  FROM       employees
  WHERE      last_name = 'Kochhar';
END;
/
```

- Use the NOT NULL constraint when the variable must hold a value.

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Guidelines for Declaring PL/SQL Variables

- Initialize the variable to an expression with the assignment operator (:=) or with the DEFAULT reserved word. If you do not assign an initial value, the new variable contains NULL by default until you assign a value. To assign or reassign a value to a variable, you write a PL/SQL assignment statement. However, it is good programming practice to initialize all variables.
- Two objects can have the same name only if they are defined in different blocks. Where they coexist, you can qualify them with labels and use them.
- Avoid using column names as identifiers. If PL/SQL variables occur in SQL statements and have the same name as a column, the Oracle Server assumes that it is the column that is being referenced. Although the code example in the slide works, code that is written using the same name for a database table and a variable is not easy to read or maintain.
- Impose the NOT NULL constraint when the variable must contain a value. You cannot assign nulls to a variable that is defined as NOT NULL. The NOT NULL constraint must be followed by an initialization clause.

```
pincode VARCHAR2(15) NOT NULL := 'Oxford';
```

Naming Conventions of PL/SQL Structures Used in This Course

PL/SQL Structure	Convention	Example
Variable	`v_variable_name`	`v_rate`
Constant	`c_constant_name`	`c_rate`
Subprogram parameter	`p_parameter_name`	`p_id`
Bind (host) variable	`b_bind_name`	`b_salary`
Cursor	`cur_cursor_name`	`cur_emp`
Record	`rec_record_name`	`rec_emp`
Type	`type_name_type`	`ename_table_type`
Exception	`e_exception_name`	`e_products_invalid`
File handle	`f_file_handle_name`	`f_file`

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Naming Conventions of PL/SQL Structures Used in This Course

The table in the slide displays some examples of the naming conventions for PL/SQL structures that are used in this course.

Scalar Data Types

- Hold a single value
- Have no internal components

TRUE

15-JAN-09

The soul of the lazy man desires, and he has nothing; but the soul of the diligent shall be made rich.

256120.08

Atlanta

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Scalar Data Types

PL/SQL provides a variety of predefined data types. For instance, you can choose from integer, floating point, character, Boolean, date, collection, and LOB types. This lesson covers the basic types that are used frequently in PL/SQL programs.

A scalar data type holds a single value and has no internal components. Scalar data types can be classified into four categories: number, character, date, and Boolean. Character and number data types have subtypes that associate a base type to a constraint. For example, `INTEGER` and `POSITIVE` are subtypes of the `NUMBER` base type.

For more information about scalar data types (as well as a complete list), see the *PL/SQL User's Guide and Reference*.

Base Scalar Data Types

- CHAR [(maximum_length)]
- VARCHAR2 (maximum_length)
- NUMBER [(precision, scale)]
- BINARY_INTEGER
- PLS_INTEGER
- BOOLEAN
- BINARY_FLOAT
- BINARY_DOUBLE

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Base Scalar Data Types

Data Type	Description
CHAR [(*maximum_length*)]	Base type for fixed-length character data up to 32,767 bytes. If you do not specify a maximum length, the default length is set to 1.
VARCHAR2 (*maximum_length*)	Base type for variable-length character data up to 32,767 bytes. There is no default size for VARCHAR2 variables and constants.
NUMBER [(*precision*, *scale*)]	Number having precision *p* and scale *s*. The precision *p* can range from 1 through 38. The scale *s* can range from –84 through 127.
BINARY_INTEGER	Base type for integers between –2,147,483,647 and 2,147,483,647

Base Scalar Data Types (continued)

Data Type	Description
PLS_INTEGER	Base type for signed integers between –2,147,483,647 and 2,147,483,647. PLS_INTEGER values require less storage and are faster than NUMBER values. In Oracle Database 11*g*, the PLS_INTEGER and BINARY_INTEGER data types are identical. The arithmetic operations on PLS_INTEGER and BINARY_INTEGER values are faster than on NUMBER values.
BOOLEAN	Base type that stores one of the three possible values used for logical calculations: TRUE, FALSE, and NULL
BINARY_FLOAT	Represents floating-point number in IEEE 754 format. It requires 5 bytes to store the value.
BINARY_DOUBLE	Represents floating-point number in IEEE 754 format. It requires 9 bytes to store the value.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Base Scalar Data Types

- DATE
- TIMESTAMP
- TIMESTAMP WITH TIME ZONE
- TIMESTAMP WITH LOCAL TIME ZONE
- INTERVAL YEAR TO MONTH
- INTERVAL DAY TO SECOND

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Base Scalar Data Types (continued)

Data Type	Description
DATE	Base type for dates and times. DATE values include the time of day in seconds since midnight. The range for dates is between 4712 B.C. and A.D. 9999.
TIMESTAMP	The TIMESTAMP data type, which extends the DATE data type, stores the year, month, day, hour, minute, second, and fraction of second. The syntax is TIMESTAMP[(precision)], where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. To specify the precision, you must use an integer in the range 0–9. The default is 6.
TIMESTAMP WITH TIME ZONE	The TIMESTAMP WITH TIME ZONE data type, which extends the TIMESTAMP data type, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. The syntax is TIMESTAMP[(precision)] WITH TIME ZONE, where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. To specify the precision, you must use an integer in the range 0–9. The default is 6.

Base Scalar Data Types (continued)

Data Type	Description
TIMESTAMP WITH LOCAL TIME ZONE	The TIMESTAMP WITH LOCAL TIME ZONE data type, which extends the TIMESTAMP data type, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time. The syntax is TIMESTAMP[(precision)] WITH LOCAL TIME ZONE, where the optional parameter precision specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0–9. The default is 6. This data type differs from TIMESTAMP WITH TIME ZONE in that when you insert a value into a database column, the value is normalized to the database time zone, and the time-zone displacement is not stored in the column. When you retrieve the value, the Oracle server returns the value in your local session time zone.
INTERVAL YEAR TO MONTH	You use the INTERVAL YEAR TO MONTH data type to store and manipulate intervals of years and months. The syntax is INTERVAL YEAR[(precision)] TO MONTH, where precision specifies the number of digits in the years field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0–4. The default is 2.
INTERVAL DAY TO SECOND	You use the INTERVAL DAY TO SECOND data type to store and manipulate intervals of days, hours, minutes, and seconds. The syntax is INTERVAL DAY[(precision1)] TO SECOND[(precision2)], where precision1 and precision2 specify the number of digits in the days field and seconds field, respectively. In both cases, you cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0–9. The defaults are 2 and 6, respectively.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Declaring Scalar Variables

Examples:

```
DECLARE
  v_emp_job          VARCHAR2(9);
  v_count_loop       BINARY_INTEGER := 0;
  v_dept_total_sal   NUMBER(9,2) := 0;
  v_orderdate        DATE := SYSDATE + 7;
  c_tax_rate         CONSTANT NUMBER(3,2) := 8.25;
  v_valid            BOOLEAN NOT NULL := TRUE;
  ...
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Declaring Scalar Variables

The examples of variable declaration shown in the slide are defined as follows:

- **`v_emp_job`:** Variable to store an employee job title
- **`v_count_loop`:** Variable to count the iterations of a loop; initialized to `0`
- **`v_dept_total_sal`:** Variable to accumulate the total salary for a department; initialized to `0`
- **`v_orderdate`:** Variable to store the ship date of an order; initialized to one week from today
- **`c_tax_rate`:** Constant variable for the tax rate (which never changes throughout the PL/SQL block); set to `8.25`
- **`v_valid`:** Flag to indicate whether a piece of data is valid or invalid; initialized to `TRUE`

%TYPE Attribute

- Is used to declare a variable according to:
  - A database column definition
  - Another declared variable
- Is prefixed with:
  - The database table and column name
  - The name of the declared variable

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

%TYPE Attribute

PL/SQL variables are usually declared to hold and manipulate data stored in a database. When you declare PL/SQL variables to hold column values, you must ensure that the variable is of the correct data type and precision. If it is not, a PL/SQL error occurs during execution. If you have to design large subprograms, this can be time consuming and error prone.

Rather than hard-coding the data type and precision of a variable, you can use the `%TYPE` attribute to declare a variable according to another previously declared variable or database column. The `%TYPE` attribute is most often used when the value stored in the variable is derived from a table in the database. When you use the `%TYPE` attribute to declare a variable, you should prefix it with the database table and column name. If you refer to a previously declared variable, prefix the variable name of the previously declared variable to the variable being declared.

`%TYPE` Attribute (continued)

Advantages of the `%TYPE` Attribute

- You can avoid errors caused by data type mismatch or wrong precision.
- You can avoid hard coding the data type of a variable.
- You need not change the variable declaration if the column definition changes. If you have already declared some variables for a particular table without using the `%TYPE` attribute, the PL/SQL block may throw errors if the column for which the variable is declared is altered. When you use the `%TYPE` attribute, PL/SQL determines the data type and size of the variable when the block is compiled. This ensures that such a variable is always compatible with the column that is used to populate it.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Declaring Variables with the %TYPE Attribute

Syntax

```
identifier     table.column_name%TYPE;
```

Examples

```
...
  v_emp_lname       employees.last_name%TYPE;
...
```

```
...
  v_balance         NUMBER(7,2);
  v_min_balance     v_balance%TYPE := 1000;
...
```

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Declaring Variables with the %TYPE Attribute

Declare variables to store the last name of an employee. The `v_emp_lname` variable is defined to be of the same data type as the `v_last_name` column in the `employees` table. The `%TYPE` attribute provides the data type of a database column.

Declare variables to store the balance of a bank account, as well as the minimum balance, which is 1,000. The `v_min_balance` variable is defined to be of the same data type as the `v_balance` variable. The `%TYPE` attribute provides the data type of a variable.

A `NOT NULL` database column constraint does not apply to variables that are declared using `%TYPE`. Therefore, if you declare a variable using the `%TYPE` attribute that uses a database column defined as `NOT NULL`, you can assign the `NULL` value to the variable.

Declaring Boolean Variables

- Only the TRUE, FALSE, and NULL values can be assigned to a Boolean variable.
- Conditional expressions use the logical operators AND and OR, and the unary operator NOT to check the variable values.
- The variables always yield TRUE, FALSE, or NULL.
- Arithmetic, character, and date expressions can be used to return a Boolean value.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a nontransferable license to use this Student Guide.

Declaring Boolean Variables

With PL/SQL, you can compare variables in both SQL and procedural statements. These comparisons, called Boolean expressions, consist of simple or complex expressions separated by relational operators. In a SQL statement, you can use Boolean expressions to specify the rows in a table that are affected by the statement. In a procedural statement, Boolean expressions are the basis for conditional control. NULL stands for a missing, inapplicable, or unknown value.

Examples

```
emp_sal1 := 50000;
emp_sal2 := 60000;
```

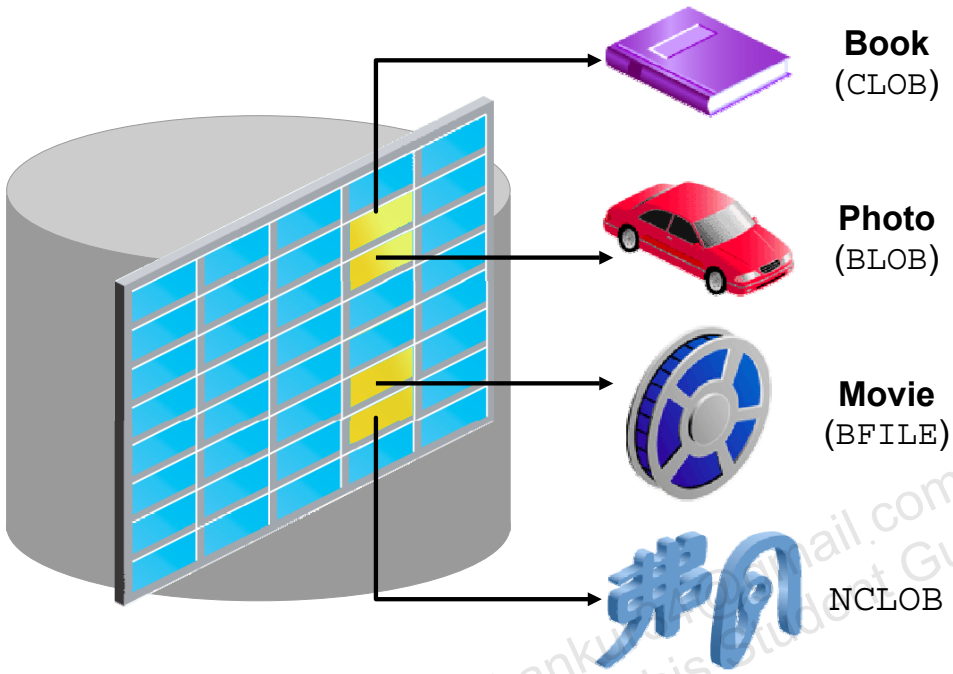
The following expression yields TRUE:

```
emp_sal1 < emp_sal2
```

Declare and initialize a Boolean variable:

```
DECLARE
 flag BOOLEAN := FALSE;
BEGIN
 flag := TRUE;
END;
/
```

LOB Data Type Variables



Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur@gmail.com) has a non-transferable license to use this Student Guide.

LOB Data Type Variables

Large objects (LOBs) are meant to store a large amount of data. A database column can be of the LOB category. With the LOB category of data types (BLOB, CLOB, and so on), you can store blocks of unstructured data (such as text, graphic images, video clips, and sound wave forms) of up to 128 terabytes depending on the database block size. LOB data types allow efficient, random, piecewise access to data and can be attributes of an object type.

- The character large object (CLOB) data type is used to store large blocks of character data in the database.
- The binary large object (BLOB) data type is used to store large unstructured or structured binary objects in the database. When you insert or retrieve such data into or from the database, the database does not interpret the data. External applications that use this data must interpret the data.
- The binary file (BFILE) data type is used to store large binary files. Unlike other LOBs, BFILES are stored outside the database and not in the database. They could be operating system files. Only a pointer to the BFILE is stored in the database.
- The national language character large object (NCLOB) data type is used to store large blocks of single-byte or fixed-width multibyte NCHAR unicode data in the database.

Composite Data Types: Records and Collections

PL/SQL Record:

TRUE	23-DEC-98	ATLANTA	

PL/SQL Collections:

1	SMITH
2	JONES
3	NANCY
4	TIM

VARCHAR2

PLS_INTEGER

1	5000
2	2345
3	12
4	3456

NUMBER

PLS_INTEGER

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Composite Data Types: Records and Collections

As mentioned previously, a scalar data type holds a single value and has no internal components. Composite data types—called PL/SQL Records and PL/SQL Collections—have internal components that that you can treat as individual variables.

- In a PL/SQL record, the internal components can be of different data types, and are called fields. You access each field with this syntax: `record_name.field_name`. A record variable can hold a table row, or some columns from a table row. Each record field corresponds to a table column.
- In a PL/SQL collection, the internal components are always of the same data type, and are called elements. You access each element by its unique subscript. Lists and arrays are classic examples of collections. There are three types of PL/SQL collections: Associative Arrays, Nested Tables, and `VARRAY` types.

Note

- PL/SQL Records and Associative Arrays are covered in the lesson titled: "Working with Composite Data Types."
- `NESTED TABLE` and `VARRAY` data types are covered in the course titled *Oracle Database 11g: Advanced PL/SQL*.

Agenda

- Introducing variables
- Examining variable data types and the %TYPE attribute
- Examining bind variables

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Bind Variables

Bind variables are:

- Created in the environment
- Also called *host* variables
- Created with the `VARIABLE` keyword*
- Used in SQL statements and PL/SQL blocks
- Accessed even after the PL/SQL block is executed
- Referenced with a preceding colon

Values can be output using the `PRINT` command.

* Required when using SQL*Plus and SQL Developer

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a nontransferable license to use this Student Guide.

Bind Variables

Bind variables are variables that you create in a host environment. For this reason, they are sometimes called *host* variables.

Uses of Bind Variables

Bind variables are created in the environment and not in the declarative section of a PL/SQL block. Therefore, bind variables are accessible even after the block is executed. When created, bind variables can be used and manipulated by multiple subprograms. They can be used in SQL statements and PL/SQL blocks just like any other variable. These variables can be passed as run-time values into or out of PL/SQL subprograms.

Note: A bind variable is an environment variable, but is not a global variable.

Creating Bind Variables

To create a bind variable in SQL Developer, use the `VARIABLE` command. For example, you declare a variable of type `NUMBER` and `VARCHAR2` as follows:

```
VARIABLE return_code NUMBER
VARIABLE return_msg  VARCHAR2(30)
```

Viewing Values in Bind Variables

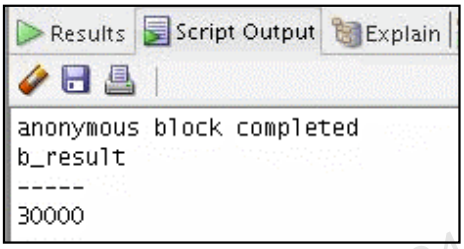
You can reference the bind variable using SQL Developer and view its value using the `PRINT` command.

Bind Variables (continued)

Example

You can reference a bind variable in a PL/SQL program by preceding the variable with a colon. For example, the following PL/SQL block creates and uses the bind variable `b_result`. The output resulting from the `PRINT` command is shown below the code.

```
VARIABLE b_result NUMBER
BEGIN
  SELECT (SALARY*12) + NVL(COMMISSION_PCT,0) INTO :b_result
  FROM employees WHERE employee_id = 144;
END;
/
PRINT b_result
```



Note: If you are creating a bind variable of the `NUMBER` type, you cannot specify the precision and scale. However, you can specify the size for character strings. An Oracle `NUMBER` is stored in the same way regardless of the dimension. The Oracle Server uses the same number of bytes to store 7, 70, and .0734. It is not practical to calculate the size of the Oracle number representation from the number format, so the code always allocates the bytes needed. With character strings, the user has to specify the size so that the required number of bytes can be allocated.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Trajapati (trajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Referencing Bind Variables

Example:

```
VARIABLE b_emp_salary NUMBER
BEGIN
   SELECT salary  INTO :b_emp_salary
   FROM  employees WHERE employee_id = 178;
END;
/
PRINT b_emp_salary
SELECT first_name, last_name
FROM employees
WHERE salary=:b_emp_salary;
```

Output →

Results | Script Output | Explain | Autotrace

```
anonymous block completed
b_emp_salary
----
7000

FIRST_NAME           LAST_NAME
-------------------- ----------------------
Oliver               Tuvault
Sarath               Sewall
Kimberely            Grant

3 rows selected
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide

Referencing Bind Variables

As stated previously, after you create a bind variable, you can reference that variable in any other SQL statement or PL/SQL program.

In the example, `b_emp_salary` is created as a bind variable in the PL/SQL block. Then, it is used in the `SELECT` statement that follows.

When you execute the PL/SQL block shown in the slide, you see the following output:

- The `PRINT` command executes:

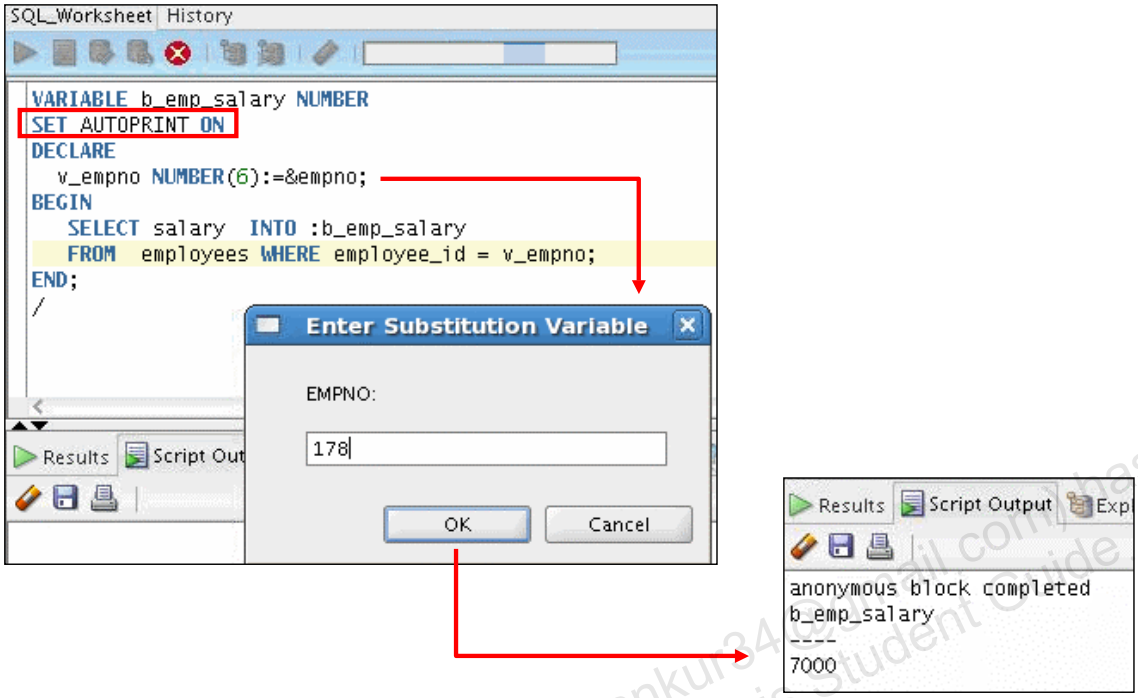
```
b_emp_salary
------------------
7000
```

- Then, the output of the SQL statement follows:

```
FIRST_NAME           LAST_NAME
------------------   ----------------------
Oliver               Tuvault
Sarath               Sewall
Kimberely            Grant
```

Note: To display all bind variables, use the `PRINT` command without a variable.

Using AUTOPRINT with Bind Variables



ORACLE
Copyright © 2009, Oracle. All rights reserved.

Using AUTOPRINT with Bind Variables

Use the SET AUTOPRINT ON command to automatically display the bind variables used in a successful PL/SQL block.

Example

In the code example:

- A bind variable named b_emp_salary is created and AUTOPRINT is turned on.
- A variable named v_empno is declared, and a substitution variable is used to receive user input.
- Finally, the bind variable and temporary variables are used in the executable section of the PL/SQL block.

When a valid employee number is entered—in this case 178—the output of the bind variable is automatically printed. The bind variable contains the salary for the employee number that is provided by the user.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Quiz

The %TYPE attribute:

1. Is used to declare a variable according to a database column definition
2. Is used to declare a variable according to a collection of columns in a database table or view
3. Is used to declare a variable according to the definition of another declared variable
4. Is prefixed with the database table and column name or the name of the declared variable

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Answer: 1, 3, 4

The %TYPE Attribute

PL/SQL variables are usually declared to hold and manipulate data stored in a database. When you declare PL/SQL variables to hold column values, you must ensure that the variable is of the correct data type and precision. If it is not, a PL/SQL error occurs during execution. If you have to design large subprograms, this can be time consuming and error prone.

Rather than hard-coding the data type and precision of a variable, you can use the %TYPE attribute to declare a variable according to another previously declared variable or database column. The %TYPE attribute is most often used when the value stored in the variable is derived from a table in the database. When you use the %TYPE attribute to declare a variable, you should prefix it with the database table and column name. If you refer to a previously declared variable, prefix the variable name of the previously declared variable to the variable being declared. The benefit of %TYPE is that you do not have to change the variable if the column is altered. Also, if the variable is used in any calculations, you need not worry about its precision.

The %ROWTYPE Attribute

The %ROWTYPE attribute is used to declare a record that can hold an entire row of a table or view. You learn about this attribute in the lesson titled "Working with Composite Data Types."

Summary

In this lesson, you should have learned how to:

- Recognize valid and invalid identifiers
- Declare variables in the declarative section of a PL/SQL block
- Initialize variables and use them in the executable section
- Differentiate between scalar and composite data types
- Use the `%TYPE` attribute
- Use bind variables

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Summary

An anonymous PL/SQL block is a basic, unnamed unit of a PL/SQL program. It consists of a set of SQL or PL/SQL statements to perform a logical function. The declarative part is the first part of a PL/SQL block and is used for declaring objects such as variables, constants, cursors, and definitions of error situations called *exceptions*.

In this lesson, you learned how to declare variables in the declarative section. You saw some of the guidelines for declaring variables. You learned how to initialize variables when you declare them.

The executable part of a PL/SQL block is the mandatory part and contains SQL and PL/SQL statements for querying and manipulating data. You learned how to initialize variables in the executable section and also how to use them and manipulate the values of variables.

Practice 2: Overview

This practice covers the following topics:

- Determining valid identifiers
- Determining valid variable declarations
- Declaring variables within an anonymous block
- Using the `%TYPE` attribute to declare variables
- Declaring and printing a bind variable
- Executing a PL/SQL block

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practice 2: Overview

Exercises 1, 2, and 3 are paper based.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

3

Writing Executable Statements

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

After completing this lesson, you should be able to do the following:

- Identify lexical units in a PL/SQL block
- Use built-in SQL functions in PL/SQL
- Describe when implicit conversions take place and when explicit conversions have to be dealt with
- Write nested blocks and qualify variables with labels
- Write readable code with appropriate indentation
- Use sequences in PL/SQL expressions

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

You learned how to declare variables and write executable statements in a PL/SQL block. In this lesson, you learn how lexical units make up a PL/SQL block. You learn to write nested blocks. You also learn about the scope and visibility of variables in nested blocks and about qualifying variables with labels.

Agenda

- Writing executable statements in a PL/SQL block
- Writing nested blocks
- Using operators and developing readable code

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Lexical Units in a PL/SQL Block

Lexical units:

- Are building blocks of any PL/SQL block
- Are sequences of characters including letters, numerals, tabs, spaces, returns, and symbols
- Can be classified as:
  - Identifiers: `v_fname, c_percent`
  - Delimiters: `; , +, -`
  - Literals: `John, 428, True`
  - Comments: `--, /* */`

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Lexical Units in a PL/SQL Block

Lexical units include letters, numerals, special characters, tabs, spaces, returns, and symbols.

- **Identifiers:** Identifiers are the names given to PL/SQL objects. You learned to identify valid and invalid identifiers. Recall that keywords cannot be used as identifiers.
  Quoted identifiers:
  - Make identifiers case-sensitive.
  - Include characters such as spaces.
  - Use reserved words.

  Examples:

  ```
  "begin date" DATE;
  "end date"   DATE;
  "exception thrown" BOOLEAN DEFAULT TRUE;
  ```

  All subsequent usage of these variables should have double quotation marks. However, use of quoted identifiers is not recommended.
- **Delimiters:** Delimiters are symbols that have special meaning. You already learned that the semicolon (`;`) is used to terminate a SQL or PL/SQL statement. Therefore, `;` is an example of a delimiter.
  For more information, refer to the *PL/SQL User's Guide and Reference*.

Lexical Units in a PL/SQL Block (continued)

- **Delimiters (continued)**
  Delimiters are simple or compound symbols that have special meaning in PL/SQL.
  Simple symbols

Symbol	Meaning
+	Addition operator
-	Subtraction/negation operator
*	Multiplication operator
/	Division operator
=	Equality operator
@	Remote access indicator
;	Statement terminator

Compound symbols

Symbol	Meaning
<>	Inequality operator
!=	Inequality operator
||	Concatenation operator
--	Single-line comment indicator
/*	Beginning comment delimiter
*/	Ending comment delimiter
:=	Assignment operator

Note: This is only a subset and not a complete list of delimiters.

- **Literals:** Any value that is assigned to a variable is a literal. Any character, numeral, Boolean, or date value that is not an identifier is a literal. Literals are classified as:
  - **Character literals:** All string literals have the data type `CHAR` or `VARCHAR2` and are, therefore, called character literals (for example, `John`, and `12C`).
  - **Numeric literals:** A numeric literal represents an integer or real value (for example, `428` and `1.276`).
  - **Boolean literals:** Values that are assigned to Boolean variables are Boolean literals. `TRUE`, `FALSE`, and `NULL` are Boolean literals or keywords.
- **Comments:** It is good programming practice to explain what a piece of code is trying to achieve. However, when you include the explanation in a PL/SQL block, the compiler cannot interpret these instructions. Therefore, there should be a way in which you can indicate that these instructions need not be compiled. Comments are mainly used for this purpose. Any instruction that is commented is not interpreted by the compiler.
  - Two hyphens (`--`) are used to comment a single line.
  - The beginning and ending comment delimiters (`/*` and `*/`) are used to comment multiple lines.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

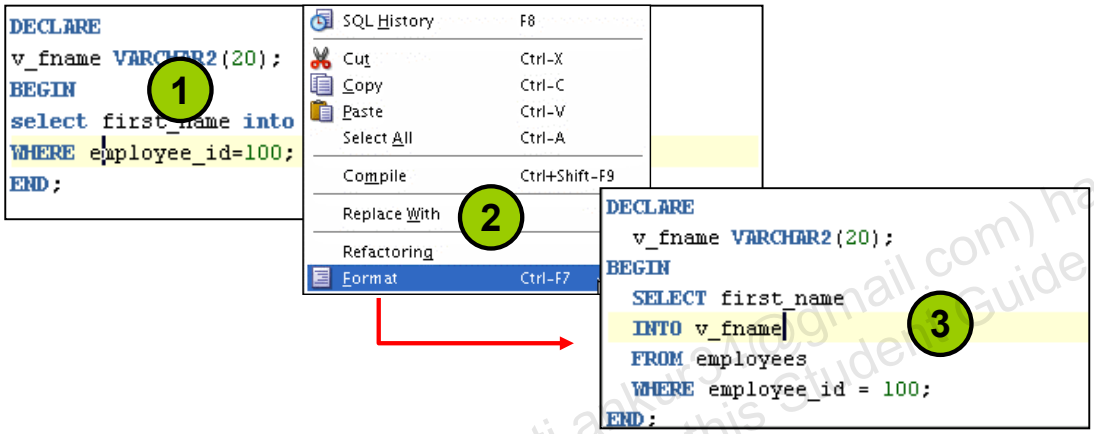
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

PL/SQL Block Syntax and Guidelines

- Using Literals
  - Character and date literals must be enclosed in single quotation marks.
  - Numbers can be simple values or in scientific notation.

```
v_name := 'Henderson';
```

- Formatting Code: Statements can span several lines.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur.m@gmail.com) has a non-transferable license to use this Student Guide.

PL/SQL Block Syntax and Guidelines

Using Literals

A literal is an explicit numeric, character string, date, or Boolean value that is not represented by an identifier.

- Character literals include all printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols.
- Numeric literals can be represented either by a simple value (for example, `-32.5`) or in scientific notation (for example, `2E5` means $2 * 10^5 = 200{,}000$).

Formatting Code

In a PL/SQL block, a SQL statement can span several lines (as shown in example 3 in the slide).

You can format an unformatted SQL statement (as shown in example 1 in the slide) by using the SQL Worksheet shortcut menu. Right-click the active SQL Worksheet and, in the shortcut menu that appears, select the Format option (as shown in example 2).

Note: You can also use the shortcut key combination of Ctrl + F7 to format your code.

Commenting Code

- Prefix single-line comments with two hyphens (--).
- Place a block comment between the symbols /* and */.

Example:

```
DECLARE
...
v_annual_sal NUMBER (9,2);
BEGIN
/* Compute the annual salary based on the
   monthly salary input from the user */
v_annual_sal := monthly_sal * 12;
--The following line displays the annual salary
DBMS_OUTPUT.PUT_LINE(v_annual_sal);
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Commenting Code

You should comment code to document each phase and to assist debugging. In PL/SQL code:

- A single-line comment is commonly prefixed with two hyphens (--)
- You can also enclose a comment between the symbols /* and */

Note: For multiline comments, you can either precede each comment line with two hyphens, or use the block comment format.

Comments are strictly informational and do not enforce any conditions or behavior on the logic or data. Well-placed comments are extremely valuable for code readability and future code maintenance.

SQL Functions in PL/SQL

- Available in procedural statements:
  - Single-row functions
- Not available in procedural statements:
  - `DECODE`
  - Group functions

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL Functions in PL/SQL

SQL provides several predefined functions that can be used in SQL statements. Most of these functions (such as single-row number and character functions, data type conversion functions, and date and time-stamp functions) are valid in PL/SQL expressions.

The following functions are not available in procedural statements:

- `DECODE`
- Group functions: `AVG`, `MIN`, `MAX`, `COUNT`, `SUM`, `STDDEV`, and `VARIANCE`
  Group functions apply to groups of rows in a table and are, therefore, available only in SQL statements in a PL/SQL block. The functions mentioned here are only a subset of the complete list.

SQL Functions in PL/SQL: Examples

- Get the length of a string:

```
v_desc_size INTEGER(5);
v_prod_description VARCHAR2(70):='You can use this
product with your radios for higher frequency';

-- get the length of the string in prod_description
v_desc_size:= LENGTH(v_prod_description);
```

- Get the number of months an employee has worked:

```
v_tenure:= MONTHS_BETWEEN (CURRENT_DATE, v_hiredate);
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL Functions in PL/SQL: Examples

You can use SQL functions to manipulate data. These functions are grouped into the following categories:

- Number
- Character
- Conversion
- Date
- Miscellaneous

Using Sequences in PL/SQL Expressions

Starting in 11*g*:

```
DECLARE
  v_new_id NUMBER;
BEGIN
  v_new_id := my_seq.NEXTVAL;
END;
/
```

Before 11*g*:

```
DECLARE
  v_new_id NUMBER;
BEGIN
  SELECT my_seq.NEXTVAL INTO v_new_id FROM Dual;
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Accessing Sequence Values

In Oracle Database 11*g*, you can use the `NEXTVAL` and `CURRVAL` pseudocolumns in any PL/SQL context, where an expression of the `NUMBER` data type may legally appear. Although the old style of using a `SELECT` statement to query a sequence is still valid, it is recommended that you do not use it.

Before Oracle Database 11*g*, you were forced to write a SQL statement in order to use a sequence object value in a PL/SQL subroutine. Typically, you would write a `SELECT` statement to reference the pseudocolumns of `NEXTVAL` and `CURRVAL` to obtain a sequence number. This method created a usability problem.

In Oracle Database 11*g*, the limitation of forcing you to write a SQL statement to retrieve a sequence value is eliminated. With the sequence enhancement feature:

- Sequence usability is improved
- The developer has to type less
- The resulting code is clearer

Data Type Conversion

- Converts data to comparable data types
- Is of two types:
  - Implicit conversion
  - Explicit conversion
- Functions:
  - TO_CHAR
  - TO_DATE
  - TO_NUMBER
  - TO_TIMESTAMP

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Data Type Conversion

In any programming language, converting one data type to another is a common requirement. PL/SQL can handle such conversions with scalar data types. Data type conversions can be of two types:

Implicit conversions: PL/SQL attempts to convert data types dynamically if they are mixed in a statement. Consider the following example:

```
DECLARE
 v_salary NUMBER(6):=6000;
 v_sal_hike VARCHAR2(5):='1000';
 v_total_salary v_salary%TYPE;
BEGIN
 v_total_salary:=v_salary + v_sal_hike;
END;
/
```

In this example, the `sal_hike` variable is of the `VARCHAR2` type. When calculating the total salary, PL/SQL first converts `sal_hike` to `NUMBER`, and then performs the operation. The result is of the `NUMBER` type.

Implicit conversions can be between:

- Characters and numbers
- Characters and dates

Data Type Conversion (continued)

Explicit conversions: To convert values from one data type to another, use built-in functions. For example, to convert a `CHAR` value to a `DATE` or `NUMBER` value, use `TO_DATE` or `TO_NUMBER`, respectively.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Data Type Conversion

1
```
-- implicit data type conversion
v_date_of_joining DATE:= '02-Feb-2000';
```

2
```
-- error in data type conversion
v_date_of_joining DATE:= 'February 02,2000';
```

3
```
-- explicit data type conversion
v_date_of_joining DATE:= TO_DATE('February
02,2000','Month DD, YYYY');
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Data Type Conversion (continued)

Note the three examples of implicit and explicit conversions of the `DATE` data type in the slide:

1. Because the string literal being assigned to `date_of_joining` is in the default format, this example performs implicit conversion and assigns the specified date to `date_of_joining`.
2. The PL/SQL returns an error because the date that is being assigned is not in the default format.
3. The `TO_DATE` function is used to explicitly convert the given date in a particular format and assign it to the `DATE` data type variable `date_of_joining`.

Agenda

- Writing executable statements in a PL/SQL block
- Writing nested blocks
- Using operators and developing readable code

ORACLE

Copyright © 2009, Oracle. All rights reserved.

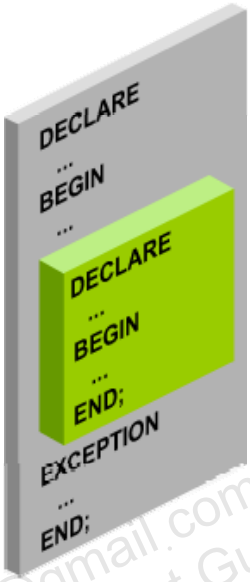
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Nested Blocks

PL/SQL blocks can be nested.

- An executable section (`BEGIN … END`) can contain nested blocks.
- An exception section can contain nested blocks.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (rajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Nested Blocks

Being procedural gives PL/SQL the ability to nest statements. You can nest blocks wherever an executable statement is allowed, thus making the nested block a statement. If your executable section has code for many logically related functionalities to support multiple business requirements, you can divide the executable section into smaller blocks. The exception section can also contain nested blocks.

Nested Blocks: Example

```
DECLARE
 v_outer_variable VARCHAR2(20):='GLOBAL VARIABLE';
BEGIN
  DECLARE
   v_inner_variable VARCHAR2(20):='LOCAL VARIABLE';
  BEGIN
   DBMS_OUTPUT.PUT_LINE(v_inner_variable);
   DBMS_OUTPUT.PUT_LINE(v_outer_variable);
  END;
 DBMS_OUTPUT.PUT_LINE(v_outer_variable);
END;
```

```
anonymous block completed
LOCAL VARIABLE
GLOBAL VARIABLE
GLOBAL VARIABLE
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Nested Blocks (continued)

The example shown in the slide has an outer (parent) block and a nested (child) block. The `v_outer_variable` variable is declared in the outer block and the `v_inner_variable` variable is declared in the inner block.

`v_outer_variable` is local to the outer block but global to the inner block. When you access this variable in the inner block, PL/SQL first looks for a local variable in the inner block with that name. There is no variable with the same name in the inner block, so PL/SQL looks for the variable in the outer block. Therefore, `v_outer_variable` is considered to be the global variable for all the enclosing blocks. You can access this variable in the inner block as shown in the slide. Variables declared in a PL/SQL block are considered local to that block and global to all its subblocks.

`v_inner_variable` is local to the inner block and is not global because the inner block does not have any nested blocks. This variable can be accessed only within the inner block. If PL/SQL does not find the variable declared locally, it looks upward in the declarative section of the parent blocks. PL/SQL does not look downward in the child blocks.

Variable Scope and Visibility

```
DECLARE
 v_father_name VARCHAR2(20):='Patrick';
 v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
   v_child_name VARCHAR2(20):='Mike';
   v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
   DBMS_OUTPUT.PUT_LINE('Father''s Name: '||v_father_name);
   DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
   DBMS_OUTPUT.PUT_LINE('Child''s Name: '||v_child_name);
  END;
 DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Variable Scope and Visibility

The output of the block shown in the slide is as follows:

```
anonymous block completed
Father's Name: Patrick
Date of Birth: 12-DEC-02
Child's Name: Mike
Date of Birth: 20-APR-72
```

Examine the date of birth that is printed for father and child. The output does not provide the correct information, because the scope and visibility of the variables are not applied correctly.

- The *scope* of a variable is the portion of the program in which the variable is declared and is accessible.
- The *visibility* of a variable is the portion of the program where the variable can be accessed without using a qualifier.

Scope

- The `v_father_name` variable and the first occurrence of the `v_date_of_birth` variable are declared in the outer block. These variables have the scope of the block in which they are declared. Therefore, the scope of these variables is limited to the outer block.

Variable Scope and Visibility (continued)

Scope (continued)

- The `v_child_name` and `v_date_of_birth` variables are declared in the inner block or the nested block. These variables are accessible only within the nested block and are not accessible in the outer block. When a variable is out of scope, PL/SQL frees the memory used to store the variable; therefore, these variables cannot be referenced.

Visibility

- The `v_date_of_birth` variable declared in the outer block has scope even in the inner block. However, this variable is not visible in the inner block because the inner block has a local variable with the same name.
  1. Examine the code in the executable section of the PL/SQL block. You can print the father's name, the child's name, and the date of birth. Only the child's date of birth can be printed here because the father's date of birth is not visible.
  2. The father's date of birth is visible in the outer block and, therefore, can be printed.

Note: You cannot have variables with the same name in a block. However, as shown in this example, you can declare variables with the same name in two different blocks (nested blocks). The two items represented by identifiers are distinct; changes in one do not affect the other.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Using a Qualifier with Nested Blocks

```
BEGIN <<outer>>
DECLARE
 v_father_name VARCHAR2(20):='Patrick';
 v_date_of_birth DATE:='20-Apr-1972';
BEGIN
  DECLARE
   v_child_name VARCHAR2(20):='Mike';
   v_date_of_birth DATE:='12-Dec-2002';
  BEGIN
   DBMS_OUTPUT.PUT_LINE('Father''s Name: '||v_father_name);
   DBMS_OUTPUT.PUT_LINE('Date of Birth: '
                        ||outer.v_date_of_birth);
   DBMS_OUTPUT.PUT_LINE('Child''s Name: '||v_child_name);
   DBMS_OUTPUT.PUT_LINE('Date of Birth: '||v_date_of_birth);
  END;
END;
END outer;
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Using a Qualifier with Nested Blocks

A *qualifier* is a label given to a block. You can use a qualifier to access the variables that have scope but are not visible.

Example

In the code example:

- The outer block is labeled `outer`
- Within the inner block, the `outer` qualifier is used to access the `v_date_of_birth` variable that is declared in the outer block. Therefore, the father's date of birth and the child's date of birth can both be printed from within the inner block.
- The output of the code in the slide shows the correct information:

```
anonymous block completed
Father's Name: Patrick
Date of Birth: 20-APR-72
Child's Name: Mike
Date of Birth: 12-DEC-02
```

Note: Labeling is not limited to the outer block. You can label any block.

Challenge: Determining Variable Scope

```
BEGIN <<outer>>
DECLARE
  v_sal       NUMBER(7,2) := 60000;
  v_comm      NUMBER(7,2) := v_sal * 0.20;
  v_message   VARCHAR2(255) := ' eligible for commission';
BEGIN
  DECLARE
        v_sal       NUMBER(7,2) := 50000;
        v_comm      NUMBER(7,2) := 0;
        v_total_comp  NUMBER(7,2) := v_sal + v_comm;
  BEGIN
1       v_message := 'CLERK not'||v_message;
        outer.v_comm := v_sal * 0.30;
  END;
2 v_message := 'SALESMAN'||v_message;
END;
END outer;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Challenge: Determining Variable Scope

Evaluate the PL/SQL block in the slide. Determine each of the following values according to the rules of scoping:

1. Value of `v_message` at position 1
2. Value of `v_total_comp` at position 2
3. Value of `v_comm` at position 1
4. Value of `outer.v_comm` at position 1
5. Value of `v_comm` at position 2
6. Value of `v_message` at position 2

Answers: Determining Variable Scope

Answers to the questions of scope are as follows:

1. Value of `v_message` at position 1: **`CLERK` not eligible for commission**

2. Value of `v_total_comp` at position 2: **Error. `v_total_comp` is not visible here because it is defined within the inner block.**

3. Value of `v_comm` at position 1: **0**

4. Value of `outer.v_comm` at position 1: **12000**

5. Value of `v_comm` at position 2: **15000**

6. Value of `v_message` at position 2: **`SALESMANCLERK` not eligible for commission**

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Agenda

- Writing executable statements in a PL/SQL block
- Writing nested blocks
- Using operators and developing readable code

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Operators in PL/SQL

- Logical
- Arithmetic
- Concatenation
- Parentheses to control order of operations

Same as in SQL

- Exponential operator (**)

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Operators in PL/SQL

The operations in an expression are performed in a particular order depending on their precedence (priority). The following table shows the default order of operations from high priority to low priority:

Operator	Operation
**	Exponentiation
+, -	Identity, negation
*, /	Multiplication, division
+, -, ||	Addition, subtraction, concatenation
=, <, >, <=, >=, <>, !=, ~=, ^=, IS NULL, LIKE, BETWEEN, IN	Comparison
NOT	Logical negation
AND	Conjunction
OR	Inclusion

Operators in PL/SQL: Examples

- Increment the counter for a loop.

```
loop_count := loop_count + 1;
```

- Set the value of a Boolean flag.

```
good_sal := sal BETWEEN 50000 AND 150000;
```

- Validate whether an employee number contains a value.

```
valid  := (empno IS NOT NULL);
```

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a nontransferable license to use this Student Guide.

Operators in PL/SQL (continued)

When you are working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Comparisons involving nulls always yield `NULL`.
- Applying the logical operator `NOT` to a null yields `NULL`.
- In conditional control statements, if the condition yields `NULL`, its associated sequence of statements is not executed.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Programming Guidelines

Make code maintenance easier by:

- Documenting code with comments
- Developing a case convention for the code
- Developing naming conventions for identifiers and other objects
- Enhancing readability by indenting

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Programming Guidelines

Follow programming guidelines shown in the slide to produce clear code and reduce maintenance when developing a PL/SQL block.

Code Conventions

The following table provides guidelines for writing code in uppercase or lowercase characters to help distinguish keywords from named objects.

Category	Case Convention	Examples
SQL statements	Uppercase	`SELECT`, `INSERT`
PL/SQL keywords	Uppercase	`DECLARE`, `BEGIN`, `IF`
Data types	Uppercase	`VARCHAR2`, `BOOLEAN`
Identifiers and parameters	Lowercase	`v_sal`, `emp_cursor`, `g_sal`, `p_empno`
Database tables	Lowercase, plural	`employees`, `departments`
Database columns	Lowercase, singular	`employee_id`, `department_id`

Indenting Code

For clarity, indent each level of code.

```
BEGIN
  IF x=0 THEN
     y:=1;
  END IF;
END;
/
```

```
DECLARE
  deptno        NUMBER(4);
  location_id   NUMBER(4);
BEGIN
  SELECT  department_id,
          location_id
  INTO    deptno,
          location_id
  FROM    departments
  WHERE   department_name
          = 'Sales';
...
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Indenting Code

For clarity and enhanced readability, indent each level of code. To show structure, you can divide lines by using carriage returns and you can indent lines by using spaces and tabs. Compare the following `IF` statements for readability:

```
IF x>y THEN max:=x;ELSE max:=y;END IF;
```

```
IF x > y THEN
  max := x;
ELSE
  max := y;
END IF;
```

Quiz

You can use most SQL single-row functions such as number, character, conversion, and date single-row functions in PL/SQL expressions.

1. True
2. False

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Answer: 1

SQL Functions in PL/SQL

SQL provides several predefined functions that can be used in SQL statements. Most of these functions (such as single-row number and character functions, data type conversion functions, and date and time-stamp functions) are valid in PL/SQL expressions.

The following functions are not available in procedural statements:

- `DECODE`
- Group functions: `AVG`, `MIN`, `MAX`, `COUNT`, `SUM`, `STDDEV`, and `VARIANCE`
  Group functions apply to groups of rows in a table and are, therefore, available only in SQL statements in a PL/SQL block. The functions mentioned here are only a subset of the complete list.

Summary

In this lesson, you should have learned how to:

- Identify lexical units in a PL/SQL block
- Use built-in SQL functions in PL/SQL
- Write nested blocks to break logically related functionalities
- Decide when to perform explicit conversions
- Qualify variables in nested blocks
- Use sequences in PL/SQL expressions

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Summary

Because PL/SQL is an extension of SQL, the general syntax rules that apply to SQL also apply to PL/SQL.

A block can have any number of nested blocks defined within its executable part. Blocks defined within a block are called subblocks. You can nest blocks only in the executable part of a block. Because the exception section is also a part of the executable section, it can also contain nested blocks. Ensure correct scope and visibility of the variables when you have nested blocks. Avoid using the same identifiers in the parent and child blocks.

Most of the functions available in SQL are also valid in PL/SQL expressions. Conversion functions convert a value from one data type to another. Comparison operators compare one expression with another. The result is always `TRUE`, `FALSE`, or `NULL`. Typically, you use comparison operators in conditional control statements and in the `WHERE` clause of SQL data manipulation statements. The relational operators enable you to compare arbitrarily complex expressions.

Practice 3: Overview

This practice covers the following topics:

- Reviewing scoping and nesting rules
- Writing and testing PL/SQL blocks

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practice 3: Overview

Exercises 1 and 2 are paper based.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

4

Interacting with Oracle Database Server: SQL Statements in PL/SQL Programs

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Objectives

After completing this lesson, you should be able to do the following:

- Determine the SQL statements that can be directly included in a PL/SQL executable block
- Manipulate data with DML statements in PL/SQL
- Use transaction control statements in PL/SQL
- Make use of the `INTO` clause to hold the values returned by a SQL statement
- Differentiate between implicit cursors and explicit cursors
- Use SQL cursor attributes

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

In this lesson, you learn to embed standard SQL `SELECT`, `INSERT`, `UPDATE`, `DELETE`, and `MERGE` statements in PL/SQL blocks. You learn how to include data definition language (DDL) and transaction control statements in PL/SQL. You learn the need for cursors and differentiate between the two types of cursors. The lesson also presents the various SQL cursor attributes that can be used with implicit cursors.

Agenda

- Retrieving data with PL/SQL
- Manipulating data with PL/SQL
- Introducing SQL cursors

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

SQL Statements in PL/SQL

- Retrieve a row from the database by using the `SELECT` command.
- Make changes to rows in the database by using DML commands.
- Control a transaction with the `COMMIT`, `ROLLBACK`, or `SAVEPOINT` command.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL Statements in PL/SQL

In a PL/SQL block, you use SQL statements to retrieve and modify data from the database table. PL/SQL supports data manipulation language (DML) and transaction control commands. You can use DML commands to modify the data in a database table. However, remember the following points while using DML statements and transaction control commands in PL/SQL blocks:

- The `END` keyword signals the end of a PL/SQL block, not the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks.
- PL/SQL does not directly support data definition language (DDL) statements such as `CREATE TABLE`, `ALTER TABLE`, or `DROP TABLE`. PL/SQL supports early binding, which cannot happen if applications have to create database objects at run time by passing values. DDL statements cannot be directly executed. These statements are dynamic SQL statements. Dynamic SQL statements are built as character strings at run time and can contain placeholders for parameters. Therefore, you can use dynamic SQL to execute your DDL statements in PL/SQL. The details of working with dynamic SQL is covered in the course titled *Oracle Database 11g: Develop PL/SQL Program Units*.
- PL/SQL does not directly support data control language (DCL) statements such as `GRANT` or `REVOKE`. You can use dynamic SQL to execute them.

SELECT Statements in PL/SQL

Retrieve data from the database with a SELECT statement.

Syntax:

```
SELECT   select_list
INTO     {variable_name[, variable_name]...
         | record_name}
FROM     table
[WHERE   condition];
```

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

SELECT Statements in PL/SQL

Use the SELECT statement to retrieve data from the database.

select_list	List of at least one column; can include SQL expressions, row functions, or group functions
variable_name	Scalar variable that holds the retrieved value
record_name	PL/SQL record that holds the retrieved values
table	Specifies the database table name
condition	Is composed of column names, expressions, constants, and comparison operators, including PL/SQL variables and constants

Guidelines for Retrieving Data in PL/SQL

- Terminate each SQL statement with a semicolon (;).
- Every value retrieved must be stored in a variable by using the INTO clause.
- The WHERE clause is optional and can be used to specify input variables, constants, literals, and PL/SQL expressions. However, when you use the INTO clause, you should fetch only one row; using the WHERE clause is required in such cases.

`SELECT` Statements in PL/SQL (continued)

- Specify the same number of variables in the `INTO` clause as the number of database columns in the `SELECT` clause. Be sure that they correspond positionally and that their data types are compatible.
- Use group functions, such as `SUM`, in a SQL statement, because group functions apply to groups of rows in a table.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SELECT Statements in PL/SQL

- The INTO clause is required.
- Queries must return only one row.

```
DECLARE
 v_fname VARCHAR2(25);
BEGIN
 SELECT first_name INTO v_fname
 FROM employees WHERE employee_id=200;
 DBMS_OUTPUT.PUT_LINE(' First Name is : '||v_fname);
END;
/
```

```
anonymous block completed
 First Name is : Jennifer
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SELECT Statements in PL/SQL (continued)

INTO Clause

The INTO clause is mandatory and occurs between the SELECT and FROM clauses. It is used to specify the names of variables that hold the values that SQL returns from the SELECT clause. You must specify one variable for each item selected, and the order of the variables must correspond with the items selected.

Use the INTO clause to populate either PL/SQL variables or host variables.

Queries Must Return Only One Row

SELECT statements within a PL/SQL block fall into the ANSI classification of embedded SQL, for which the following rule applies: Queries must return only one row. A query that returns more than one row or no row generates an error.

PL/SQL manages these errors by raising standard exceptions, which you can handle in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions. Include a WHERE condition in the SQL statement so that the statement returns a single row. You learn about exception handling in the lesson titled "Handling Exceptions."

Note: In all cases where DBMS_OUTPUT.PUT_LINE is used in the code examples, the SET SERVEROUTPUT ON statement precedes the block.

SELECT Statements in PL/SQL (continued)

How to Retrieve Multiple Rows from a Table and Operate on the Data

A SELECT statement with the INTO clause can retrieve only one row at a time. If your requirement is to retrieve multiple rows and operate on the data, you can make use of explicit cursors. You are introduced to cursors later in this lesson and learn about explicit cursors in the lesson titled "Using Explicit Cursors."

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Retrieving Data in PL/SQL: Example

Retrieve `hire_date` and `salary` for the specified employee.

```
DECLARE
 v_emp_hiredate   employees.hire_date%TYPE;
 v_emp_salary     employees.salary%TYPE;
BEGIN
  SELECT   hire_date, salary
  INTO     v_emp_hiredate, v_emp_salary
  FROM     employees
  WHERE    employee_id = 100;
  DBMS_OUTPUT.PUT_LINE ('Hire date is :'|| v_emp_hiredate);
  DBMS_OUTPUT.PUT_LINE ('Salary is :'|| v_emp_ salary);
END;
/
```

```
anonymous block completed
 Hire date is : 17-JUN-87
 Salary  is : 24000
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Retrieving Data in PL/SQL

In the example in the slide, the `v_emp_hiredate` and `v_emp_salary` variables are declared in the declarative section of the PL/SQL block. In the executable section, the values of the `hire_date` and `salary` columns for the employee with the `employee_id` `100` are retrieved from the `employees` table. Next, they are stored in the `emp_hiredate` and `emp_salary` variables, respectively. Observe how the `INTO` clause, along with the `SELECT` statement, retrieves the database column values and stores them in the PL/SQL variables.

Note: The `SELECT` statement retrieves `hire_date`, and then `salary`. The variables in the `INTO` clause must thus be in the same order. For example, if you exchange `v_emp_hiredate` and `v_emp_salary` in the statement in the slide, the statement results in an error.

Retrieving Data in PL/SQL

Return the sum of salaries for all the employees in the specified department.

Example:

```
DECLARE
   v_sum_sal    NUMBER(10,2);
   v_deptno     NUMBER NOT NULL := 60;
BEGIN
  SELECT  SUM(salary)  -- group function
  INTO v_sum_sal  FROM employees
  WHERE     department_id = v_deptno;
  DBMS_OUTPUT.PUT_LINE ('The sum of salary is ' || v_sum_sal);
END;
```

```
anonymous block completed
The sum of salary is 28800
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Retrieving Data in PL/SQL (continued)

In the example in the slide, the `v_sum_sal` and `v_deptno` variables are declared in the declarative section of the PL/SQL block. In the executable section, the total salary for the employees in the department with `department_id` 60 is computed using the SQL aggregate function `SUM`. The calculated total salary is assigned to the `v_sum_sal` variable.

Note: Group functions cannot be used in PL/SQL syntax. They must be used in SQL statements within a PL/SQL block as shown in the example in the slide.

For instance, you *cannot* use group functions using the following syntax:

```
V_sum_sal := SUM(employees.salary);
```

Naming Ambiguities

```
DECLARE
  hire_date       employees.hire_date%TYPE;
  sysdate         hire_date%TYPE;
  employee_id     employees.employee_id%TYPE := 176;
BEGIN
  SELECT      hire_date, sysdate
  INTO        hire_date, sysdate
  FROM        employees
  WHERE       employee_id = employee_id;
END;
/
```

```
Error report:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 6
01422. 00000 -  "exact fetch returns more than requested number of rows"
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur.84@gmail.com) has a non-transferable license to use this Student Guide.

Naming Ambiguities

In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables.

The example shown in the slide is defined as follows: Retrieve the hire date and today's date from the `employees` table for `employee_id` `176`. This example raises an unhandled run-time exception because, in the `WHERE` clause, the PL/SQL variable names are the same as the database column names in the `employees` table.

The following `DELETE` statement removes all employees from the `employees` table, where the last name is not null (not just "King"), because the Oracle Server assumes that both occurrences of `last_name` in the `WHERE` clause refer to the database column:

```
DECLARE
   last_name VARCHAR2(25) := 'King';
BEGIN
   DELETE FROM employees WHERE last_name = last_name;
. . .
```

Naming Conventions

- Use a naming convention to avoid ambiguity in the `WHERE` clause.
- Avoid using database column names as identifiers.
- Syntax errors can arise because PL/SQL checks the database first for a column in the table.
- The names of local variables and formal parameters take precedence over the names of database *tables*.
- The names of database table *columns* take precedence over the names of local variables.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (rajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Naming Conventions

Avoid ambiguity in the `WHERE` clause by adhering to a naming convention that distinguishes database column names from PL/SQL variable names.

- Database columns and identifiers should have distinct names.
- Syntax errors can arise because PL/SQL checks the database first for a column in the table.

Note: There is no possibility of ambiguity in the `SELECT` clause because any identifier in the `SELECT` clause must be a database column name. There is no possibility of ambiguity in the `INTO` clause because identifiers in the `INTO` clause must be PL/SQL variables. The possibility of confusion is present only in the `WHERE` clause.

Agenda

- Retrieving data with PL/SQL
- Manipulating data with PL/SQL
- Introducing SQL cursors

ORACLE

Copyright © 2009, Oracle. All rights reserved.

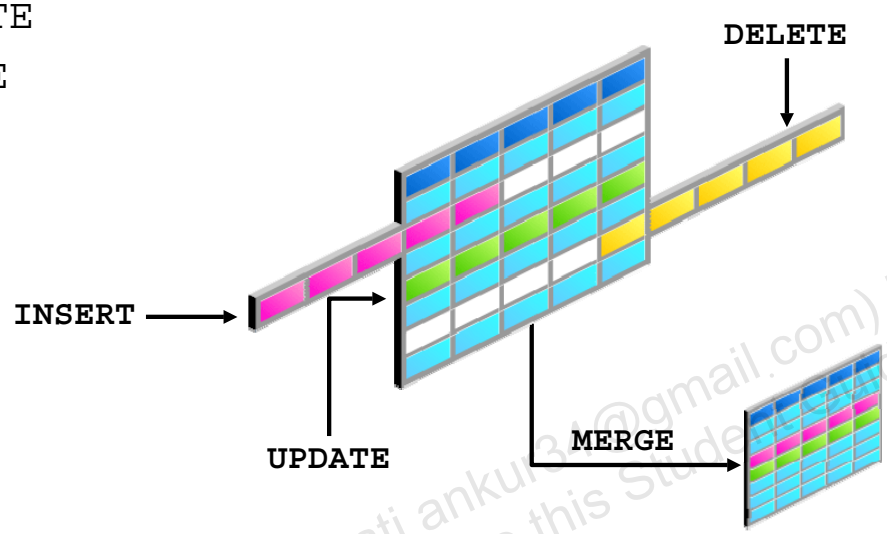
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Using PL/SQL to Manipulate Data

Make changes to database tables by using DML commands:

- INSERT
- UPDATE
- DELETE
- MERGE



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur84@gmail.com) has a non-transferable license to use this Student Guide.

Using PL/SQL to Manipulate Data

You manipulate data in the database by using DML commands. You can issue DML commands such as `INSERT`, `UPDATE`, `DELETE`, and `MERGE` without restriction in PL/SQL. Row locks (and table locks) are released by including the `COMMIT` or `ROLLBACK` statements in the PL/SQL code.

- The `INSERT` statement adds new rows to the table.
- The `UPDATE` statement modifies existing rows in the table.
- The `DELETE` statement removes rows from the table.
- The `MERGE` statement selects rows from one table to update or insert into another table. The decision whether to update or insert into the target table is based on a condition in the `ON` clause.

Note: `MERGE` is a deterministic statement. That is, you cannot update the same row of the target table multiple times in the same `MERGE` statement. You must have `INSERT` and `UPDATE` object privileges on the target table and `SELECT` privilege on the source table.

Inserting Data: Example

Add new employee information to the `EMPLOYEES` table.

```
BEGIN
 INSERT INTO employees
  (employee_id, first_name, last_name, email,
   hire_date, job_id, salary)
   VALUES(employees_seq.NEXTVAL, 'Ruth', 'Cores',
   'RCORES',CURRENT_DATE, 'AD_ASST', 4000);
END;
/
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Inserting Data

In the example in the slide, an `INSERT` statement is used within a PL/SQL block to insert a record into the `employees` table. While using the `INSERT` command in a PL/SQL block, you can:

- Use SQL functions such as `USER` and `CURRENT_DATE`
- Generate primary key values by using existing database sequences
- Derive values in the PL/SQL block

Note: The data in the `employees` table needs to remain unchanged. Even though the `employees` table is not read-only, inserting, updating, and deleting are not allowed on this table to ensure consistency of output, as shown in code example `code_04_15_s.sql`.

Updating Data: Example

Increase the salary of all employees who are stock clerks.

```
DECLARE
  sal_increase   employees.salary%TYPE := 800;
BEGIN
  UPDATE      employees
  SET         salary = salary + sal_increase
  WHERE       job_id = 'ST_CLERK';
END;
/
```

```
anonymous block completed
FIRST_NAME           SALARY
-------------------- ----------------------
Julia                4000
Irene                3500
James                3200
Steven               3000
...
Curtis               3900
Randall              3400
Peter                3300

20 rows selected
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Updating Data

There may be ambiguity in the `SET` clause of the `UPDATE` statement because, although the identifier on the left of the assignment operator is always a database column, the identifier on the right can be either a database column or a PL/SQL variable. Recall that if column names and identifier names are identical in the `WHERE` clause, the Oracle Server looks to the database first for the name.

Remember that the `WHERE` clause is used to determine the rows that are affected. If no rows are modified, no error occurs (unlike the `SELECT` statement in PL/SQL).

Note: PL/SQL variable assignments always use `:=`, and SQL column assignments always use `=`.

Deleting Data: Example

Delete rows that belong to department 10 from the `employees` table.

```
DECLARE
  deptno   employees.department_id%TYPE := 10;
BEGIN
  DELETE FROM   employees
  WHERE  department_id = deptno;
END;
/
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Deleting Data

The `DELETE` statement removes unwanted rows from a table. If the `WHERE` clause is not used, all the rows in a table can be removed if there are no integrity constraints.

Merging Rows

Insert or update rows in the `copy_emp` table to match the `employees` table.

```
BEGIN
MERGE INTO copy_emp c
     USING employees e
     ON (e.employee_id = c.empno)
   WHEN MATCHED THEN
     UPDATE SET
       c.first_name      = e.first_name,
       c.last_name       = e.last_name,
       c.email           = e.email,
       . . .
   WHEN NOT MATCHED THEN
     INSERT VALUES(e.employee_id, e.first_name, e.last_name,
          . . .,e.department_id);
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Merging Rows

The `MERGE` statement inserts or updates rows in one table by using data from another table. Each row is inserted or updated in the target table depending on an equijoin condition.

The example shown matches the `empno` column in the `copy_emp` table to the `employee_id` column in the `employees` table. If a match is found, the row is updated to match the row in the `employees` table. If the row is not found, it is inserted into the `copy_emp` table.

The complete example of using `MERGE` in a PL/SQL block is shown on the next page.

Merging Rows (continued)

```
BEGIN
MERGE INTO copy_emp c
     USING employees e
     ON (e.employee_id = c.empno)
   WHEN MATCHED THEN
     UPDATE SET
       c.first_name     = e.first_name,
       c.last_name      = e.last_name,
       c.email          = e.email,
       c.phone_number   = e.phone_number,
       c.hire_date      = e.hire_date,
       c.job_id         = e.job_id,
       c.salary         = e.salary,
       c.commission_pct = e.commission_pct,
       c.manager_id     = e.manager_id,
       c.department_id  = e.department_id
   WHEN NOT MATCHED THEN
     INSERT VALUES(e.employee_id,  e.first_name, e.last_name,
          e.email, e.phone_number, e.hire_date, e.job_id,
          e.salary, e.commission_pct, e.manager_id,
          e.department_id);
END;
/
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Agenda

- Retrieving data with PL/SQL
- Manipulating data with PL/SQL
- Introducing SQL cursors

ORACLE

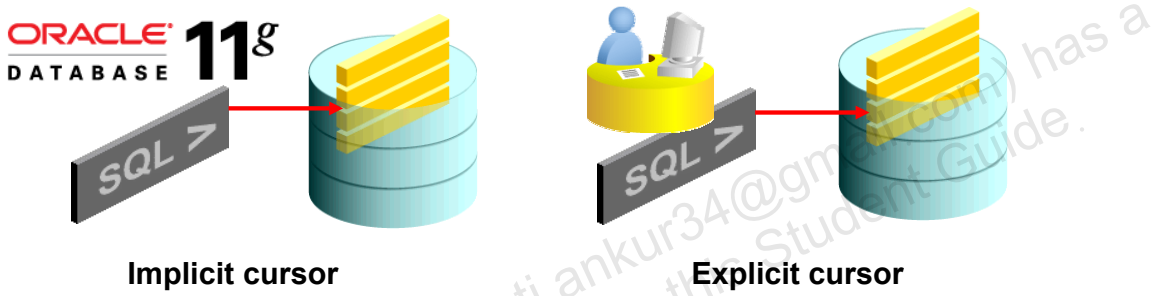
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

SQL Cursor

- A cursor is a pointer to the private memory area allocated by the Oracle Server. It is used to handle the result set of a `SELECT` statement.
- There are two types of cursors: implicit and explicit.
  - **Implicit:** Created and managed internally by the Oracle Server to process SQL statements
  - **Explicit:** Declared explicitly by the programmer



Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL Cursor

You have already learned that you can include SQL statements that return a single row in a PL/SQL block. The data retrieved by the SQL statement should be held in variables using the `INTO` clause.

Where Does the Oracle Server Process SQL Statements?

The Oracle Server allocates a private memory area called the *context area* for processing SQL statements. The SQL statement is parsed and processed in this area. The information required for processing and the information retrieved after processing are all stored in this area. You have no control over this area because it is internally managed by the Oracle Server.

A cursor is a pointer to the context area. However, this cursor is an implicit cursor and is automatically managed by the Oracle Server. When the executable block issues a SQL statement, PL/SQL creates an implicit cursor.

Types of Cursors

There are two types of cursors:

- **Implicit:** An *implicit cursor* is created and managed by the Oracle Server. You do not have access to it. The Oracle Server creates such a cursor when it has to execute a SQL statement.

SQL Cursor (continued)

Types of Cursors (continued)

- **Explicit:** As a programmer, you may want to retrieve multiple rows from a database table, have a pointer to each row that is retrieved, and work on the rows one at a time. In such cases, you can declare cursors explicitly depending on your business requirements. A cursor that is declared by programmers is called an *explicit cursor*. You declare such a cursor in the declarative section of a PL/SQL block.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL Cursor Attributes for Implicit Cursors

Using SQL cursor attributes, you can test the outcome of your SQL statements.

SQL%FOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement affected at least one row
SQL%NOTFOUND	Boolean attribute that evaluates to TRUE if the most recent SQL statement did not affect even one row
SQL%ROWCOUNT	An integer value that represents the number of rows affected by the most recent SQL statement

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL Cursor Attributes for Implicit Cursors

SQL cursor attributes enable you to evaluate what happened when an implicit cursor was last used. Use these attributes in PL/SQL statements but not in SQL statements.

You can test the SQL%ROWCOUNT, SQL%FOUND, and SQL%NOTFOUND attributes in the executable section of a block to gather information after the appropriate DML command executes. PL/SQL does not return an error if a DML statement does not affect rows in the underlying table. However, if a SELECT statement does not retrieve any rows, PL/SQL returns an exception.

Observe that the attributes are prefixed with SQL. These cursor attributes are used with implicit cursors that are automatically created by PL/SQL and for which you do not know the names. Therefore, you use SQL instead of the cursor name.

The SQL%NOTFOUND attribute is the opposite of SQL%FOUND. This attribute may be used as the exit condition in a loop. It is useful in UPDATE and DELETE statements when no rows are changed because exceptions are not returned in these cases.

You learn about explicit cursor attributes in the lesson titled “Using Explicit Cursors.”

SQL Cursor Attributes for Implicit Cursors

Delete rows that have the specified employee ID from the `employees` table. Print the number of rows deleted.

Example:

```
DECLARE
  v_rows_deleted VARCHAR2(30)
  v_empno employees.employee_id%TYPE := 176;
BEGIN
  DELETE FROM  employees
  WHERE employee_id = v_empno;
  v_rows_deleted := (SQL%ROWCOUNT ||
                     ' row deleted.');
  DBMS_OUTPUT.PUT_LINE (v_rows_deleted);

END;
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL Cursor Attributes for Implicit Cursors (continued)

The example in the slide deletes a row with `employee_id` `176` from the `employees` table. Using the `SQL%ROWCOUNT` attribute, you can print the number of rows deleted.

Quiz

When using the SELECT statement in PL/SQL, the INTO clause is required and queries can return one or more row.

1. True
2. False

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Answer: 2

INTO Clause

The INTO clause is mandatory and occurs between the SELECT and FROM clauses. It is used to specify the names of variables that hold the values that SQL returns from the SELECT clause. You must specify one variable for each item selected, and the order of the variables must correspond with the items selected.

Use the INTO clause to populate either PL/SQL variables or host variables.

Queries Must Return Only One Row

SELECT statements within a PL/SQL block fall into the ANSI classification of embedded SQL, for which the following rule applies: Queries must return only one row. A query that returns more than one row or no row generates an error.

PL/SQL manages these errors by raising standard exceptions, which you can handle in the exception section of the block with the NO_DATA_FOUND and TOO_MANY_ROWS exceptions. Include a WHERE condition in the SQL statement so that the statement returns a single row. You learn about exception handling later in the course.

Summary

In this lesson, you should have learned how to:

- Embed DML statements, transaction control statements, and DDL statements in PL/SQL
- Use the `INTO` clause, which is mandatory for all `SELECT` statements in PL/SQL
- Differentiate between implicit cursors and explicit cursors
- Use SQL cursor attributes to determine the outcome of SQL statements

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Summary

DML commands and transaction control statements can be used in PL/SQL programs without restriction. However, the DDL commands cannot be used directly.

A `SELECT` statement in a PL/SQL block can return only one row. It is mandatory to use the `INTO` clause to hold the values retrieved by the `SELECT` statement.

A cursor is a pointer to the memory area. There are two types of cursors. Implicit cursors are created and managed internally by the Oracle Server to execute SQL statements. You can use SQL cursor attributes with these cursors to determine the outcome of the SQL statement. Explicit cursors are declared by programmers.

Practice 4: Overview

This practice covers the following topics:

- Selecting data from a table
- Inserting data into a table
- Updating data in a table
- Deleting a record from a table

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Humber College

5

Writing Control Structures

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

After completing this lesson, you should be able to do the following:

- Identify the uses and types of control structures
- Construct an `IF` statement
- Use `CASE` statements and `CASE` expressions
- Construct and identify loop statements
- Use guidelines when using conditional control structures

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

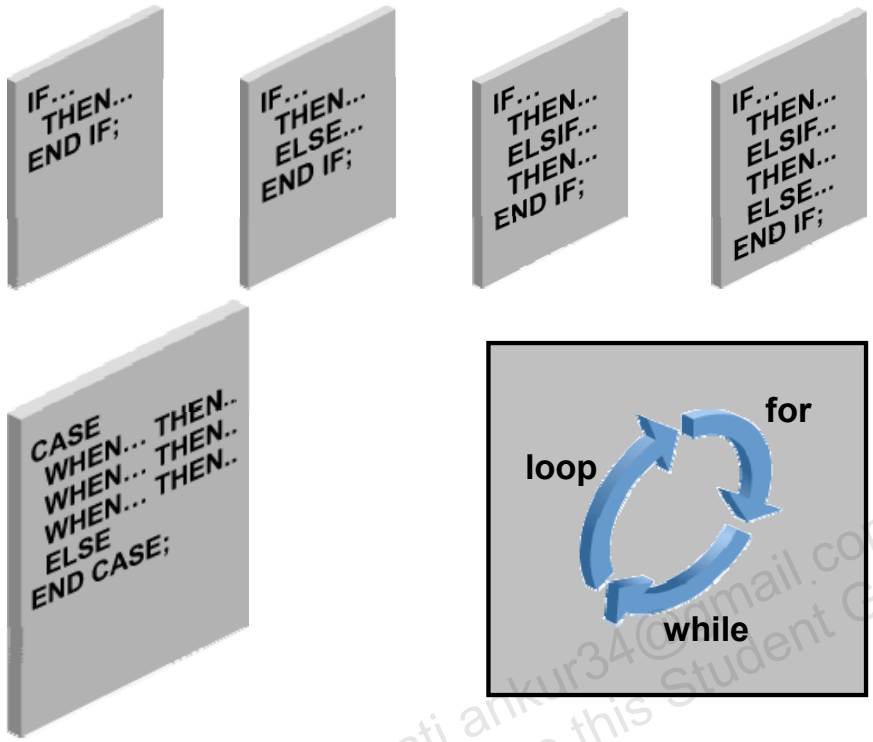
Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

You have learned to write PL/SQL blocks containing declarative and executable sections. You have also learned to include expressions and SQL statements in the executable block.

In this lesson, you learn how to use control structures such as `IF` statements, `CASE` expressions, and `LOOP` structures in a PL/SQL block.

Controlling Flow of Execution



ORACLE
Copyright © 2009, Oracle. All rights reserved.

Controlling Flow of Execution

You can change the logical flow of statements within the PL/SQL block with a number of control structures. This lesson addresses four types of PL/SQL control structures: conditional constructs with the `IF` statement, `CASE` expressions, `LOOP` control structures, and the `CONTINUE` statement.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Agenda

- Using `IF` statements
- Using `CASE` statements and `CASE` expressions
- Constructing and identifying loop statements

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

IF Statement

Syntax:

```
IF condition THEN
  statements;
[ELSIF condition THEN
  statements;]
[ELSE
  statements;]
END IF;
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

IF Statement

The structure of the PL/SQL IF statement is similar to the structure of IF statements in other procedural languages. It allows PL/SQL to perform actions selectively based on conditions.

In the syntax:

condition	Is a Boolean variable or expression that returns TRUE, FALSE, or NULL
THEN	Introduces a clause that associates the Boolean expression with the sequence of statements that follows it
statements	Can be one or more PL/SQL or SQL statements. (They may include additional IF statements containing several nested IF, ELSE, and ELSIF statements.) The statements in the THEN clause are executed only if the condition in the associated IF clause evaluates to TRUE.

IF Statement (continued)

In the syntax:

ELSIF	Is a keyword that introduces a Boolean expression (If the first condition yields FALSE or NULL, the ELSIF keyword introduces additional conditions.)
ELSE	Introduces the default clause that is executed if and only if none of the earlier predicates (introduced by IF and ELSIF) are TRUE. The tests are executed in sequence so that a later predicate that might be true is preempted by an earlier predicate that is true.
END IF	Marks the end of an IF statement

Note: ELSIF and ELSE are optional in an IF statement. You can have any number of ELSIF keywords but only one ELSE keyword in your IF statement. END IF marks the end of an IF statement and must be terminated by a semicolon.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Simple IF Statement

```
DECLARE
  v_myage  number:=31;
BEGIN
  IF v_myage  < 11
  THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  END IF;
END;
/
```

```
anonymous block completed
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Simple IF Statement

Simple IF Example

The slide shows an example of a simple IF statement with the THEN clause.

- The v_myage variable is initialized to 31.
- The condition for the IF statement returns FALSE because v_myage is not less than 11.
- Therefore, the control never reaches the THEN clause.

Adding Conditional Expressions

An IF statement can have multiple conditional expressions related with logical operators such as AND, OR, and NOT.

For example:

```
IF (myfirstname='Christopher' AND v_myage <11)
…
```

The condition uses the AND operator and therefore, evaluates to TRUE only if both conditions are evaluated as TRUE. There is no limitation on the number of conditional expressions. However, these statements must be related with appropriate logical operators.

IF THEN ELSE Statement

```
DECLARE
  v_myage  number:=31;
BEGIN
  IF v_myage  < 11
   THEN
     DBMS_OUTPUT.PUT_LINE(' I am a child ');
  ELSE
   DBMS_OUTPUT.PUT_LINE(' I am not a child ');
  END IF;
END;
/
```

```
anonymous block completed
 I am not a child
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

IF THEN ELSE Statement

An ELSE clause is added to the code in the previous slide. The condition has not changed and, therefore, still evaluates to FALSE. Recall that the statements in the THEN clause are executed only if the condition returns TRUE. In this case, the condition returns FALSE and the control moves to the ELSE statement.

The output of the block is shown below the code.

IF ELSIF ELSE Clause

```
DECLARE
  v_myage number:=31;
BEGIN
  IF v_myage  < 11 THEN
       DBMS_OUTPUT.PUT_LINE(' I am a child ');
    ELSIF v_myage  < 20 THEN
       DBMS_OUTPUT.PUT_LINE(' I am young ');
    ELSIF v_myage  < 30 THEN
       DBMS_OUTPUT.PUT_LINE(' I am in my twenties');
    ELSIF v_myage  < 40 THEN
       DBMS_OUTPUT.PUT_LINE(' I am in my thirties');
    ELSE
       DBMS_OUTPUT.PUT_LINE(' I am always young ');
  END IF;
END;
/
```

```
anonymous block completed
 I am in my thirties
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

IF ELSIF ELSE Clause

The `IF` clause may contain multiple `ELSIF` clauses and an `ELSE` clause. The example illustrates the following characteristics of these clauses:

- The `ELSIF` clauses can have conditions, unlike the `ELSE` clause.
- The condition for `ELSIF` should be followed by the `THEN` clause, which is executed if the condition for `ELSIF` returns `TRUE`.
- When you have multiple `ELSIF` clauses, if the first condition is `FALSE` or `NULL`, the control shifts to the next `ELSIF` clause.
- Conditions are evaluated one by one from the top.
- If all conditions are `FALSE` or `NULL`, the statements in the `ELSE` clause are executed.
- The final `ELSE` clause is optional.

In the example, the output of the block is shown below the code.

NULL Value in IF Statement

```
DECLARE
  v_myage  number;
BEGIN
  IF v_myage  < 11 THEN
    DBMS_OUTPUT.PUT_LINE(' I am a child ');
  ELSE
    DBMS_OUTPUT.PUT_LINE(' I am not a child ');
  END IF;
END;
/
```

```
anonymous block completed
 I am not a child
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

NULL Value in IF Statement

In the example shown in the slide, the variable `v_myage` is declared but not initialized. The condition in the `IF` statement returns `NULL` rather than `TRUE` or `FALSE`. In such a case, the control goes to the `ELSE` statement.

Guidelines

- You can perform actions selectively based on conditions that are being met.
- When you write code, remember the spelling of the keywords:
  - `ELSIF` is one word.
  - `END IF` is two words.
- If the controlling Boolean condition is `TRUE`, the associated sequence of statements is executed; if the controlling Boolean condition is `FALSE` or `NULL`, the associated sequence of statements is passed over. Any number of `ELSIF` clauses is permitted.
- Indent the conditionally executed statements for clarity.

Agenda

- Using `IF` statements
- Using `CASE` statements and `CASE` expressions
- Constructing and identifying loop statements

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

CASE Expressions

- A CASE expression selects a result and returns it.
- To select the result, the CASE expression uses expressions. The value returned by these expressions is used to select one of several alternatives.

```
CASE selector
   WHEN expression1 THEN result1
   WHEN expression2 THEN result2
   ...
   WHEN expressionN THEN resultN
  [ELSE resultN+1]
END;
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

CASE Expressions

A CASE expression returns a result based on one or more alternatives. To return the result, the CASE expression uses a *selector*, which is an expression whose value is used to return one of several alternatives. The selector is followed by one or more WHEN clauses that are checked sequentially. The value of the selector determines which result is returned. If the value of the selector equals the value of a WHEN clause expression, that WHEN clause is executed and that result is returned.

PL/SQL also provides a searched CASE expression, which has the form:

```
CASE
   WHEN search_condition1 THEN result1
   WHEN search_condition2 THEN result2
   ...
   WHEN search_conditionN THEN resultN
  [ELSE resultN+1]
END;
```

A searched CASE expression has no selector. Furthermore, the WHEN clauses in CASE expressions contain search conditions that yield a Boolean value rather than expressions that can yield a value of any type.

CASE Expressions: Example

```
SET VERIFY OFF
DECLARE
   v_grade  CHAR(1) := UPPER('&grade');
   v_appraisal VARCHAR2(20);
BEGIN
   v_appraisal := CASE v_grade
         WHEN 'A' THEN 'Excellent'
         WHEN 'B' THEN 'Very Good'
         WHEN 'C' THEN 'Good'
         ELSE 'No such grade'
      END;
DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade  || '
                      Appraisal ' || v_appraisal);
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

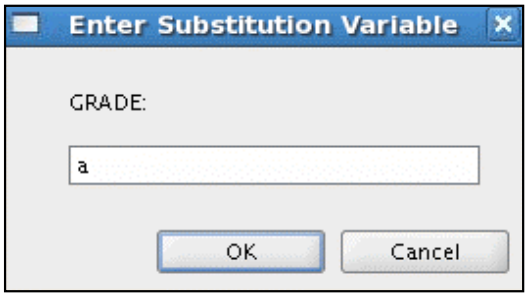
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

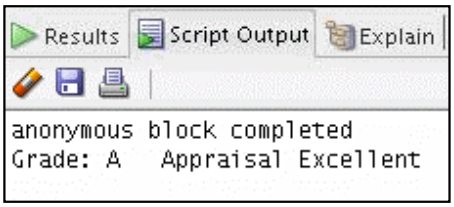
CASE Expressions: Example

In the example in the slide, the `CASE` expression uses the value in the `v_grade` variable as the expression. This value is accepted from the user by using a substitution variable. Based on the value entered by the user, the `CASE` expression returns the value of the `v_appraisal` variable based on the value of the `v_grade` value.

Result

When you enter `a` or `A` for `v_grade`, as shown in the Substitution Variable window, the output of the example is as follows:





Searched CASE Expressions

```
DECLARE
   v_grade  CHAR(1) := UPPER('&grade');
   v_appraisal VARCHAR2(20);
BEGIN
    v_appraisal := CASE
         WHEN v_grade  = 'A' THEN 'Excellent'
         WHEN v_grade  IN ('B','C') THEN 'Good'
         ELSE 'No such grade'
     END;
   DBMS_OUTPUT.PUT_LINE ('Grade: '|| v_grade  || '
                  Appraisal ' || v_appraisal);
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

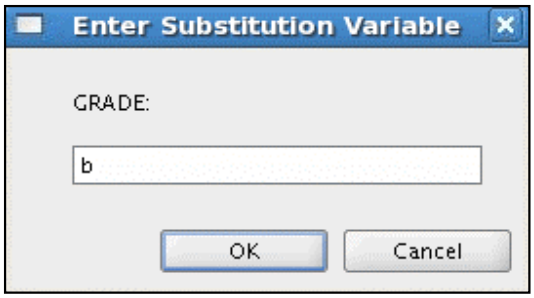
Searched CASE Expressions

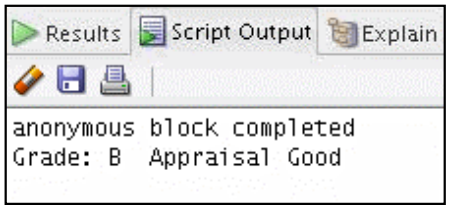
In the previous example, you saw a single test expression, the `v_grade` variable. The `WHEN` clause compared a value against this test expression.

In searched `CASE` statements, you do not have a test expression. Instead, the `WHEN` clause contains an expression that results in a Boolean value. The same example is rewritten in this slide to show searched `CASE` statements.

Result

The output of the example is as follows when you enter `b` or `B` for `v_grade`:





CASE Statement

```
DECLARE
   v_deptid NUMBER;
   v_deptname VARCHAR2(20);
   v_emps NUMBER;
   v_mngid NUMBER:= 108;
BEGIN
  CASE  v_mngid
   WHEN  108 THEN
    SELECT department_id, department_name
     INTO v_deptid, v_deptname FROM departments
     WHERE manager_id=108;
    SELECT count(*) INTO v_emps FROM employees
     WHERE department_id=v_deptid;
   WHEN  200 THEN
    ...
 END CASE;
DBMS_OUTPUT.PUT_LINE ('You are working in the '|| v_deptname||
' department. There are '||v_emps ||' employees in this
department');
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

A. Kur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

CASE Statement

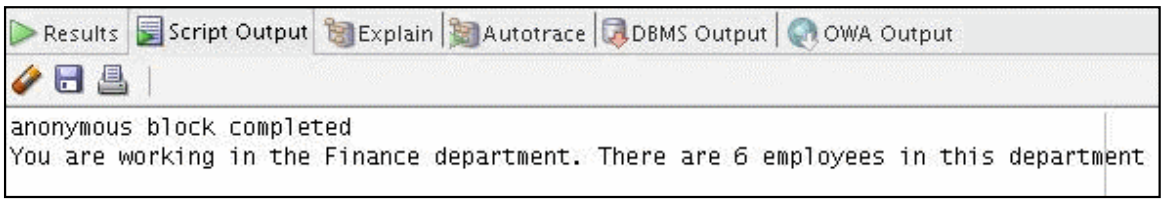
Recall the use of the IF statement. You may include *n* number of PL/SQL statements in the THEN clause and also in the ELSE clause. Similarly, you can include statements in the CASE statement, which is more readable compared to multiple IF and ELSIF statements.

How a CASE Expression Differs from a CASE Statement

A CASE expression evaluates the condition and returns a value, whereas a CASE statement evaluates the condition and performs an action. A CASE statement can be a complete PL/SQL block.

- CASE statements end with END CASE;
- CASE expressions end with END;

The output of the slide code example is as follows:



Note: Whereas an IF statement is able to do nothing (the conditions could be all false and the ELSE clause is not mandatory), a CASE statement must execute some PL/SQL statement.

Handling Nulls

When you are working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Simple comparisons involving nulls always yield `NULL`.
- Applying the logical operator `NOT` to a null yields `NULL`.
- If the condition yields `NULL` in conditional control statements, its associated sequence of statements is not executed.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Handling Nulls

Consider the following example:

```
x := 5;
y := NULL;
...
IF x != y THEN  -- yields NULL, not TRUE
  --  sequence_of_statements that are not executed
END IF;
```

You may expect the sequence of statements to execute because `x` and `y` seem unequal. But nulls are indeterminate. Whether or not `x` is equal to `y` is unknown. Therefore, the `IF` condition yields `NULL` and the sequence of statements is bypassed.

```
a := NULL;
b := NULL;
...
IF a = b THEN  -- yields NULL, not TRUE
  --  sequence_of_statements that are not executed
END IF;
```

In the second example, you may expect the sequence of statements to execute because `a` and `b` seem equal. But, again, equality is unknown, so the `IF` condition yields `NULL` and the sequence of statements is bypassed.

Logic Tables

Build a simple Boolean condition with a comparison operator.

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (rajapati.ankur34@gmail.com) has a nontransferable license to use this Student Guide.

Logic Tables

You can build a simple Boolean condition by combining number, character, and date expressions with comparison operators.

You can build a complex Boolean condition by combining simple Boolean conditions with the logical operators `AND`, `OR`, and `NOT`. The logical operators are used to check the Boolean variable values and return `TRUE`, `FALSE`, or `NULL`. In the logic tables shown in the slide:

- `FALSE` takes precedence in an `AND` condition, and `TRUE` takes precedence in an `OR` condition
- `AND` returns `TRUE` only if both of its operands are `TRUE`
- `OR` returns `FALSE` only if both of its operands are `FALSE`
- `NULL AND TRUE` always evaluates to `NULL` because it is not known whether the second operand evaluates to `TRUE`

Note: The negation of `NULL` (`NOT NULL`) results in a null value because null values are indeterminate.

Boolean Expressions or Logical Expression?

What is the value of `flag` in each case?

```
flag := reorder_flag AND available_flag;
```

REORDER_FLAG	AVAILABLE_FLAG	FLAG
TRUE	TRUE	? (1)
TRUE	FALSE	? (2)
NULL	TRUE	? (3)
NULL	FALSE	? (4)

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Boolean Expressions or Logical Expression?

The AND logic table can help you to evaluate the possibilities for the Boolean condition in the slide.

Answers

1. TRUE
2. FALSE
3. NULL
4. FALSE

Agenda

- Using `IF` statements
- Using `CASE` statements and `CASE` expressions
- Constructing and identifying loop statements

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Iterative Control: `LOOP` Statements

- Loops repeat a statement (or a sequence of statements) multiple times.
- There are three loop types:
  - Basic loop
  - `FOR` loop
  - `WHILE` loop

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Iterative Control: `LOOP` Statements

PL/SQL provides several facilities to structure loops to repeat a statement or sequence of statements multiple times. Loops are mainly used to execute statements repeatedly until an exit condition is reached. It is mandatory to have an exit condition in a loop; otherwise, the loop is infinite.

Looping constructs are the third type of control structures. PL/SQL provides the following types of loops:

- Basic loop that performs repetitive actions without overall conditions
- `FOR` loops that perform iterative actions based on a count
- `WHILE` loops that perform iterative actions based on a condition

Note: An `EXIT` statement can be used to terminate loops. A basic loop must have an `EXIT`. The cursor `FOR` loop (which is another type of `FOR` loop) is discussed in the lesson titled "Using Explicit Cursors."

Basic Loops

Syntax:

```
LOOP
  statement1;
  . . .
  EXIT [WHEN condition];
END LOOP;
```

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Basic Loops

The simplest form of a `LOOP` statement is the basic loop, which encloses a sequence of statements between the `LOOP` and `END LOOP` keywords. Each time the flow of execution reaches the `END LOOP` statement, control is returned to the corresponding `LOOP` statement above it. A basic loop allows execution of its statements at least once, even if the `EXIT` condition is already met upon entering the loop. Without the `EXIT` statement, the loop would be infinite.

`EXIT` Statement

You can use the `EXIT` statement to terminate a loop. Control passes to the next statement after the `END LOOP` statement. You can issue `EXIT` either as an action within an `IF` statement or as a stand-alone statement within the loop. The `EXIT` statement must be placed inside a loop. In the latter case, you can attach a `WHEN` clause to enable conditional termination of the loop. When the `EXIT` statement is encountered, the condition in the `WHEN` clause is evaluated. If the condition yields `TRUE`, the loop ends and control passes to the next statement after the loop.
A basic loop can contain multiple `EXIT` statements, but it is recommended that you have only one `EXIT` point.

Basic Loop: Example

```
DECLARE
  v_countryid    locations.country_id%TYPE := 'CA';
  v_loc_id       locations.location_id%TYPE;
  v_counter      NUMBER(2) := 1;
  v_new_city     locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_loc_id FROM locations
  WHERE country_id = v_countryid;
  LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
    v_counter := v_counter + 1;
    EXIT WHEN v_counter > 3;
  END LOOP;
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Basic Loop: Example

The basic loop example shown in the slide is defined as follows: "Insert three new location IDs for the CA country code and the city of Montreal."

Note

- A basic loop allows execution of its statements until the `EXIT WHEN` condition is met.
- If the condition is placed in the loop such that it is not checked until after the loop statements execute, the loop executes at least once.
- However, if the exit condition is placed at the top of the loop (before any of the other executable statements) and if that condition is true, the loop exits and the statements never execute.

Results

To view the output, run the code example: `code_05_22_s.sql`.

WHILE Loops

Syntax:

```
WHILE condition LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

Use the WHILE loop to repeat statements while a condition is TRUE.

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

WHILE Loops

You can use the WHILE loop to repeat a sequence of statements until the controlling condition is no longer TRUE. The condition is evaluated at the start of each iteration. The loop terminates when the condition is FALSE or NULL. If the condition is FALSE or NULL at the start of the loop, no further iterations are performed. Thus, it is possible that none of the statements inside the loop are executed.

In the syntax:

condition	Is a Boolean variable or expression (TRUE, FALSE, or NULL)
statement	Can be one or more PL/SQL or SQL statements

If the variables involved in the conditions do not change during the body of the loop, the condition remains TRUE and the loop does not terminate.

Note: If the condition yields NULL, the loop is bypassed and control passes to the next statement.

WHILE Loops: Example

```
DECLARE
  v_countryid   locations.country_id%TYPE := 'CA';
  v_loc_id      locations.location_id%TYPE;
  v_new_city    locations.city%TYPE := 'Montreal';
  v_counter     NUMBER := 1;
BEGIN
  SELECT MAX(location_id) INTO v_loc_id FROM locations
  WHERE country_id = v_countryid;
  WHILE v_counter <= 3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + v_counter), v_new_city, v_countryid);
    v_counter := v_counter + 1;
  END LOOP;
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

WHILE Loops: Example

In the example in the slide, three new location IDs for the `CA` country code and the city of Montreal are added.

- With each iteration through the `WHILE` loop, a counter (`v_counter`) is incremented.
- If the number of iterations is less than or equal to the number `3`, the code within the loop is executed and a row is inserted into the `locations` table.
- After `v_counter` exceeds the number of new locations for this city and country, the condition that controls the loop evaluates to `FALSE` and the loop terminates.

Results

To view the output, run the code example: `code_05_24_s.sql`.

FOR Loops

- Use a FOR loop to shortcut the test for the number of iterations.
- Do not declare the counter; it is declared implicitly.

```
FOR counter IN [REVERSE]
    lower_bound..upper_bound LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

FOR Loops

FOR loops have the same general structure as the basic loop. In addition, they have a control statement before the LOOP keyword to set the number of iterations that the PL/SQL performs.

In the syntax:

counter	Is an implicitly declared integer whose value automatically increases or decreases (decreases if the REVERSE keyword is used) by 1 on each iteration of the loop until the upper or lower bound is reached
REVERSE	Causes the counter to decrement with each iteration from the upper bound to the lower bound **Note:** The lower bound is still referenced first.
lower_bound	Specifies the lower bound for the range of counter values
upper_bound	Specifies the upper bound for the range of counter values

Do not declare the counter. It is declared implicitly as an integer.

FOR Loops (continued)

Note: The sequence of statements is executed each time the counter is incremented, as determined by the two bounds. The lower bound and upper bound of the loop range can be literals, variables, or expressions, but they must evaluate to integers. The bounds are rounded to integers; that is, 11/3 and 8/5 are valid upper or lower bounds. The lower bound and upper bound are inclusive in the loop range. If the lower bound of the loop range evaluates to a larger integer than the upper bound, the sequence of statements is not executed.
For example, the following statement is executed only once:

```
FOR i IN 3..3
LOOP
 statement1;
END LOOP;
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

FOR Loops: Example

```
DECLARE
  v_countryid   locations.country_id%TYPE := 'CA';
  v_loc_id      locations.location_id%TYPE;
  v_new_city    locations.city%TYPE := 'Montreal';
BEGIN
  SELECT MAX(location_id) INTO v_loc_id
    FROM locations
    WHERE country_id = v_countryid;
  FOR i IN 1..3 LOOP
    INSERT INTO locations(location_id, city, country_id)
    VALUES((v_loc_id + i), v_new_city, v_countryid );
  END LOOP;
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

FOR Loops: Example

You have already learned how to insert three new locations for the CA country code and the city of Montreal by using the basic loop and the WHILE loop. The example in this slide shows how to achieve the same by using the FOR loop.

Results

To view the output, run the code example code_05_27_s.sql.

FOR Loop Rules

- Reference the counter only within the loop; it is undefined outside the loop.
- Do not reference the counter as the target of an assignment.
- Neither loop bound should be NULL.

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

FOR Loop Rules

The slide lists the guidelines to follow when writing a FOR loop.

Note: The lower and upper bounds of a LOOP statement do not need to be numeric literals. They can be expressions that convert to numeric values.

Example:

```
DECLARE
  v_lower  NUMBER := 1;
  v_upper  NUMBER := 100;
BEGIN
  FOR i IN v_lower..v_upper LOOP
  ...
  END LOOP;
END;
/
```

Suggested Use of Loops

- Use the basic loop when the statements inside the loop must execute at least once.
- Use the `WHILE` loop if the condition must be evaluated at the start of each iteration.
- Use a `FOR` loop if the number of iterations is known.

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Suggested Use of Loops

A basic loop allows the execution of its statement at least once, even if the condition is already met upon entering the loop. Without the `EXIT` statement, the loop would be infinite.

You can use the `WHILE` loop to repeat a sequence of statements until the controlling condition is no longer `TRUE`. The condition is evaluated at the start of each iteration. The loop terminates when the condition is `FALSE`. If the condition is `FALSE` at the start of the loop, no further iterations are performed.

`FOR` loops have a control statement before the `LOOP` keyword to determine the number of iterations that the PL/SQL performs. Use a `FOR` loop if the number of iterations is predetermined.

Nested Loops and Labels

- You can nest loops to multiple levels.
- Use labels to distinguish between blocks and loops.
- Exit the outer loop with the `EXIT` statement that references the label.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Nested Loops and Labels

You can nest the `FOR`, `WHILE`, and basic loops within one another. The termination of a nested loop does not terminate the enclosing loop unless an exception is raised. However, you can label loops and exit the outer loop with the `EXIT` statement.

Label names follow the same rules as the other identifiers. A label is placed before a statement, either on the same line or on a separate line. White space is insignificant in all PL/SQL parsing except inside literals. Label basic loops by placing the label before the word `LOOP` within label delimiters (<<*label*>>). In `FOR` and `WHILE` loops, place the label before `FOR` or `WHILE`.

If the loop is labeled, the label name can be included (optionally) after the `END LOOP` statement for clarity.

Nested Loops and Labels: Example

```
...
BEGIN
  <<Outer_loop>>
  LOOP
    v_counter := v_counter+1;
  EXIT WHEN v_counter>10;
    <<Inner_loop>>
    LOOP
      ...
      EXIT Outer_loop WHEN total_done = 'YES';
      -- Leave both loops
      EXIT WHEN inner_done = 'YES';
      -- Leave inner loop only
      ...
    END LOOP Inner_loop;
    ...
  END LOOP Outer_loop;
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Nested Loops and Labels: Example

In the example in the slide, there are two loops. The outer loop is identified by the label `<<Outer_Loop>>` and the inner loop is identified by the label `<<Inner_Loop>>`. The identifiers are placed before the word `LOOP` within label delimiters (*<<label>>*). The inner loop is nested within the outer loop. The label names are included after the `END LOOP` statements for clarity.

PL/SQL CONTINUE Statement

- Definition
  - Adds the functionality to begin the next loop iteration
  - Provides programmers with the ability to transfer control to the next iteration of a loop
  - Uses parallel structure and semantics to the EXIT statement
- Benefits
  - Eases the programming process
  - May provide a small performance improvement over the previous programming workarounds to simulate the CONTINUE statement

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

PL/SQL CONTINUE Statement

The CONTINUE statement enables you to transfer control within a loop back to a new iteration or to leave the loop. Many other programming languages have this functionality. With the Oracle Database 11*g* release, PL/SQL also offers this functionality. Before the Oracle Database 11*g* release, you could code a workaround by using Boolean variables and conditional statements to simulate the CONTINUE programmatic functionality. In some cases, the workarounds are less efficient.

The CONTINUE statement offers you a simplified means to control loop iterations. It may be more efficient than the previous coding workarounds.

The CONTINUE statement is commonly used to filter data within a loop body before the main processing begins.

PL/SQL CONTINUE Statement: Example 1

```
DECLARE
  v_total SIMPLE_INTEGER := 0;
BEGIN
  FOR i IN 1..10 LOOP
    v_total := v_total + i;
    dbms_output.put_line
     ('Total is: '|| v_total);
    CONTINUE WHEN i > 5;
    v_total := v_total + i;
    dbms_output.put_line
     ('Out of Loop Total is:
      '|| v_total);
  END LOOP;
END;
/
```

```
Results | Script Output | Exp
anonymous block completed
Total is: 1
Out of Loop Total is:
      2
Total is: 4
Out of Loop Total is:
      6
Total is: 9
Out of Loop Total is:
      12
Total is: 16
Out of Loop Total is:
      20
Total is: 25
Out of Loop Total is:
      30
Total is: 36
Total is: 43
Total is: 51
Total is: 60
Total is: 70
```

ORACLE
Copyright © 2009, Oracle. All rights reserved.

PL/SQL CONTINUE Statement: Example 1

In the example, there are two assignments using the `v_total` variable:

1. The first assignment is executed for each of the 10 iterations of the loop.
2. The second assignment is executed for the first five iterations of the loop. The `CONTINUE` statement transfers control within a loop back to a new iteration, so for the last five iterations of the loop, the second `TOTAL` assignment is not executed.

The end result of the `TOTAL` variable is 70.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur.a@gmail.com) has a non-transferable license to use this Student Guide.

PL/SQL CONTINUE Statement: Example 2

```
DECLARE
 v_total NUMBER := 0;
BEGIN
 <<BeforeTopLoop>>
 FOR i IN 1..10 LOOP
   v_total := v_total + 1;
   dbms_output.put_line
     ('Total is: ' || v_total);
   FOR j IN 1..10 LOOP
     CONTINUE BeforeTopLoop WHEN i + j > 5;
     v_total := v_total + 1;
   END LOOP;
 END LOOP;
END two_loop;
```

Results | Script Output | Ex

```
anonymous block completed
Total is: 1
Total is: 6
Total is: 10
Total is: 13
Total is: 15
Total is: 16
Total is: 17
Total is: 18
Total is: 19
Total is: 20
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur@hotmail.com) has a non-transferable license to use this Student Guide.

PL/SQL CONTINUE Statement: Example 2

You can use the CONTINUE statement to jump to the next iteration of an outer loop. To do this, provide the outer loop a label to identify where the CONTINUE statement should go.

The CONTINUE statement in the innermost loop terminates that loop whenever the WHEN condition is true (just like the EXIT keyword). After the innermost loop is terminated by the CONTINUE statement, control transfers to the next iteration of the outermost loop labeled BeforeTopLoop in this example.

When this pair of loops completes, the value of the TOTAL variable is 20.

You can also use the CONTINUE statement within an inner block of code, which does not contain a loop as long as the block is nested inside an appropriate outer loop.

Restrictions

- The CONTINUE statement cannot appear outside a loop at all—this generates a compiler error.
- You cannot use the CONTINUE statement to pass through a procedure, function, or method boundary—this generates a compiler error.

Quiz

There are three types of loops: basic, `FOR`, and `WHILE`.

1. True
2. False

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Answer: 1

Loop Types

PL/SQL provides the following types of loops:

- Basic loops that perform repetitive actions without overall conditions
- `FOR` loops that perform iterative actions based on a count
- `WHILE` loops that perform iterative actions based on a condition

Summary

In this lesson, you should have learned to change the logical flow of statements by using the following control structures:

- Conditional (`IF` statement)
- `CASE` expressions and `CASE` statements
- Loops:
  - Basic loop
  - `FOR` loop
  - `WHILE` loop
- `EXIT` statement
- `CONTINUE` statement

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Summary

A language can be called a programming language only if it provides control structures for the implementation of business logic. These control structures are also used to control the flow of the program. PL/SQL is a programming language that integrates programming constructs with SQL.

A conditional control construct checks for the validity of a condition and performs an action accordingly. You use the `IF` construct to perform a conditional execution of statements.

An iterative control construct executes a sequence of statements repeatedly, as long as a specified condition holds `TRUE`. You use the various loop constructs to perform iterative operations.

Practice 5: Overview

This practice covers the following topics:

- Performing conditional actions by using `IF` statements
- Performing iterative steps by using `LOOP` structures

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (rajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Practice 5: Overview

In this practice, you create the PL/SQL blocks that incorporate loops and conditional control structures. The exercises test your understanding of writing various `IF` statements and `LOOP` constructs.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

6

Working with Composite Data Types

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

After completing this lesson, you should be able to do the following:

- Describe PL/SQL collections and records
- Create user-defined PL/SQL records
- Create a PL/SQL record with the `%ROWTYPE` attribute
- Create associative arrays
  - `INDEX BY` table
  - `INDEX BY` table of records

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

You have already been introduced to composite data types. In this lesson, you learn more about composite data types and their uses.

Agenda

- Introducing composite data types
- Using PL/SQL records
  - Manipulating data with PL/SQL records
  - Advantages of the `%ROWTYPE` attribute
- Using PL/SQL collections
  - Examining associative arrays
  - Introducing nested tables
  - Introducing `VARRAY`

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Composite Data Types

- Can hold multiple values (unlike scalar types)
- Are of two types:
  - PL/SQL records
  - PL/SQL collections
    - Associative array (`INDEX BY` table)
    - Nested table
    - `VARRAY`

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Composite Data Types

You learned that variables of the scalar data type can hold only one value, whereas a variable of the composite data type can hold multiple values of the scalar data type or the composite data type. There are two types of composite data types:

- **PL/SQL records:** Records are used to treat related but dissimilar data as a logical unit. A PL/SQL record can have variables of different types. For example, you can define a record to hold employee details. This involves storing an employee number as `NUMBER`, a first name and last name as `VARCHAR2`, and so on. By creating a record to store employee details, you create a logical collective unit. This makes data access and manipulation easier.
- **PL/SQL collections:** Collections are used to treat data as a single unit. Collections are of three types:
  - Associative array
  - Nested table
  - `VARRAY`

Why Use Composite Data Types?

You have all the related data as a single unit. You can easily access and modify data. Data is easier to manage, relate, and transport if it is composite. An analogy is having a single bag for all your laptop components rather than a separate bag for each component.

PL/SQL Records or Collections?

- Use PL/SQL records when you want to store values of different data types but only one occurrence at a time.
- Use PL/SQL collections when you want to store values of the same data type.

PL/SQL Record:

TRUE	23-DEC-98	ATLANTA	

PL/SQL Collection:

1	SMITH
2	JONES
3	BENNETT
4	KRAMER

VARCHAR2

PLS_INTEGER

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

PL/SQL Records or Collections?

If both PL/SQL records and PL/SQL collections are composite types, how do you choose which one to use?

- Use PL/SQL records when you want to store values of different data types that are logically related. For example, you can create a PL/SQL record to hold employee details and indicate that all the values stored are related because they provide information about a particular employee.
- Use PL/SQL collections when you want to store values of the same data type. Note that this data type can also be of the composite type (such as records). You can define a collection to hold the first names of all employees. You may have stored *n* names in the collection; however, name 1 is not related to name 2. The relation between these names is only that they are employee names. These collections are similar to arrays in programming languages such as C, C++, and Java.

Agenda

- Examining composite data types
- Using PL/SQL records
  - Manipulating data with PL/SQL records
  - Advantages of the `%ROWTYPE` attribute
- Using PL/SQL collections
  - Examining associative arrays
  - Introducing nested tables
  - Introducing `VARRAY`

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

PL/SQL Records

- Must contain one or more components (called *fields*) of any scalar, `RECORD`, or `INDEX BY` table data type
- Are similar to structures in most third-generation languages (including C and C++)
- Are user-defined and can be a subset of a row in a table
- Treat a collection of fields as a logical unit
- Are convenient for fetching a row of data from a table for processing

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

PL/SQL Records

A record is a group of related data items stored in fields, each with its own name and data type.

- Each record defined can have as many fields as necessary.
- Records can be assigned initial values and can be defined as `NOT NULL`.
- Fields without initial values are initialized to `NULL`.
- The `DEFAULT` keyword as well as `:=` can be used in initializing fields.
- You can define `RECORD` types and declare user-defined records in the declarative part of any block, subprogram, or package.
- You can declare and reference nested records. One record can be the component of another record.

Creating a PL/SQL Record

Syntax:

1
```
TYPE type_name IS RECORD
     (field_declaration[, field_declaration]…);
```

2
```
identifier    type_name;
```

field_declaration:

```
field_name {field_type | variable%TYPE
           | table.column%TYPE | table%ROWTYPE}
           [[NOT NULL] {:= | DEFAULT} expr]
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating a PL/SQL Record

PL/SQL records are user-defined composite types. To use them, perform the following steps:

1. Define the record in the declarative section of a PL/SQL block. The syntax for defining the record is shown in the slide.
2. Declare (and optionally initialize) the internal components of this record type.

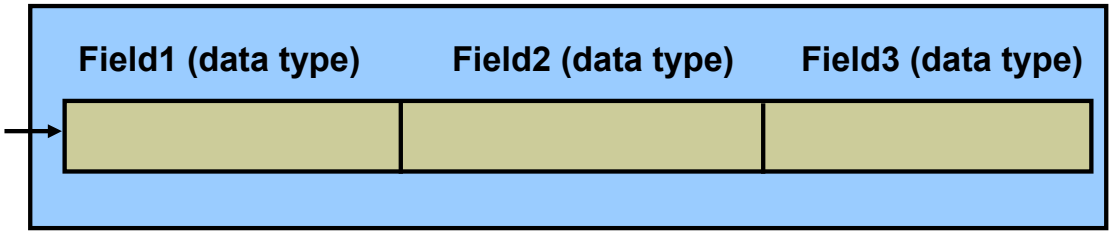
In the syntax:

type_name	Is the name of the `RECORD` type (This identifier is used to declare records.)
field_name	Is the name of a field within the record
field_type	Is the data type of the field (It represents any PL/SQL data type except `REF CURSOR`. You can use the `%TYPE` and `%ROWTYPE` attributes.)
expr	Is the *field_type* or an initial value

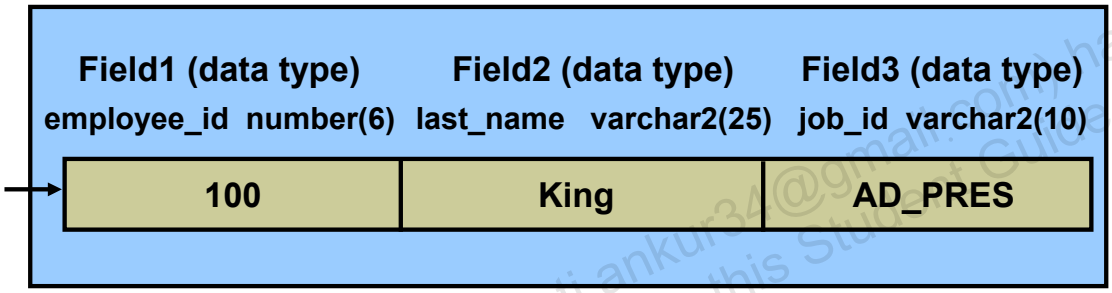
The `NOT NULL` constraint prevents assigning of nulls to the specified fields. Be sure to initialize the `NOT NULL` fields.

PL/SQL Record Structure

Field declarations:



Example:



Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

PL/SQL Record Structure

Fields in a record are accessed with the name of the record. To reference or initialize an individual field, use the dot notation:

```
record_name.field_name
```

For example, you reference the `job_id` field in the `emp_record` record as follows:

```
emp_record.job_id
```

You can then assign a value to the record field:

```
emp_record.job_id := 'ST_CLERK';
```

In a block or subprogram, user-defined records are instantiated when you enter the block or subprogram. They cease to exist when you exit the block or subprogram.

%ROWTYPE Attribute

- Declare a variable according to a collection of columns in a database table or view.
- Prefix %ROWTYPE with the database table or view.
- Fields in the record take their names and data types from the columns of the table or view.

Syntax:

```
DECLARE
   identifier  reference%ROWTYPE;
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

%ROWTYPE Attribute

You learned that %TYPE is used to declare a variable of the column type. The variable has the same data type and size as the table column. The benefit of %TYPE is that you do not have to change the variable if the column is altered. Also, if the variable is a number and is used in any calculations, you need not worry about its precision.

The %ROWTYPE attribute is used to declare a record that can hold an entire row of a table or view. The fields in the record take their names and data types from the columns of the table or view. The record can also store an entire row of data fetched from a cursor or cursor variable.

The slide shows the syntax for declaring a record. In the syntax:

identifier Is the name chosen for the record as a whole

reference Is the name of the table, view, cursor, or cursor variable on which the record is to be based (The table or view must exist for this reference to be valid.)

In the following example, a record is declared using %ROWTYPE as a data type specifier:

```
DECLARE
 emp_record  employees%ROWTYPE;
 ...
```

%ROWTYPE Attribute (continued)

The emp_record record has a structure consisting of the following fields, each representing a column in the employees table.

Note: This is not code, but simply the structure of the composite variable.

```
(employee_id        NUMBER(6),
first_name          VARCHAR2(20),
last_name           VARCHAR2(20),
email               VARCHAR2(20),
phone_number        VARCHAR2(20),
hire_date           DATE,
salary              NUMBER(8,2),
commission_pct      NUMBER(2,2),
manager_id          NUMBER(6),
department_id       NUMBER(4))
```

To reference an individual field, use the dot notation:

```
record_name.field_name
```

For example, you reference the commission_pct field in the emp_record record as follows:

```
emp_record.commission_pct
```

You can then assign a value to the record field:

```
emp_record.commission_pct:= .35;
```

Assigning Values to Records

You can assign a list of common values to a record by using the SELECT or FETCH statement. Make sure that the column names appear in the same order as the fields in your record. You can also assign one record to another if both have the same corresponding data types. A record of type employees%ROWTYPE and a user-defined record type having analogous fields of the employees table will have the same data type. Therefore, if a user-defined record contains fields similar to the fields of a %ROWTYPE record, you can assign that user-defined record to the %ROWTYPE record.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating a PL/SQL Record: Example

```
DECLARE
  TYPE t_rec IS RECORD
    (v_sal number(8),
     v_minsal number(8) default 1000,
     v_hire_date employees.hire_date%type,
     v_rec1 employees%rowtype);
  v_myrec t_rec;
BEGIN
  v_myrec.v_sal := v_myrec.v_minsal + 500;
  v_myrec.v_hire_date := sysdate;
  SELECT * INTO v_myrec.v_rec1
      FROM employees WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE(v_myrec.v_rec1.last_name ||' '||
  to_char(v_myrec.v_hire_date) ||' '|| to_char(v_myrec.v_sal));
END;
```

```
anonymous block completed
King 16-FEB-09 1500
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating a PL/SQL Record: Example

The field declarations used in defining a record are like variable declarations. Each field has a unique name and a specific data type. There are no predefined data types for PL/SQL records, as there are for scalar variables. Therefore, you must create the record type first, and then declare an identifier using that type.

In the example in the slide, a PL/SQL record is created using the required two-step process:

1. A record type (`t_rec`) is defined
2. A record (`v_myrec`) of the `t_rec` type is declared

Note

- The record contains four fields: `v_sal`, `v_minsal`, `v_hire_date`, and `v_rec1`.
- `v_rec1` is defined using the `%ROWTYPE` attribute, which is similar to the `%TYPE` attribute. With `%TYPE`, a field inherits the data type of a specified column. With `%ROWTYPE`, a field inherits the column names and data types of all columns in the referenced table.
- PL/SQL record fields are referenced using the `<record>.<field>` notation, or the `<record>.<field>.<column>` notation for fields that are defined with the `%ROWTYPE` attribute.
- You can add the `NOT NULL` constraint to any field declaration to prevent assigning nulls to that field. Remember that fields that are declared as `NOT NULL` must be initialized.

Advantages of Using the %ROWTYPE Attribute

- The number and data types of the underlying database columns need not be known—and, in fact, might change at run time.
- The %ROWTYPE attribute is useful when you want to retrieve a row with:
  - The SELECT * statement
  - Row-level INSERT and UPDATE statements

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Advantages of Using %ROWTYPE

The advantages of using the %ROWTYPE attribute are listed in the slide. Use the %ROWTYPE attribute when you are not sure about the structure of the underlying database table.

The main advantage of using %ROWTYPE is that it simplifies maintenance. Using %ROWTYPE ensures that the data types of the variables declared with this attribute change dynamically when the underlying table is altered. If a DDL statement changes the columns in a table, the PL/SQL program unit is invalidated. When the program is recompiled, it automatically reflects the new table format.

The %ROWTYPE attribute is particularly useful when you want to retrieve an entire row from a table. In the absence of this attribute, you would be forced to declare a variable for each of the columns retrieved by the SELECT statement.

Another %ROWTYPE Attribute Example

```
DECLARE
  v_employee_number number:= 124;
  v_emp_rec   employees%ROWTYPE;
BEGIN
 SELECT * INTO v_emp_rec FROM employees
 WHERE  employee_id = v_employee_number;
 INSERT INTO retired_emps(empno, ename, job, mgr,
                 hiredate, leavedate, sal, comm, deptno)
   VALUES (v_emp_rec.employee_id, v_emp_rec.last_name,
           v_emp_rec.job_id, v_emp_rec.manager_id,
           v_emp_rec.hire_date, SYSDATE,
           v_emp_rec.salary,  v_emp_rec.commission_pct,
           v_emp_rec.department_id);
END;
/
```

```
SELECT * FROM retired_emps;
/
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-JUN-09	5800	(null)	50

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Another %ROWTYPE Attribute Example

Another example of the %ROWTYPE attribute is shown in the slide. If an employee is retiring, information about that employee is added to a table that holds information about retired employees. The user supplies the employee number. The record of the employee specified by the user is retrieved from the employees table and stored in the emp_rec variable, which is declared using the %ROWTYPE attribute.

The CREATE statement that creates the retired_emps table is:

```
CREATE TABLE retired_emps
   (EMPNO      NUMBER(4), ENAME      VARCHAR2(10),
    JOB        VARCHAR2(9),MGR        NUMBER(4),
    HIREDATE   DATE, LEAVEDATE  DATE,
    SAL        NUMBER(7,2), COMM       NUMBER(7,2),
    DEPTNO     NUMBER(2))
```

Note

- The record that is inserted into the retired_emps table is shown in the slide.
- To see the output shown in the slide, place your cursor on the SELECT statement at the bottom of the code example in SQL Developer and press F9.
- The complete code example is found in code_6_14_n-s.sql.

Inserting a Record by Using %ROWTYPE

```
...
DECLARE
  v_employee_number number:= 124;
  v_emp_rec retired_emps%ROWTYPE;
BEGIN
 SELECT employee_id, last_name, job_id, manager_id,
 hire_date, hire_date, salary, commission_pct,
 department_id INTO v_emp_rec FROM employees
 WHERE  employee_id = v_employee_number;
 INSERT INTO retired_emps VALUES v_emp_rec;
END;
/
SELECT * FROM retired_emps;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-NOV-99	5800	(null)	50

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Inserting a Record by Using %ROWTYPE

Compare the INSERT statement in the previous slide with the INSERT statement in this slide. The emp_rec record is of type retired_emps. The number of fields in the record must be equal to the number of field names in the INTO clause. You can use this record to insert values into a table. This makes the code more readable.

Examine the SELECT statement in the slide. You select hire_date twice and insert the hire_date value in the leavedate field of retired_emps. No employee retires on the hire date. The inserted record is shown in the slide. (You will see how to update this in the next slide.)

Note: To see the output shown in the slide, place your cursor on the SELECT statement at the bottom of the code example in SQL Developer and press F9.

Updating a Row in a Table by Using a Record

```
SET VERIFY OFF
DECLARE
  v_employee_number number:= 124;
  v_emp_rec retired_emps%ROWTYPE;
BEGIN
 SELECT * INTO v_emp_rec FROM retired_emps;
 v_emp_rec.leavedate:=CURRENT_DATE;
 UPDATE retired_emps SET ROW = v_emp_rec WHERE
  empno=v_employee_number;
END;
/
SELECT * FROM retired_emps;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	EMPNO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-99	16-NOV-99	5800	(null)	50

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Updating a Row in a Table by Using a Record

You learned to insert a row by using a record. This slide shows you how to update a row by using a record.

- The `ROW` keyword is used to represent the entire row.
- The code shown in the slide updates the `leavedate` of the employee.
- The record is updated as shown in the slide.

Note: To see the output shown in the slide, place your cursor on the `SELECT` statement at the bottom of the code example in SQL Developer and press F9.

Agenda

- Examining composite data types
- Using PL/SQL records
  - Manipulating data with PL/SQL records
  - Advantages of the `%ROWTYPE` attribute
- Using PL/SQL collections
  - Examining associative arrays
  - Introducing nested tables
  - Introducing `VARRAY`

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Agenda

As stated previously, PL/SQL collections are used when you want to store values of the same data type. This data type can also be of the composite type (such as records).

Therefore, collections are used to treat data as a single unit. Collections are of three types:

- Associative array
- Nested table
- `VARRAY`

Note: Of these three collections, the associative array is the focus of this lesson. The Nested table and `VARRARY` are introduced only for comparative purposes. These two collections are covered in detail in the course *Oracle Database 11g: Advanced PL/SQL*.

Associative Arrays (`INDEX BY` Tables)

An associative array is a PL/SQL collection with two columns:

- Primary key of integer or string data type
- Column of scalar or record data type

Key	Values
1	JONES
2	HARDEY
3	MADURO
4	KRAMER

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

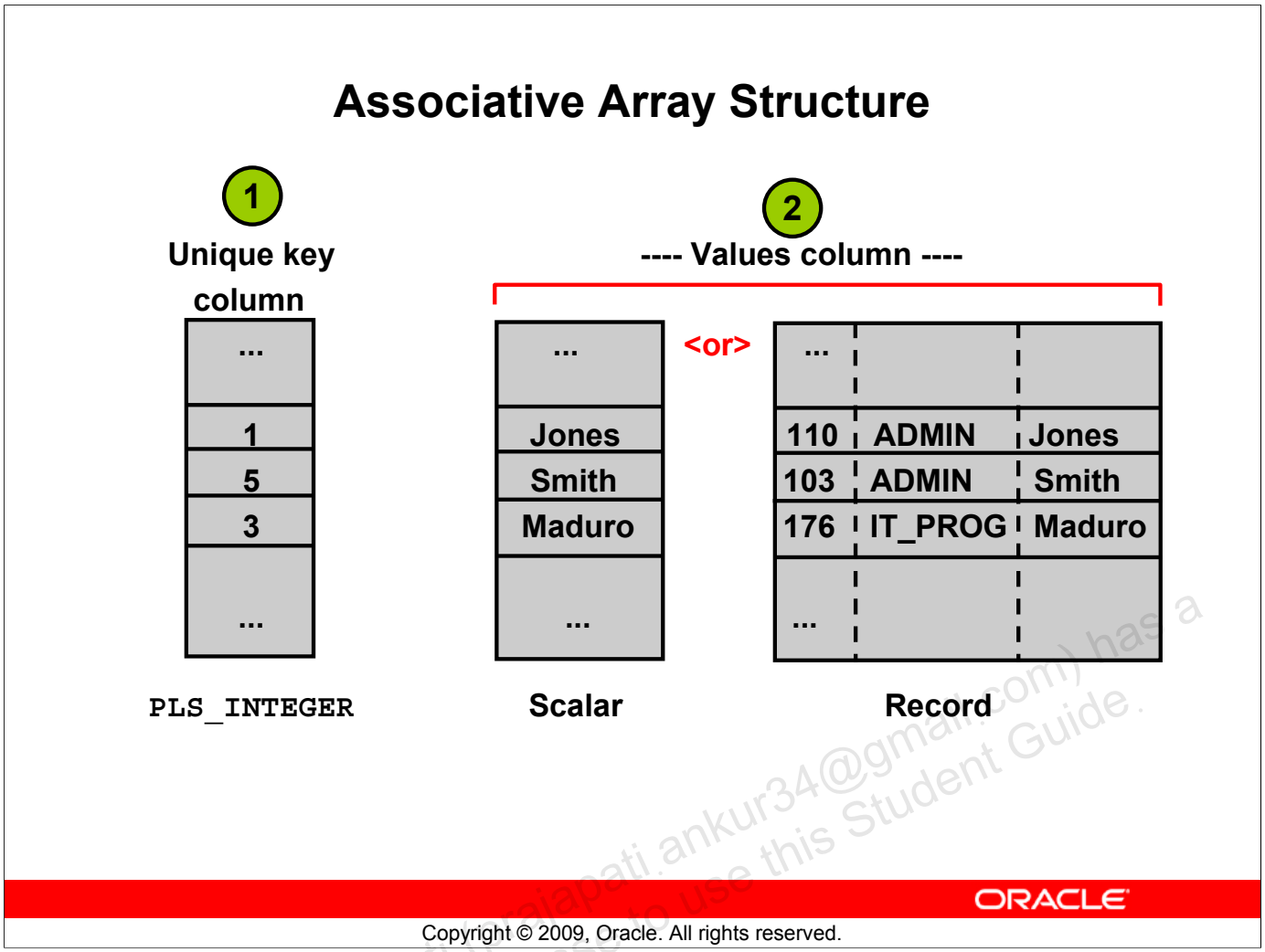
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Associative Arrays (`INDEX BY` Tables)

An associative array is a type of PL/SQL collection. It is a composite data type, and is user defined. Associative arrays are sets of key-value pairs. They can store data using a primary key value as the index, where the key values are not necessarily sequential. Associative arrays are also known as *`INDEX BY`* tables.

Associative arrays have only two columns, neither of which can be named:

- The first column, of integer or string type, acts as the primary key.
- The second column, of scalar or record data type, holds values.



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Associative Array Structure

As previously mentioned, associative arrays have two columns. The second column either holds one value per row, or multiple values.

Unique Key Column: The data type of the key column can be:

- Numeric, either `BINARY_INTEGER` or `PLS_INTEGER`. These two numeric data types require less storage than `NUMBER`, and arithmetic operations on these data types are faster than the `NUMBER` arithmetic.
- `VARCHAR2` or one of its subtypes

"Value" Column: The value column can be either a scalar data type or a record data type. A column with scalar data type can hold only one value per row, whereas a column with record data type can hold multiple values per row.

Other Characteristics

- An associative array is not populated at the time of declaration. It contains no keys or values, and you cannot initialize an associative array in its declaration.
- An explicit executable statement is required to populate the associative array.
- Like the size of a database table, the size of an associative array is unconstrained. That is, the number of rows can increase dynamically so that your associative array grows as new rows are added. Note that the keys do not have to be sequential, and can be both positive and negative.

Steps to Create an Associative Array

Syntax:

```
1  TYPE type_name IS TABLE OF
        {column_type | variable%TYPE
        | table.column%TYPE} [NOT NULL]
        | table%ROWTYPE
        | INDEX BY PLS_INTEGER | BINARY_INTEGER
         | VARCHAR2(<size>);
2  identifier type_name;
```

Example:

```
...
TYPE ename_table_type IS TABLE OF
 employees.last_name%TYPE
 INDEX BY PLS_INTEGER;
 ...
ename_table ename_table_type;
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Arun M. Prajapati (prajapati.arun34@gmail.com) has a non-transferable license to use this Student Guide.

Steps to Create an Associative Array

There are two steps involved in creating an associative array:

1. Declare a `TABLE` data type using the `INDEX BY` option.
2. Declare a variable of that data type.

Syntax

type_name	Is the name of the `TABLE` type (This name is used in the subsequent declaration of the array identifier.)
column_type	Is any scalar or composite data type such as `VARCHAR2`, `DATE`, `NUMBER`, or `%TYPE` (You can use the `%TYPE` attribute to provide the column data type.)
identifier	Is the name of the identifier that represents an entire associative array

Note: The `NOT NULL` constraint prevents nulls from being assigned to the associative array.

Example

In the example, an associative array with the variable name `ename_table` is declared to store the last names of employees.

Creating and Accessing Associative Arrays

```
...
DECLARE
  TYPE ename_table_type IS TABLE OF
    employees.last_name%TYPE
    INDEX BY PLS_INTEGER;
  TYPE hiredate_table_type IS TABLE OF DATE
    INDEX BY PLS_INTEGER;
  ename_table       ename_table_type;
  hiredate_table    hiredate_table_type;
BEGIN
  ename_table(1)     := 'CAMERON';
  hiredate_table(8)  := SYSDATE + 7;
    IF ename_table.EXISTS(1) THEN
    INSERT INTO ...
    ...
END;
/
...
```

```
anonymous block completed
ENAME                     HIREDT
------------------------- -------------------
CAMERON                   23-JUN-09

1 rows selected
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating and Accessing Associative Arrays

The example in the slide creates two associative arrays, with the identifiers `ename_table` and `hiredate_table`.

The key of each associative array is used to access an element in the array, by using the following syntax:

```
identifier(index)
```

In both arrays, the `index` value belongs to the `PLS_INTEGER` type.

- To reference the first row in the `ename_table` associative array, specify: `ename_table(1)`
- To reference the eighth row in the `hiredate_table` associative array, specify: `hiredate_table(8)`

Note

- The magnitude range of a `PLS_INTEGER` is –2,147,483,647 through 2,147,483,647, so the primary key value can be negative. Indexing does not need to start with 1.
- The `exists(i)` method returns `TRUE` if a row with index *i* is returned. Use the `exists` method to prevent an error that is raised in reference to a nonexistent table element.
- The complete code example is found in `code_6_21_s.sql`.

Using INDEX BY Table Methods

The following methods make associative arrays easier to use:

- EXISTS
- COUNT
- FIRST
- LAST
- PRIOR
- NEXT
- DELETE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Using INDEX BY Table Methods

An INDEX BY table method is a built-in procedure or function that operates on an associative array and is called by using the dot notation.

Syntax: *table_name.method_name*[(*parameters*)]

Method	Description
EXISTS(*n*)	Returns TRUE if the *n*th element in an associative array exists
COUNT	Returns the number of elements that an associative array currently contains
FIRST	• Returns the first (smallest) index number in an associative array • Returns NULL if the associative array is empty
LAST	• Returns the last (largest) index number in an associative array • Returns NULL if the associative array is empty
PRIOR(*n*)	Returns the index number that precedes index *n* in an associative array
NEXT(*n*)	Returns the index number that succeeds index *n* in an associative array
DELETE	• DELETE removes all elements from an associative array. • DELETE(n) removes the *n*th element from an associative array. • DELETE(m, n) removes all elements in the range *m* ... *n* from an associative array.

INDEX BY Table of Records Option

Define an associative array to hold an entire row from a table.

```
DECLARE
  TYPE dept_table_type IS TABLE OF
       departments%ROWTYPE INDEX PLS_INTEGER;
  dept_table dept_table_type;
  -- Each element of dept_table is a record
Begin
  SELECT * INTO dept_table(1) FROM departments
    WHERE department_id = 10;
  DBMS_OUTPUT.PUT_LINE(dept_table(1).department_id || ||
   dept_table(1).department_name || ||
   dept_table(1).manager_id);
END;
/
```

Results | Script Output | Expl

```
anonymous block completed
10 Administration 200
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Anko M. Rajapati (rajapati.anko@gmail.com) has a non-transferable license to use this Student Guide.

INDEX BY Table of Records Option

As previously discussed, an associative array that is declared as a table of scalar data type can store the details of only one column in a database table. However, there is often a need to store all the columns retrieved by a query. The INDEX BY table of records option enables one array definition to hold information about all the fields of a database table.

Creating and Referencing a Table of Records

As shown in the associative array example in the slide, you can:

- Use the %ROWTYPE attribute to declare a record that represents a row in a database table
- Refer to fields within the dept_table array because each element of the array is a record

The differences between the %ROWTYPE attribute and the composite data type PL/SQL record are as follows:

- PL/SQL record types can be user-defined, whereas %ROWTYPE implicitly defines the record.
- PL/SQL records enable you to specify the fields and their data types while declaring them. When you use %ROWTYPE, you cannot specify the fields. The %ROWTYPE attribute represents a table row with all the fields based on the definition of that table.
- User-defined records are static, but %ROWTYPE records are dynamic—they are based on a table structure. If the table structure changes, the record structure also picks up the change.

INDEX BY Table of Records Option: Example 2

```
DECLARE
   TYPE emp_table_type IS TABLE OF
      employees%ROWTYPE INDEX BY PLS_INTEGER;
   my_emp_table  emp_table_type;
   max_count         NUMBER(3):= 104;
BEGIN
  FOR i IN 100..max_count
  LOOP
   SELECT * INTO my_emp_table(i) FROM employees
   WHERE employee_id = i;
  END LOOP;
  FOR i IN my_emp_table.FIRST..my_emp_table.LAST
  LOOP
     DBMS_OUTPUT.PUT_LINE(my_emp_table(i).last_name);
  END LOOP;
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

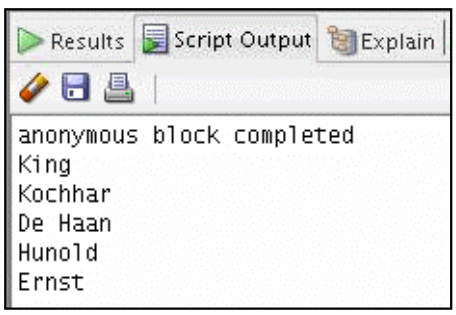
INDEX BY Table of Records: Example 2

The example in the slide declares an associative array, using the `INDEX BY` table of records option, to temporarily store the details of employees whose employee IDs are between 100 and 104. The variable name for the array is `emp_table_type`.

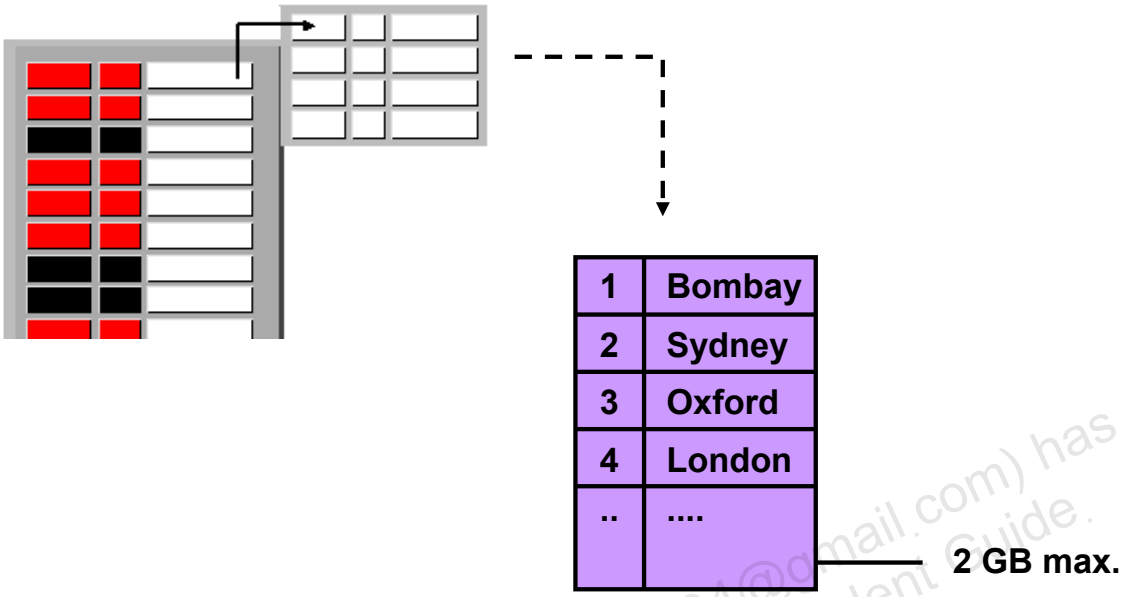
Using a loop, the information of the employees from the `EMPLOYEES` table is retrieved and stored in the array. Another loop is used to print the last names from the array. Note the use of the `first` and `last` methods in the example.

Note: The slide demonstrates one way to work with an associative array that uses the `INDEX BY` table of records method. However, you can do the same more efficiently using cursors. Cursors are explained in the lesson titled "Using Explicit Cursors."

The results of the code example is as follows:



Nested Tables



Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Nested Tables

The functionality of nested tables is similar to that of associative arrays; however, there are differences in the nested table implementation.

- The nested table is a valid data type in a schema-level table, but an associative array is not. Therefore, unlike associative arrays, nested tables can be stored in the database.
- The size of a nested table can increase dynamically, although the maximum size is 2 GB.
- The "key" cannot be a negative value (unlike in the associative array). Though reference is made to the first column as key, there is no key in a nested table. There is a column with numbers.
- Elements can be deleted from anywhere in a nested table, leaving a sparse table with nonsequential "keys." The rows of a nested table are not in any particular order.
- When you retrieve values from a nested table, the rows are given consecutive subscripts starting from 1.

Syntax

```
TYPE type_name IS TABLE OF
 {column_type | variable%TYPE
 | table.column%TYPE} [NOT NULL]
 | table.%ROWTYPE
```

Nested Tables (continued)

Example:

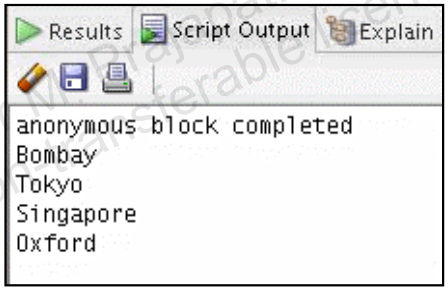
```
TYPE location_type IS TABLE OF locations.city%TYPE;
offices location_type;
```

If you do not initialize a nested table, it is automatically initialized to `NULL`. You can initialize the `offices` nested table by using a constructor:

```
offices := location_type('Bombay', 'Tokyo','Singapore',
'Oxford');
```

The complete code example and output is as follows:

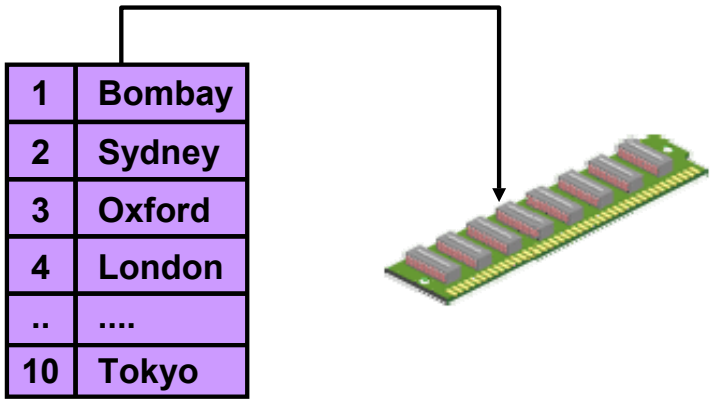
```
SET SERVEROUTPUT ON;

DECLARE
   TYPE location_type IS TABLE OF locations.city%TYPE;
    offices location_type;
    table_count NUMBER;
BEGIN
   offices := location_type('Bombay', 'Tokyo','Singapore',
    'Oxford');
FOR i in 1.. offices.count() LOOP
     DBMS_OUTPUT.PUT_LINE(offices(i));
   END LOOP;
END;
/
```



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide

VARRAY



Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

VARRAY

A variable-size array (VARRAY) is similar to an associative array, except that a VARRAY is constrained in size.

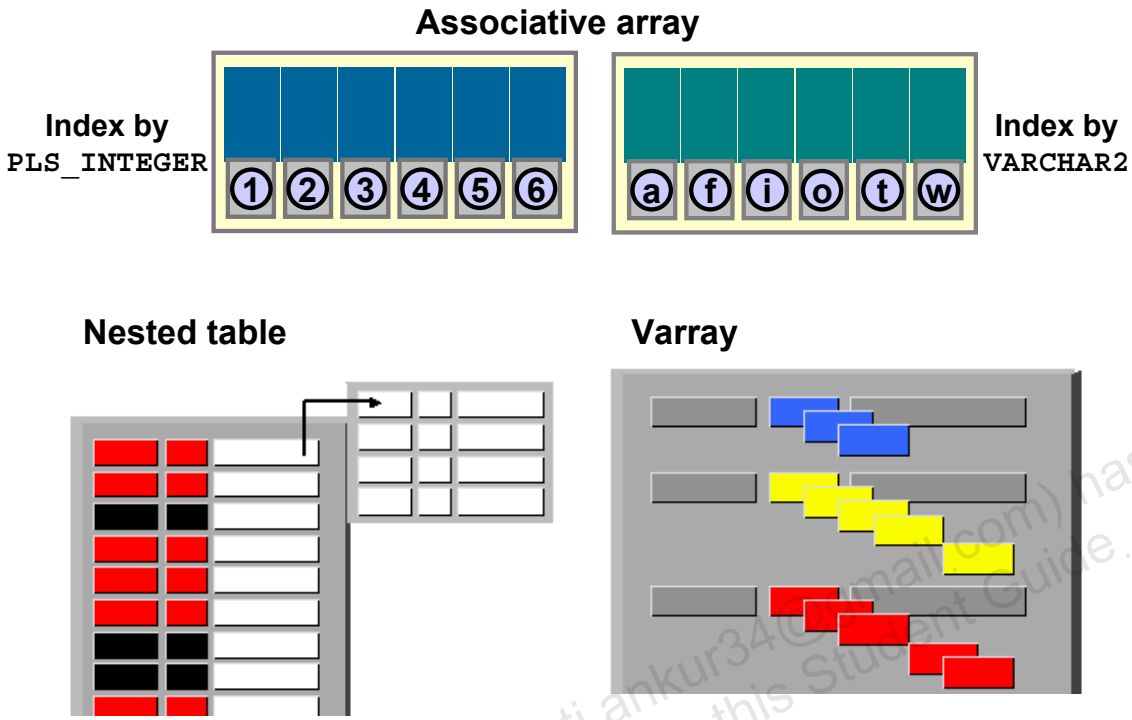
- A VARRAY is valid in a schema-level table.
- Items of VARRAY type are called VARRAYs.
- VARRAYs have a fixed upper bound. You have to specify the upper bound when you declare them. This is similar to arrays in C language. The maximum size of a VARRAY is 2 GB, as in nested tables.
- The distinction between a nested table and a VARRAY is the physical storage mode. The elements of a VARRAY are stored inline with the table's data unless the size of the VARRAY is greater than 4 KB. Contrast that with nested tables, which are always stored out-of-line.
- You can create a VARRAY type in the database by using SQL.

Example:

```
TYPE location_type IS VARRAY(3) OF locations.city%TYPE;
offices location_type;
```

The size of this VARRAY is restricted to 3. You can initialize a VARRAY by using constructors. If you try to initialize the VARRAY with more than three elements, a "Subscript outside of limit" error message is displayed.

Summary of Collection Types



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Summary of Collection Types

Associative Arrays

Associative arrays are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be either integer- or character-based. The array value may be of the scalar data type (single value) or the record data type (multiple values).

Because associative arrays are intended for storing temporary data, you cannot use them with SQL statements such as `INSERT` and `SELECT INTO`.

Nested Tables

A nested table holds a set of values. In other words, it is a table within a table. Nested tables are unbounded; that is, the size of the table can increase dynamically. Nested tables are available in both PL/SQL and the database. Within PL/SQL, nested tables are like one-dimensional arrays whose size can increase dynamically.

Varrays

Variable-size arrays, or varrays, are also collections of homogeneous elements that hold a fixed number of elements (although you can change the number of elements at run time). They use sequential numbers as subscripts. You can define equivalent SQL types, thereby allowing varrays to be stored in database tables.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Quiz

Identify situations in which you can use the `%ROWTYPE` attribute.

1. When you are not sure about the structure of the underlying database table
2. When you want to retrieve an entire row from a table
3. When you want to declare a variable according to another previously declared variable or database column

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Answer: 1, 2

Advantages of Using the `%ROWTYPE` Attribute

Use the `%ROWTYPE` attribute when you are not sure about the structure of the underlying database table.

The main advantage of using `%ROWTYPE` is that it simplifies maintenance. Using `%ROWTYPE` ensures that the data types of the variables declared with this attribute change dynamically when the underlying table is altered. If a DDL statement changes the columns in a table, the PL/SQL program unit is invalidated. When the program is recompiled, it automatically reflects the new table format.

The `%ROWTYPE` attribute is particularly useful when you want to retrieve an entire row from a table. In the absence of this attribute, you would be forced to declare a variable for each of the columns retrieved by the `SELECT` statement.

Summary

In this lesson, you should have learned to:

- Define and reference PL/SQL variables of composite data types
  - PL/SQL record
  - Associative array
    - `INDEX BY` table
    - `INDEX BY` table of records
- Define a PL/SQL record by using the `%ROWTYPE` attribute
- Compare and contrast the three PL/SQL collection types:
  - Associative array
  - Nested table
  - `VARRAY`

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Summary

A PL/SQL record is a collection of individual fields that represent a row in a table. By using records, you can group the data into one structure, and then manipulate this structure as one entity or logical unit. This helps reduce coding and keeps the code easy to maintain and understand.

Like PL/SQL records, a PL/SQL collection is another composite data type. PL/SQL collections include:

- Associative arrays (also known as `INDEX BY` tables). They are objects of `TABLE` type and look similar to database tables, but with a slight difference. The so-called `INDEX BY` tables use a primary key to give you array-like access to rows. The size of an associative array is unconstrained.
- Nested tables. The key for nested tables cannot have a negative value, unlike `INDEX BY` tables. The key must also be in a sequence.
- Variable-size arrays (`VARRAY`). A `VARRAY` is similar to associative arrays, except that a `VARRAY` is constrained in size.

Practice 6: Overview

This practice covers the following topics:

- Declaring associative arrays
- Processing data by using associative arrays
- Declaring a PL/SQL record
- Processing data by using a PL/SQL record

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Practice 6: Overview

In this practice, you define, create, and use associative arrays and PL/SQL records.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

7 Using Explicit Cursors

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

After completing this lesson, you should be able to do the following:

- Distinguish between implicit and explicit cursors
- Discuss the reasons for using explicit cursors
- Declare and control explicit cursors
- Use simple loops and cursor `FOR` loops to fetch data
- Declare and use cursors with parameters
- Lock rows with the `FOR UPDATE` clause
- Reference the current row with the `WHERE CURRENT OF` clause

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

You have learned about implicit cursors that are automatically created by PL/SQL when you execute a SQL `SELECT` or DML statement. In this lesson, you learn about explicit cursors. You learn to differentiate between implicit and explicit cursors. You also learn to declare and control simple cursors, as well as cursors with parameters.

Agenda

- What are explicit cursors?
- Using explicit cursors
- Using cursors with parameters
- Locking rows and referencing the current row

ORACLE

Copyright © 2009, Oracle. All rights reserved.

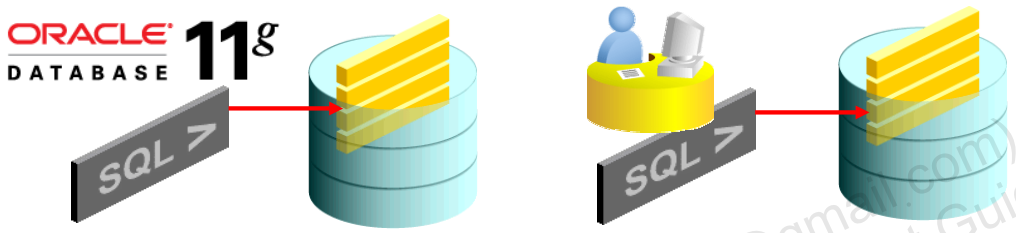
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Cursors

Every SQL statement that is executed by the Oracle Server has an associated individual cursor:

- Implicit cursors: declared and managed by PL/SQL for all DML and PL/SQL `SELECT` statements
- Explicit cursors: declared and managed by the programmer



Implicit cursor

Explicit cursor

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur3@gmail.com) has a non-transferable license to use this Student Guide.

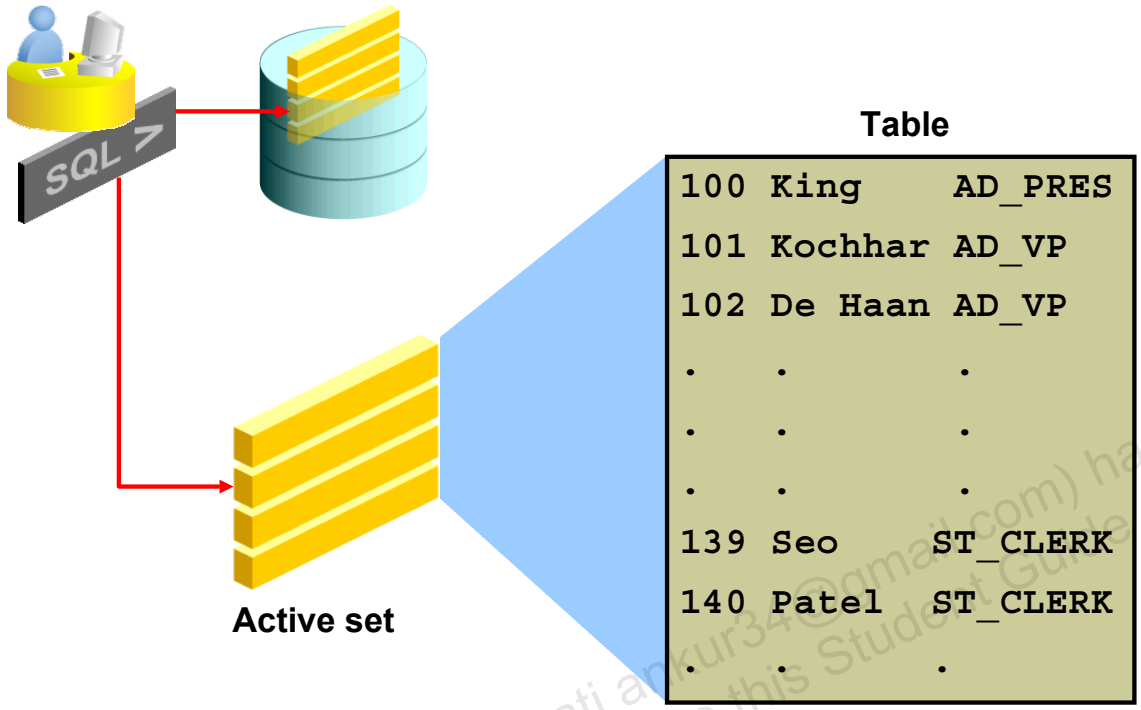
Cursors

The Oracle Server uses work areas (called *private SQL areas*) to execute SQL statements and to store processing information. You can use explicit cursors to name a private SQL area and to access its stored information.

Cursor Type	Description
Implicit	Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL `SELECT` statements.
Explicit	For queries that return multiple rows, explicit cursors are declared and managed by the programmer, and manipulated through specific statements in the block's executable actions.

The Oracle Server implicitly opens a cursor to process each SQL statement that is not associated with an explicitly declared cursor. Using PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor.

Explicit Cursor Operations



Copyright © 2009, Oracle. All rights reserved.

Explicit Cursor Operations

You declare explicit cursors in PL/SQL when you have a `SELECT` statement that returns multiple rows. You can process each row returned by the `SELECT` statement.

The set of rows returned by a multiple-row query is called the *active set*. Its size is the number of rows that meet your search criteria. The diagram in the slide shows how an explicit cursor "points" to the current row in the active set. This enables your program to process the rows one at a time.

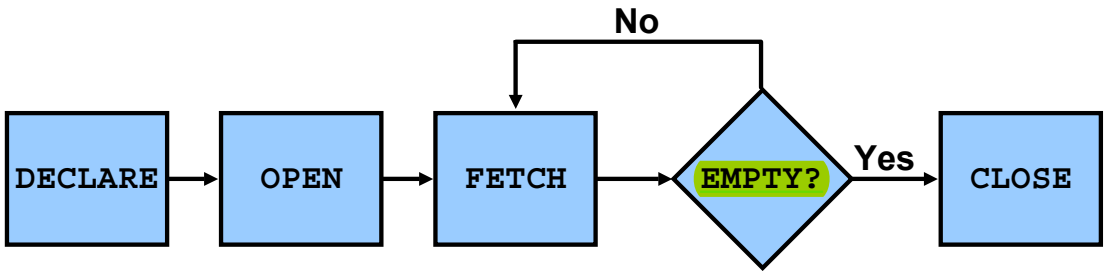
Explicit cursor functions:

- Can perform row-by-row processing beyond the first row returned by a query
- Keep track of the row that is currently being processed
- Enable the programmer to manually control explicit cursors in the PL/SQL block

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Controlling Explicit Cursors



- Create a named SQL area.
- Identify the active set.
- Load the current row into variables.
- Test for existing rows.
- Return to FETCH if rows are found.
- Release the active set.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur3@gmail.com) has a non-transferable license to use this Student Guide.

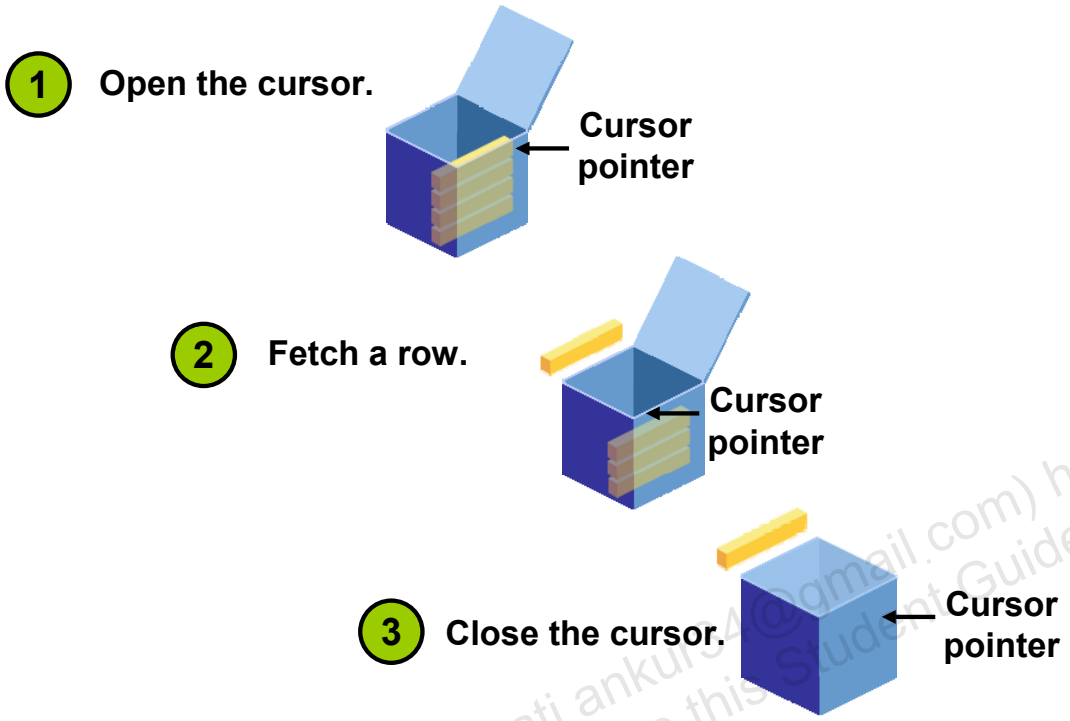
Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Controlling Explicit Cursors

Now that you have a conceptual understanding of cursors, review the steps to use them.

1. In the declarative section of a PL/SQL block, declare the cursor by naming it and defining the structure of the query to be associated with it.
2. Open the cursor.
   The `OPEN` statement executes the query and binds any variables that are referenced. Rows identified by the query are called the *active set* and are now available for fetching.
3. Fetch data from the cursor.
   In the flow diagram shown in the slide, after each fetch, you test the cursor for any existing row. If there are no more rows to process, you must close the cursor.
4. Close the cursor.
   The `CLOSE` statement releases the active set of rows. It is now possible to reopen the cursor to establish a fresh active set.

Controlling Explicit Cursors



Copyright © 2009, Oracle. All rights reserved.

Controlling Explicit Cursors (continued)

A PL/SQL program opens a cursor, processes rows returned by a query, and then closes the cursor. The cursor marks the current position in the active set.

1. The `OPEN` statement executes the query associated with the cursor, identifies the active set, and positions the cursor at the first row.
2. The `FETCH` statement retrieves the current row and advances the cursor to the next row until there are no more rows or a specified condition is met.
3. The `CLOSE` statement releases the cursor.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Agenda

- What are explicit cursors?
- Using explicit cursors
- Using cursors with parameters
- Locking rows and referencing the current row

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Declaring the Cursor

Syntax:

```
CURSOR cursor_name IS
     select_statement;
```

Examples:

```
DECLARE
  CURSOR c_emp_cursor IS
  SELECT employee_id, last_name FROM employees
  WHERE department_id =30;
```

```
DECLARE
  v_locid NUMBER:= 1700;
  CURSOR c_dept_cursor IS
  SELECT * FROM departments
  WHERE location_id = v_locid;
...
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Declaring the Cursor

The syntax to declare a cursor is shown in the slide. In the syntax:

cursor_name Is a PL/SQL identifier
select_statement Is a `SELECT` statement without an `INTO` clause

The active set of a cursor is determined by the `SELECT` statement in the cursor declaration. It is mandatory to have an `INTO` clause for a `SELECT` statement in PL/SQL. However, note that the `SELECT` statement in the cursor declaration cannot have an `INTO` clause. That is because you are only defining a cursor in the declarative section and not retrieving any rows into the cursor.

Note

- Do not include the `INTO` clause in the cursor declaration because it appears later in the `FETCH` statement.
- If you want the rows to be processed in a specific sequence, use the `ORDER BY` clause in the query.
- The cursor can be any valid `SELECT` statement, including joins, subqueries, and so on.

Declaring the Cursor (continued)

The `c_emp_cursor` cursor is declared to retrieve the `employee_id` and `last_name` columns for those employees working in the department with `department_id` 30.

The `c_dept_cursor` cursor is declared to retrieve all the details for the department with the `location_id` 1700. Note that a variable is used while declaring the cursor. These variables are considered bind variables, which must be visible when you are declaring the cursor. These variables are examined only once at the time the cursor opens. You have learned that explicit cursors are used when you have to retrieve and operate on multiple rows in PL/SQL. However, this example shows that you can use the explicit cursor even if your `SELECT` statement returns only one row.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Opening the Cursor

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
...
BEGIN
  OPEN c_emp_cursor;
```

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Opening the Cursor

The `OPEN` statement executes the query associated with the cursor, identifies the active set, and positions the cursor pointer at the first row. The `OPEN` statement is included in the executable section of the PL/SQL block.

`OPEN` is an executable statement that performs the following operations:

1. Dynamically allocates memory for a context area
2. Parses the `SELECT` statement
3. Binds the input variables (sets the values for the input variables by obtaining their memory addresses)
4. Identifies the active set (the set of rows that satisfy the search criteria). Rows in the active set are not retrieved into variables when the `OPEN` statement is executed. Rather, the `FETCH` statement retrieves the rows from the cursor to the variables.
5. Positions the pointer to the first row in the active set

Note: If a query returns no rows when the cursor is opened, PL/SQL does not raise an exception. You can find out the number of rows returned with an explicit cursor by using the `<cursor_name>%ROWCOUNT` attribute.

Fetching Data from the Cursor

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
  v_empno employees.employee_id%TYPE;
  v_lname employees.last_name%TYPE;
BEGIN
  OPEN c_emp_cursor;
  FETCH c_emp_cursor INTO v_empno, v_lname;
  DBMS_OUTPUT.PUT_LINE( v_empno ||'  '||v_lname);
END;
/
```

```
anonymous block completed
114  Raphaely
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Fetching Data from the Cursor

The `FETCH` statement retrieves the rows from the cursor one at a time. After each fetch, the cursor advances to the next row in the active set. You can use the `%NOTFOUND` attribute to determine whether the entire active set has been retrieved.

Consider the example shown in the slide. Two variables, `empno` and `lname`, are declared to hold the fetched values from the cursor. Examine the `FETCH` statement.

You have successfully fetched the values from the cursor to the variables. However, there are six employees in department 30, but only one row was fetched. To fetch all rows, you must use loops. In the next slide, you see how a loop is used to fetch all the rows.

The `FETCH` statement performs the following operations:

1. Reads the data for the current row into the output PL/SQL variables
2. Advances the pointer to the next row in the active set

Fetching Data from the Cursor (continued)

You can include the same number of variables in the `INTO` clause of the `FETCH` statement as there are columns in the `SELECT` statement; be sure that the data types are compatible. Match each variable to correspond to the columns positionally. Alternatively, you can also define a record for the cursor and reference the record in the `FETCH INTO` clause. Finally, test to see whether the cursor contains rows. If a fetch acquires no values, there are no rows left to process in the active set and no error is recorded.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Fetching Data from the Cursor

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
  v_empno employees.employee_id%TYPE;
  v_lname employees.last_name%TYPE;
BEGIN
  OPEN c_emp_cursor;
  LOOP
    FETCH c_emp_cursor INTO v_empno, v_lname;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_empno ||'  '||v_lname);
  END LOOP;
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Fetching Data from the Cursor (continued)

Observe that a simple `LOOP` is used to fetch all the rows. Also, the cursor attribute `%NOTFOUND` is used to test for the exit condition. The output of the PL/SQL block is:

```
anonymous block completed
114  Raphaely
115  Khoo
116  Baida
117  Tobias
118  Himuro
119  Colmenares
```

Closing the Cursor

```
...
  LOOP
    FETCH c_emp_cursor INTO empno, lname;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_empno ||' '||v_lname);
  END LOOP;
 CLOSE c_emp_cursor;
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Closing the Cursor

The `CLOSE` statement disables the cursor, releases the context area, and "undefines" the active set. Close the cursor after completing the processing of the `FETCH` statement. You can reopen the cursor if required. A cursor can be reopened only if it is closed. If you attempt to fetch data from a cursor after it is closed, an `INVALID_CURSOR` exception is raised.

Note: Although it is possible to terminate the PL/SQL block without closing cursors, you should make it a habit to close any cursor that you declare explicitly to free resources.
There is a maximum limit on the number of open cursors per session, which is determined by the `OPEN_CURSORS` parameter in the database parameter file. (`OPEN_CURSORS` = `50` by default.)

Cursors and Records

Process the rows of the active set by fetching values into a PL/SQL record.

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE  department_id =30;
   v_emp_record   c_emp_cursor%ROWTYPE;
BEGIN
  OPEN c_emp_cursor;
  LOOP
    FETCH c_emp_cursor INTO v_emp_record;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( v_emp_record.employee_id
                          ||' '||v_emp_record.last_name);
  END LOOP;
  CLOSE c_emp_cursor;
END;
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Cursors and Records

You have already seen that you can define records that have the structure of columns in a table. You can also define a record based on the selected list of columns in an explicit cursor. This is convenient for processing the rows of the active set, because you can simply fetch into the record. Therefore, the values of the rows are loaded directly into the corresponding fields of the record.

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

Cursor FOR Loops

Syntax:

```
FOR record_name IN cursor_name LOOP
  statement1;
  statement2;
  . . .
END LOOP;
```

- The cursor FOR loop is a shortcut to process explicit cursors.
- Implicit open, fetch, exit, and close occur.
- The record is implicitly declared.

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (rajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Cursor FOR Loops

You learned to fetch data from cursors by using simple loops. You now learn to use a cursor FOR loop, which processes rows in an explicit cursor. It is a shortcut because the cursor is opened, a row is fetched once for each iteration in the loop, the loop exits when the last row is processed, and the cursor is closed automatically. The loop itself is terminated automatically at the end of the iteration where the last row is fetched.

In the syntax:

record_name	Is the name of the implicitly declared record
cursor_name	Is a PL/SQL identifier for the previously declared cursor

Guidelines

- Do not declare the record that controls the loop; it is declared implicitly.
- Test the cursor attributes during the loop if required.
- Supply the parameters for a cursor, if required, in parentheses following the cursor name in the FOR statement.

Cursor FOR Loops

```
DECLARE
  CURSOR c_emp_cursor IS
   SELECT employee_id, last_name FROM employees
   WHERE department_id =30;
BEGIN
   FOR emp_record IN c_emp_cursor
    LOOP
     DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
     ||' ' ||emp_record.last_name);
    END LOOP;
END;
/
```

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Cursor FOR Loops (continued)

The example that was used to demonstrate the usage of a simple loop to fetch data from cursors is rewritten to use the cursor `FOR` loop.

`emp_record` is the record that is implicitly declared. You can access the fetched data with this implicit record (as shown in the slide). Observe that no variables are declared to hold the fetched data using the `INTO` clause. The code does not have the `OPEN` and `CLOSE` statements to open and close the cursor, respectively.

Explicit Cursor Attributes

Use explicit cursor attributes to obtain status information about a cursor.

Attribute	Type	Description
`%ISOPEN`	Boolean	Evaluates to `TRUE` if the cursor is open
`%NOTFOUND`	Boolean	Evaluates to `TRUE` if the most recent fetch does not return a row
`%FOUND`	Boolean	Evaluates to `TRUE` if the most recent fetch returns a row; complement of `%NOTFOUND`
`%ROWCOUNT`	Number	Evaluates to the total number of rows returned so far

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Explicit Cursor Attributes

As with implicit cursors, there are four attributes for obtaining the status information of a cursor. When appended to the cursor variable name, these attributes return useful information about the execution of a cursor manipulation statement.

Note: You cannot reference cursor attributes directly in a SQL statement.

%ISOPEN Attribute

- You can fetch rows only when the cursor is open.
- Use the %ISOPEN cursor attribute before performing a fetch to test whether the cursor is open.

Example:

```
IF NOT c_emp_cursor%ISOPEN THEN
   OPEN c_emp_cursor;
END IF;
LOOP
  FETCH c_emp_cursor...
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

%ISOPEN Attribute

- You can fetch rows only when the cursor is open. Use the %ISOPEN cursor attribute to determine whether the cursor is open.
- Fetch rows in a loop. Use cursor attributes to determine when to exit the loop.
- Use the %ROWCOUNT cursor attribute to do the following:
  - Process an exact number of rows.
  - Fetch the rows in a loop and determine when to exit the loop.

Note: %ISOPEN returns the status of the cursor: TRUE if open and FALSE if not.

%ROWCOUNT and %NOTFOUND: Example

```
DECLARE
  CURSOR c_emp_cursor IS SELECT employee_id,
    last_name FROM employees;
  v_emp_record   c_emp_cursor%ROWTYPE;
BEGIN
  OPEN c_emp_cursor;
  LOOP
   FETCH c_emp_cursor INTO v_emp_record;
   EXIT WHEN c_emp_cursor%ROWCOUNT > 10 OR
                     c_emp_cursor%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE( v_emp_record.employee_id
               ||' '||v_emp_record.last_name);
  END LOOP;
  CLOSE c_emp_cursor;
END ; /
```

```
anonymous block completed
174 Abel
166 Ande
130 Atkinson
105 Austin
204 Baer
116 Baida
167 Banda
172 Bates
192 Bell
151 Bernstein
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

%ROWCOUNT and %NOTFOUND: Example

The example in the slide retrieves the first 10 employees one by one. This example shows how the %ROWCOUNT and %NOTFOUND attributes can be used for exit conditions in a loop.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Arjun M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Cursor FOR Loops Using Subqueries

There is no need to declare the cursor.

```
BEGIN
  FOR emp_record IN (SELECT employee_id, last_name
   FROM employees WHERE department_id =30)
  LOOP
   DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
   ||' '||emp_record.last_name);
  END LOOP;
END;
/
```

```
anonymous block completed
114 Raphaely
115 Khoo
116 Baida
117 Tobias
118 Himuro
119 Colmenares
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Cursor FOR Loops Using Subqueries

Note that there is no declarative section in this PL/SQL block. The difference between the cursor FOR loops using subqueries and the cursor FOR loop lies in the cursor declaration. If you are writing cursor FOR loops using subqueries, you need not declare the cursor in the declarative section. You have to provide the SELECT statement that determines the active set in the loop itself.

The example that was used to illustrate a cursor FOR loop is rewritten to illustrate a cursor FOR loop using subqueries.

Note: You cannot reference explicit cursor attributes if you use a subquery in a cursor FOR loop because you cannot give the cursor an explicit name.

Agenda

- What are explicit cursors?
- Using explicit cursors
- Using cursors with parameters
- Locking rows and referencing the current row

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Cursors with Parameters

Syntax:

```
CURSOR cursor_name
  [(parameter_name datatype, ...)]
IS
  select_statement;
```

- Pass parameter values to a cursor when the cursor is opened and the query is executed.
- Open an explicit cursor several times with a different active set each time.

```
OPEN  cursor_name(parameter_value,.....) ;
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Cursors with Parameters

You can pass parameters to a cursor. This means that you can open and close an explicit cursor several times in a block, returning a different active set on each occasion. For each execution, the previous cursor is closed and reopened with a new set of parameters.

Each formal parameter in the cursor declaration must have a corresponding actual parameter in the `OPEN` statement. Parameter data types are the same as those for scalar variables, but you do not give them sizes. The parameter names are for reference in the query expression of the cursor.

In the syntax:

cursor_name	Is a PL/SQL identifier for the declared cursor
parameter_name	Is the name of a parameter
datatype	Is the scalar data type of the parameter
select_statement	Is a `SELECT` statement without the `INTO` clause

The parameter notation does not offer greater functionality; it simply allows you to specify input values easily and clearly. This is particularly useful when the same cursor is referenced repeatedly.

Cursors with Parameters

```
DECLARE
  CURSOR   c_emp_cursor (deptno NUMBER) IS
   SELECT  employee_id, last_name
   FROM    employees
   WHERE   department_id = deptno;
   ...
BEGIN
  OPEN c_emp_cursor (10);
  ...
  CLOSE c_emp_cursor;
  OPEN c_emp_cursor (20);
  ...
```

```
anonymous block completed
200 Whalen
201 Hartstein
202 Fay
```

Copyright © 2009, Oracle. All rights reserved.

Cursors with Parameters (continued)

Parameter data types are the same as those for scalar variables, but you do not give them sizes. The parameter names are for reference in the cursor's query. In the following example, a cursor is declared and is defined with one parameter:

```
DECLARE
   CURSOR c_emp_cursor(deptno NUMBER) IS SELECT ...
```

The following statements open the cursor and return different active sets:

```
OPEN c_emp_cursor(10);
OPEN c_emp_cursor(20);
```

You can pass parameters to the cursor that is used in a cursor `FOR` loop:

```
DECLARE
CURSOR c_emp_cursor(p_deptno NUMBER, p_job VARCHAR2)IS
   SELECT ...
BEGIN
   FOR emp_record IN c_emp_cursor(10, 'Sales') LOOP ...
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Agenda

- What are explicit cursors?
- Using explicit cursors
- Using cursors with parameters
- Locking rows and referencing the current row

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

FOR UPDATE Clause

Syntax:

```
SELECT ...
FROM          ...
FOR UPDATE [OF column_reference] [NOWAIT | WAIT n];
```

- Use explicit locking to deny access to other sessions for the duration of a transaction.
- Lock the rows *before* the update or delete.

Copyright © 2009, Oracle. All rights reserved.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

FOR UPDATE Clause

If there are multiple sessions for a single database, there is the possibility that the rows of a particular table were updated after you opened your cursor. You see the updated data only when you reopen the cursor. Therefore, it is better to have locks on the rows before you update or delete rows. You can lock the rows with the `FOR UPDATE` clause in the cursor query.

In the syntax:

column_reference	Is a column in the table against which the query is performed (A list of columns may also be used.)
`NOWAIT`	Returns an Oracle Server error if the rows are locked by another session

The `FOR UPDATE` clause is the last clause in a `SELECT` statement, even after `ORDER BY` (if it exists). When you want to query multiple tables, you can use the `FOR UPDATE` clause to confine row locking to particular tables. `FOR UPDATE OF col_name(s)` locks rows only in tables that contain `col_name(s)`.

FOR UPDATE Clause (continued)

The SELECT . . . FOR UPDATE statement identifies the rows that are to be updated or deleted, and then locks each row in the result set. This is useful when you want to base an update on the existing values in a row. In that case, you must make sure that the row is not changed by another session before the update.

The optional NOWAIT keyword tells the Oracle Server not to wait if the requested rows have been locked by another user. Control is immediately returned to your program so that it can do other work before trying again to acquire the lock. If you omit the NOWAIT keyword, the Oracle Server waits until the rows are available.

Example:

```
DECLARE
  CURSOR c_emp_cursor IS
  SELECT employee_id, last_name, FROM employees
  WHERE department_id = 80 FOR UPDATE OF salary NOWAIT;
  ...
```

If the Oracle Server cannot acquire the locks on the rows it needs in a SELECT FOR UPDATE operation, it waits indefinitely. Use NOWAIT to handle such situations. If the rows are locked by another session and you have specified NOWAIT, opening the cursor results in an error. You can try to open the cursor later. You can use WAIT instead of NOWAIT, specify the number of seconds to wait, and then determine whether the rows are unlocked. If the rows are still locked after *n* seconds, an error is returned.

It is not mandatory for the FOR UPDATE OF clause to refer to a column, but it is recommended for better readability and maintenance.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

WHERE CURRENT OF Clause

Syntax:

```
WHERE CURRENT OF cursor ;
```

- Use cursors to update or delete the current row.
- Include the FOR UPDATE clause in the cursor query to first lock the rows.
- Use the WHERE CURRENT OF clause to reference the current row from an explicit cursor.

```
UPDATE employees
   SET    salary = ...
   WHERE CURRENT OF c_emp_cursor;
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

WHERE CURRENT OF Clause

The WHERE CURRENT OF clause is used in conjunction with the FOR UPDATE clause to refer to the current row in an explicit cursor. The WHERE CURRENT OF clause is used in the UPDATE or DELETE statement, whereas the FOR UPDATE clause is specified in the cursor declaration. You can use the combination for updating and deleting the current row from the corresponding database table. This enables you to apply updates and deletes to the row currently being addressed, without the need to explicitly reference the row ID. You must include the FOR UPDATE clause in the cursor query so that the rows are locked on OPEN.

In the syntax:

cursor Is the name of a declared cursor (The cursor must have been declared with the FOR UPDATE clause.)

Quiz

Implicit cursors are declared by PL/SQL implicitly for all DML and PL/SQL `SELECT` statements. The Oracle Server implicitly opens a cursor to process each SQL statement that is not associated with an explicitly declared cursor.

1. True
2. False

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Answer: 1

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Summary

In this lesson, you should have learned to:

- Distinguish cursor types:
  - Implicit cursors are used for all DML statements and single-row queries.
  - Explicit cursors are used for queries of zero, one, or more rows.
- Create and handle explicit cursors
- Use simple loops and cursor `FOR` loops to handle multiple rows in the cursors
- Evaluate cursor status by using cursor attributes
- Use the `FOR UPDATE` and `WHERE CURRENT OF` clauses to update or delete the current fetched row

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankursi@gmail.com) has a non-transferable license to use this Student Guide.

Summary

The Oracle Server uses work areas to execute SQL statements and store processing information. You can use a PL/SQL construct called a *cursor* to name a work area and access its stored information. There are two kinds of cursors: implicit and explicit. PL/SQL implicitly declares a cursor for all SQL data manipulation statements, including queries that return only one row. For queries that return multiple rows, you must explicitly declare a cursor to process the rows individually.

Every explicit cursor and cursor variable has four attributes: `%FOUND`, `%ISOPEN`, `%NOTFOUND`, and `%ROWCOUNT`. When appended to the cursor variable name, these attributes return useful information about the execution of a SQL statement. You can use cursor attributes in procedural statements but not in SQL statements.

Use simple loops or cursor `FOR` loops to operate on the multiple rows fetched by the cursor. If you are using simple loops, you have to open, fetch, and close the cursor; however, cursor `FOR` loops do this implicitly. If you are updating or deleting rows, lock the rows by using a `FOR UPDATE` clause. This ensures that the data you are using is not updated by another session after you open the cursor. Use a `WHERE CURRENT OF` clause in conjunction with the `FOR UPDATE` clause to reference the current row fetched by the cursor.

Practice 7: Overview

This practice covers the following topics:

- Declaring and using explicit cursors to query rows of a table
- Using a cursor `FOR` loop
- Applying cursor attributes to test the cursor status
- Declaring and using cursors with parameters
- Using the `FOR UPDATE` and `WHERE CURRENT OF` clauses

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Practice 7: Overview

In this practice, you apply your knowledge of cursors to process a number of rows from a table and populate another table with the results using a cursor `FOR` loop. You also write a cursor with parameters.

Humber College

8

Handling Exceptions

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Objectives

After completing this lesson, you should be able to do the following:

- 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

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

You learned to write PL/SQL blocks with a declarative section and an executable section. All the SQL and PL/SQL code that must be executed is written in the executable block.

So far it has been assumed that the code works satisfactorily if you take care of compile-time errors. However, the code may cause some unanticipated errors at run time. In this lesson, you learn how to deal with such errors in the PL/SQL block.

Agenda

- Understanding PL/SQL exceptions
- Trapping exceptions

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

What Is an Exception?

```
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname
  FROM employees
  WHERE first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is :' ||v_lname);
END;
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
Error starting at line 3 in command:
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname FROM employees WHERE
  first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is : ' ||v_lname);
END;
Error report:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
01422. 00000 -  "exact fetch returns more than requested number of rows"
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Anton M. Prajapati (prajapati.anton@gmail.com) has a non-transferable license to use this Student Guide.

What Is an Exception?

Consider the example shown in the slide. There are no syntax errors in the code, which means that you must be able to successfully execute the anonymous block. The `SELECT` statement in the block retrieves the last name of John.

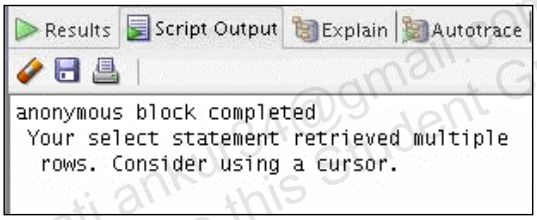
However, you see the following error report when you execute the code:

```
Error report:
ORA-01422: exact fetch returns more than requested number of rows
ORA-06512: at line 4
01422. 00000 -  "exact fetch returns more than requested number of rows"
*Cause:    The number specified in exact fetch is less than the rows returned.
*Action:   Rewrite the query or change number of rows requested
```

The code does not work as expected. You expected the `SELECT` statement to retrieve only one row; however, it retrieves multiple rows. Such errors that occur at run time are called *exceptions*. When an exception occurs, the PL/SQL block is terminated. You can handle such exceptions in your PL/SQL block.

Handling the Exception: An Example

```
DECLARE
  v_lname VARCHAR2(15);
BEGIN
  SELECT last_name INTO v_lname
  FROM employees
  WHERE first_name='John';
  DBMS_OUTPUT.PUT_LINE ('John''s last name is :' ||v_lname);
EXCEPTION
  WHEN TOO_MANY_ROWS THEN
  DBMS_OUTPUT.PUT_LINE (' Your select statement retrieved
   multiple rows. Consider using a cursor.');
END;
/
```



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ashish M. Prajapati (prajapati.ankit.ca@gmail.com) has a non-transferable license to use this Student Guide.

Handling the Exception: An Example

You have previously learned how to write PL/SQL blocks with a declarative section (beginning with the `DECLARE` keyword) and an executable section (beginning and ending with the `BEGIN` and `END` keywords, respectively).

For exception handling, you include another optional section called the *exception section*.

- This section begins with the `EXCEPTION` keyword.
- If present, this must be the last section in a PL/SQL block.

Example

In the example in the slide, the code from the previous slide is rewritten to handle the exception that occurred. The output of the code is shown in the slide as well.

By adding the `EXCEPTION` section of the code, the PL/SQL program does not terminate abruptly. When the exception is raised, the control shifts to the exception section and all the statements in the exception section are executed. The PL/SQL block terminates with normal, successful completion

Understanding Exceptions with PL/SQL

- An exception is a PL/SQL error that is raised during program execution.
- An exception can be raised:
  - Implicitly by the Oracle Server
  - Explicitly by the program
- An exception can be handled:
  - By trapping it with a handler
  - By propagating it to the calling environment

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

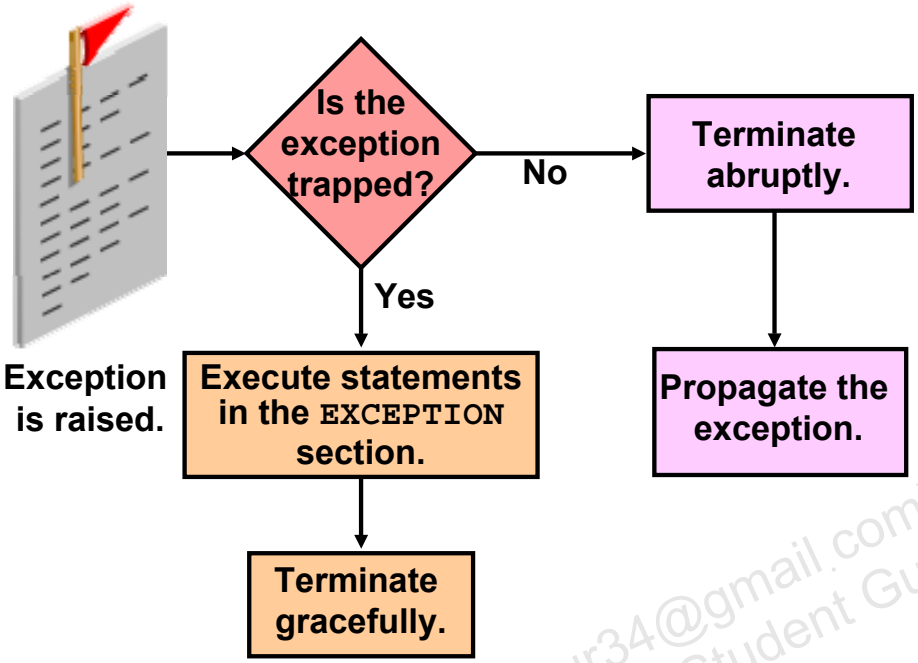
Understanding Exceptions with PL/SQL

An exception is an error in PL/SQL that is raised during the execution of a block. A block always terminates when PL/SQL raises an exception, but you can specify an exception handler to perform final actions before the block ends.

Two Methods for Raising an Exception

- An Oracle error occurs and the associated exception is raised automatically. For example, if the `ORA-01403` error occurs when no rows are retrieved from the database in a `SELECT` statement, PL/SQL raises the `NO_DATA_FOUND` exception. These errors are converted into predefined exceptions.
- Depending on the business functionality your program implements, you may have to explicitly raise an exception. You raise an exception explicitly by issuing the `RAISE` statement in the block. The raised exception may be either user-defined or predefined. There are also some non-predefined Oracle errors. These errors are any standard Oracle errors that are not predefined. You can explicitly declare exceptions and associate them with the non-predefined Oracle errors.

Handling Exceptions



ORACLE
Copyright © 2009, Oracle. All rights reserved.

Handling Exceptions

Trapping an Exception

Include an `EXCEPTION` section in your PL/SQL program to trap exceptions. If the exception is raised in the executable section of the block, processing branches to the corresponding exception handler in the exception section of the block. If PL/SQL successfully handles the exception, the exception does not propagate to the enclosing block or to the calling environment. The PL/SQL block terminates successfully.

Propagating an Exception

If the exception is raised in the executable section of the block and there is no corresponding exception handler, the PL/SQL block terminates with failure and the exception is propagated to an enclosing block or to the calling environment. The calling environment can be any application (such as SQL*Plus that invokes the PL/SQL program).

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Exception Types

- Predefined Oracle Server
- Non-predefined Oracle Server

} **Implicitly raised**

- User-defined

Explicitly raised

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Exception Types

There are three types of exceptions.

Exception	Description	Directions for Handling
Predefined Oracle Server error	One of approximately 20 errors that occur most often in PL/SQL code	You need not declare these exceptions. They are predefined by the Oracle server and are raised implicitly.
Non-predefined Oracle Server error	Any other standard Oracle Server error	You need to declare these within the declarative section; the Oracle server raises the error implicitly, and you can catch the error in the exception handler.
User-defined error	A condition that the developer determines is abnormal	You need to declare in the declarative section and raise explicitly.

Note: Some application tools with client-side PL/SQL (such as Oracle Developer Forms) have their own exceptions.

Agenda

- Understanding PL/SQL exceptions
- Trapping exceptions

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Syntax to Trap Exceptions

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Syntax to Trap Exceptions

You can trap any error by including a corresponding handler within the exception-handling section of the PL/SQL block. Each handler consists of a `WHEN` clause, which specifies an exception name, followed by a sequence of statements to be executed when that exception is raised.

You can include any number of handlers within an `EXCEPTION` section to handle specific exceptions. However, you cannot have multiple handlers for a single exception.

Exception trapping syntax includes the following elements:

exception	Is the standard name of a predefined exception or the name of a user-defined exception declared within the declarative section
statement	Is one or more PL/SQL or SQL statements
`OTHERS`	Is an optional exception-handling clause that traps any exceptions that have not been explicitly handled

Exception Trapping Syntax (continued)

WHEN OTHERS Exception Handler

As stated previously, the exception-handling section traps only those exceptions that are specified.

To trap any exceptions that are not specified, you use the OTHERS exception handler. This option traps any exception not yet handled. For this reason, if the OTHERS handler is used, it must be the last exception handler that is defined.

For example:

```
WHEN NO_DATA_FOUND THEN
 statement1;
 ...
WHEN TOO_MANY_ROWS THEN
 statement1;
 ...
WHEN OTHERS THEN
 statement1;
```

Example

Consider the preceding example. If the NO_DATA_FOUND exception is raised by the program, the statements in the corresponding handler are executed. If the TOO_MANY_ROWS exception is raised, the statements in the corresponding handler are executed. However, if some other exception is raised, the statements in the OTHERS exception handler are executed.

The OTHERS handler traps all the exceptions that are not already trapped. Some Oracle tools have their own predefined exceptions that you can raise to cause events in the application. The OTHERS handler also traps these exceptions.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankurm@gmail.com) has a non-transferable license to use this Student Guide

Guidelines for Trapping Exceptions

- The `EXCEPTION` keyword starts the exception-handling section.
- Several exception handlers are allowed.
- Only one handler is processed before leaving the block.
- `WHEN OTHERS` is the last clause.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Guidelines for Trapping Exceptions

- Begin the exception-handling section of the block with the `EXCEPTION` keyword.
- Define several exception handlers, each with its own set of actions, for the block.
- When an exception occurs, PL/SQL processes only one handler before leaving the block.
- Place the `OTHERS` clause after all other exception-handling clauses.
- You can have only one `OTHERS` clause.
- Exceptions cannot appear in assignment statements or SQL statements.

Trapping Predefined Oracle Server Errors

- Reference the predefined name in the exception-handling routine.
- Sample predefined exceptions:
  - `NO_DATA_FOUND`
  - `TOO_MANY_ROWS`
  - `INVALID_CURSOR`
  - `ZERO_DIVIDE`
  - `DUP_VAL_ON_INDEX`

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Trapping Predefined Oracle Server Errors

Trap a predefined Oracle Server error by referencing its predefined name within the corresponding exception-handling routine.

For a complete list of predefined exceptions, see the *PL/SQL User's Guide and Reference*.

Note: PL/SQL declares predefined exceptions in the `STANDARD` package.

Predefined Exceptions

Exception Name	Oracle Server Error Number	Description
ACCESS_INTO_NULL	ORA-06530	Attempted to assign values to the attributes of an uninitialized object
CASE_NOT_FOUND	ORA-06592	None of the choices in the WHEN clauses of a CASE statement are selected, and there is no ELSE clause.
COLLECTION_IS_NULL	ORA-06531	Attempted to apply collection methods other than EXISTS to an uninitialized nested table or VARRAY
CURSOR_ALREADY_OPEN	ORA-06511	Attempted to open an already open cursor
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value
INVALID_CURSOR	ORA-01001	Illegal cursor operation occurred.
INVALID_NUMBER	ORA-01722	Conversion of character string to number failed.
LOGIN_DENIED	ORA-01017	Logging on to the Oracle server with an invalid username or password
NO_DATA_FOUND	ORA-01403	Single row SELECT returned no data.
NOT_LOGGED_ON	ORA-01012	The PL/SQL program issues a database call without being connected to the Oracle server.
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	ORA-06504	The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

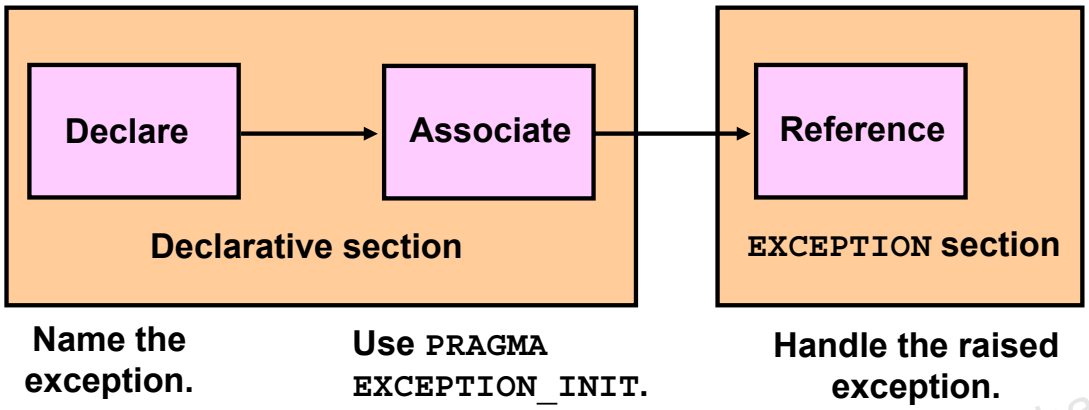
Predefined Exceptions (continued)

Exception Name	Oracle Server Error Number	Description
STORAGE_ERROR	ORA-06500	PL/SQL ran out of memory, or memory is corrupted.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Referenced a nested table or VARRAY element by using an index number larger than the number of elements in the collection
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Referenced a nested table or VARRAY element by using an index number that is outside the legal range (for example, –1)
SYS_INVALID_ROWID	ORA-01410	The conversion of a character string into a universal ROWID fails because the character string does not represent a valid ROWID.
TIMEOUT_ON_RESOURCE	ORA-00051	Time-out occurred while the Oracle server was waiting for a resource.
TOO_MANY_ROWS	ORA-01422	Single-row SELECT returned multiple rows.
VALUE_ERROR	ORA-06502	Arithmetic, conversion, truncation, or size-constraint error occurred.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Trapping Non-Predefined Oracle Server Errors



Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Trapping Non-Predefined Oracle Server Errors

Non-predefined exceptions are similar to predefined exceptions; however, they are not defined as PL/SQL exceptions in the Oracle Server. They are standard Oracle errors. You create exceptions with standard Oracle errors by using the `PRAGMA EXCEPTION_INIT` function. Such exceptions are called non-predefined exceptions.

You can trap a non-predefined Oracle Server error by declaring it first. The declared exception is raised implicitly. In PL/SQL, `PRAGMA EXCEPTION_INIT` tells the compiler to associate an exception name with an Oracle error number. This enables you to refer to any internal exception by name and to write a specific handler for it.

Note: `PRAGMA` (also called *pseudoinstructions*) is the keyword that signifies that the statement is a compiler directive, which is not processed when the PL/SQL block is executed. Rather, it directs the PL/SQL compiler to interpret all occurrences of the exception name within the block as the associated Oracle Server error number.

Non-Predefined Error Trapping: Example

To trap Oracle Server error `01400` ("cannot insert `NULL`"):

```
DECLARE
 e_insert_excep EXCEPTION;                          (1)
 PRAGMA EXCEPTION_INIT(e_insert_excep, -01400);     (2)
BEGIN
 INSERT INTO departments
 (department_id, department_name) VALUES (280, NULL);
EXCEPTION                                           (3)
 WHEN e_insert_excep THEN
   DBMS_OUTPUT.PUT_LINE('INSERT OPERATION FAILED');
   DBMS_OUTPUT.PUT_LINE(SQLERRM);
END;
/
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

```
anonymous block completed
INSERT OPERATION FAILED
ORA-01400: cannot insert NULL into ("ORA41"."DEPARTMENTS"."DEPARTMENT_NAME")
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide

Non-Predefined Error Trapping: Example

The example illustrates the three steps associated with trapping a non-predefined error:

1. Declare the name of the exception in the declarative section, using the syntax:
   exception `EXCEPTION;`
   In the syntax, *exception* is the name of the exception.
2. Associate the declared exception with the standard Oracle Server error number by using the `PRAGMA EXCEPTION_INIT` function. Use the following syntax:
   `PRAGMA EXCEPTION_INIT(`*exception, error_number*`);`
   In the syntax, *exception* is the previously declared exception and *error_number* is a standard Oracle Server error number.
3. Reference the declared exception within the corresponding exception-handling routine.

Example

The example in the slide tries to insert the `NULL` value for the `department_name` column of the `departments` table. However, the operation is not successful because `department_name` is a `NOT NULL` column. Note the following line in the example:

```
DBMS_OUTPUT.PUT_LINE(SQLERRM);
```

The `SQLERRM` function is used to retrieve the error message. You learn more about `SQLERRM` in the next few slides.

Functions for Trapping Exceptions

- SQLCODE: Returns the numeric value for the error code
- SQLERRM: Returns the message associated with the error number

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Functions for Trapping Exceptions

When an exception occurs, you can identify the associated error code or error message by using two functions. Based on the values of the code or the message, you can decide which subsequent actions to take.

SQLCODE returns the Oracle error number for internal exceptions. SQLERRM returns the message associated with the error number.

Function	Description
SQLCODE	Returns the numeric value for the error code (You can assign it to a NUMBER variable.)
SQLERRM	Returns character data containing the message associated with the error number

SQLCODE Values: Examples

SQLCODE Value	Description
0	No exception encountered
1	User-defined exception
+100	NO_DATA_FOUND exception
negative number	Another Oracle server error number

Functions for Trapping Exceptions

```
DECLARE
  error_code      NUMBER;
  error_message   VARCHAR2(255);
BEGIN
...
EXCEPTION
...
  WHEN OTHERS THEN
    ROLLBACK;
    error_code := SQLCODE ;
    error_message := SQLERRM ;
   INSERT INTO errors (e_user, e_date, error_code,
   error_message) VALUES(USER,SYSDATE,error_code,
   error_message);
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

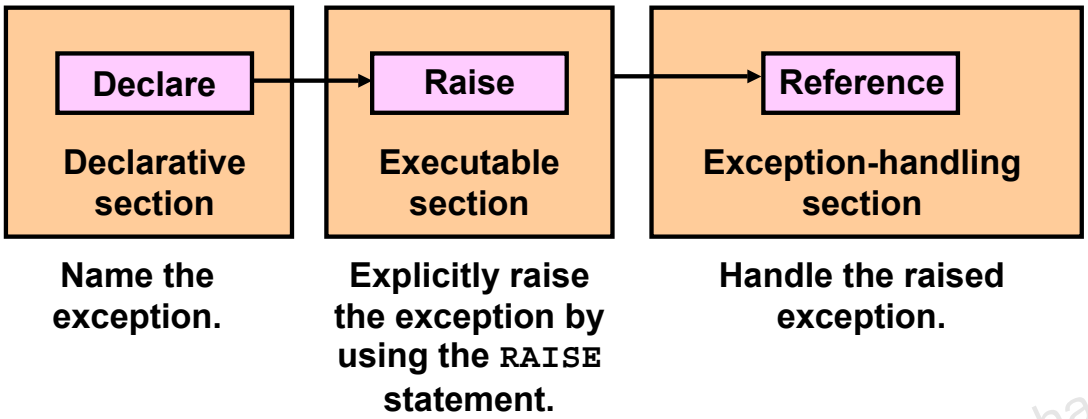
Functions for Trapping Exceptions (continued)

When an exception is trapped in the `WHEN OTHERS` exception handler, you can use a set of generic functions to identify those errors. The example in the slide illustrates the values of `SQLCODE` and `SQLERRM` assigned to variables, and then those variables being used in a SQL statement.

You cannot use `SQLCODE` or `SQLERRM` directly in a SQL statement. Instead, you must assign their values to local variables, and then use the variables in the SQL statement, as shown in the following example:

```
DECLARE
   err_num NUMBER;
   err_msg VARCHAR2(100);
BEGIN
   ...
EXCEPTION
   ...
   WHEN OTHERS THEN
      err_num := SQLCODE;
      err_msg := SUBSTR(SQLERRM, 1, 100);
      INSERT INTO errors VALUES (err_num, err_msg);
END;
/
```

Trapping User-Defined Exceptions



Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Trapping User-Defined Exceptions

PL/SQL enables you to define your own exceptions depending on the requirements of your application. For example, you may prompt the user to enter a department number. Define an exception to deal with error conditions in the input data. Check whether the department number exists. If it does not, you may have to raise the user-defined exception.

PL/SQL exceptions must be:

- Declared in the declarative section of a PL/SQL block
- Raised explicitly with `RAISE` statements
- Handled in the `EXCEPTION` section

Trapping User-Defined Exceptions

```
DECLARE
  v_deptno NUMBER := 500;
  v_name VARCHAR2(20) := 'Testing';
  e_invalid_department EXCEPTION;
BEGIN
  UPDATE departments
  SET department_name = v_name
  WHERE department_id = v_deptno;
  IF SQL%NOTFOUND THEN
    RAISE e_invalid_department;
  END IF;
  COMMIT;
EXCEPTION
WHEN e_invalid_department THEN
  DBMS_OUTPUT.PUT_LINE('No such department id.');
END;
/
```

Results | Script Output | Explain

```
anonymous block completed
No such department id.
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (ankur.m.prajapati@gmail.com) has a non-transferable license to use this Student Guide.

Trapping User-Defined Exceptions (continued)

You trap a user-defined exception by declaring it and raising it explicitly.

1. Declare the name of the user-defined exception within the declarative section.
   Syntax:
   `exception EXCEPTION;`
   In the syntax, *exception* is the name of the exception.
2. Use the `RAISE` statement to raise the exception explicitly within the executable section.
   Syntax:
   `RAISE exception;`
   In the syntax, *exception* is the previously declared exception.
3. Reference the declared exception within the corresponding exception-handling routine.

Example

The block shown in the slide updates the `department_name` of a department. The user supplies the department number and the new name. If the supplied department number does not exist, no rows are updated in the `departments` table. An exception is raised and a message is printed for the user that an invalid department number was entered.

Note: Use the `RAISE` statement by itself within an exception handler to raise the same exception again and propagate it back to the calling environment.

Propagating Exceptions in a Subblock

Subblocks can handle an exception or pass the exception to the enclosing block.

```
DECLARE
  . . .
  e_no_rows       exception;
  e_integrity     exception;
  PRAGMA EXCEPTION_INIT (e_integrity, -2292);
BEGIN
  FOR c_record IN emp_cursor LOOP
    BEGIN
     SELECT ...
     UPDATE ...
     IF SQL%NOTFOUND THEN
       RAISE e_no_rows;
     END IF;
    END;
  END LOOP;
EXCEPTION
  WHEN e_integrity THEN ...
  WHEN e_no_rows THEN ...
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Propagating Exceptions in a Subblock

When a subblock handles an exception, it terminates normally. Control resumes in the enclosing block immediately after the subblock's `END` statement.

However, if a PL/SQL raises an exception and the current block does not have a handler for that exception, the exception propagates to successive enclosing blocks until it finds a handler. If none of these blocks handles the exception, an unhandled exception in the host environment results.

When the exception propagates to an enclosing block, the remaining executable actions in that block are bypassed.

One advantage of this behavior is that you can enclose statements that require their own exclusive error handling in their own block, while leaving more general exception handling to the enclosing block.

Note in the example that the exceptions (`no_rows` and `integrity`) are declared in the outer block. In the inner block, when the `no_rows` exception is raised, PL/SQL looks for the exception to be handled in the subblock. Because the exception is not handled in the subblock, the exception propagates to the outer block, where PL/SQL finds the handler.

RAISE_APPLICATION_ERROR Procedure

Syntax:

```
raise_application_error (error_number,
                         message[, {TRUE | FALSE}]);
```

- You can use this procedure to issue user-defined error messages from stored subprograms.
- You can report errors to your application and avoid returning unhandled exceptions.

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

RAISE_APPLICATION_ERROR Procedure

Use the RAISE_APPLICATION_ERROR procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With RAISE_APPLICATION_ERROR, you can report errors to your application and avoid returning unhandled exceptions.

In the syntax:

error_number	Is a user-specified number for the exception between –20,000 and –20,999
message	Is the user-specified message for the exception; is a character string up to 2,048 bytes long
TRUE | FALSE	Is an optional Boolean parameter (If TRUE, the error is placed on the stack of previous errors. If FALSE, which is the default, the error replaces all previous errors.)

RAISE_APPLICATION_ERROR Procedure

- Is used in two different places:
  - Executable section
  - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle Server errors

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

RAISE_APPLICATION_ERROR Procedure (continued)

The RAISE_APPLICATION_ERROR procedure can be used in either the executable section or the exception section of a PL/SQL program, or both. The returned error is consistent with how the Oracle Server produces a predefined, non-predefined, or user-defined error. The error number and message are displayed to the user.

RAISE_APPLICATION_ERROR Procedure

Executable section:

```
BEGIN
...
  DELETE FROM employees
     WHERE  manager_id = v_mgr;
  IF SQL%NOTFOUND THEN
     RAISE_APPLICATION_ERROR(-20202,
       'This is not a valid manager');
  END IF;
   ...
```

Exception section:

```
...
EXCEPTION
   WHEN NO_DATA_FOUND THEN
     RAISE_APPLICATION_ERROR (-20201,
        'Manager is not a valid employee.');
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

RAISE_APPLICATION_ERROR Procedure (continued)

The slide shows that the RAISE_APPLICATION_ERROR procedure can be used in both the executable and the exception sections of a PL/SQL program.

Here is another example of using the RAISE_APPLICATION_ERROR procedure:

```
DECLARE
  e_name EXCEPTION;
  BEGIN
  ...
  DELETE FROM employees
  WHERE  last_name = 'Higgins';
  IF SQL%NOTFOUND THEN RAISE e_name;
  END IF;
EXCEPTION
  WHEN e_name THEN
    RAISE_APPLICATION_ERROR (-20999, 'This is not a valid
last name');   ...
END;
/
```

Quiz

You can trap any error by including a corresponding handler within the exception-handling section of the PL/SQL block.

1. True
2. False

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Answer: 1

You can trap any error by including a corresponding handler within the exception-handling section of the PL/SQL block. Each handler consists of a `WHEN` clause, which specifies an exception name, followed by a sequence of statements to be executed when that exception is raised. You can include any number of handlers within an `EXCEPTION` section to handle specific exceptions. However, you cannot have multiple handlers for a single exception.

Summary

In this lesson, you should have learned to:

- Define PL/SQL exceptions
- Add an `EXCEPTION` section to the PL/SQL block to deal with exceptions at run time
- Handle different types of exceptions:
  - Predefined exceptions
  - Non-predefined exceptions
  - User-defined exceptions
- Propagate exceptions in nested blocks and call applications

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Summary

In this lesson, you learned how to deal with different types of exceptions. In PL/SQL, a warning or error condition at run time is called an exception. Predefined exceptions are error conditions that are defined by the Oracle Server. Non-predefined exceptions can be any standard Oracle Server errors. User-defined exceptions are exceptions specific to your application. The `PRAGMA EXCEPTION_INIT` function can be used to associate a declared exception name with an Oracle Server error.

You can define exceptions of your own in the declarative section of any PL/SQL block. For example, you can define an exception named `INSUFFICIENT_FUNDS` to flag overdrawn bank accounts.

When an error occurs, an exception is raised. Normal execution stops and transfers control to the exception-handling section of your PL/SQL block. Internal exceptions are raised implicitly (automatically) by the run-time system; however, user-defined exceptions must be raised explicitly. To handle raised exceptions, you write separate routines called exception handlers.

Practice 8: Overview

This practice covers the following topics:

- Creating and invoking user-defined exceptions
- Handling named Oracle Server exceptions

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Practice 8: Overview

In these practices, you create exception handlers for a predefined exception and a standard Oracle Server exception.

Humber College

9

Introducing Stored Procedures and Functions

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

After completing this lesson, you should be able to do the following:

- Differentiate between anonymous blocks and subprograms
- Create a simple procedure and invoke it from an anonymous block
- Create a simple function
- Create a simple function that accepts a parameter
- Differentiate between procedures and functions

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

You learned about anonymous blocks. This lesson introduces you to named blocks, which are also called *subprograms*. Procedures and functions are PL/SQL subprograms. In the lesson, you learn to differentiate between anonymous blocks and subprograms.

Agenda

- Introducing procedures and functions
- Previewing procedures
- Previewing functions

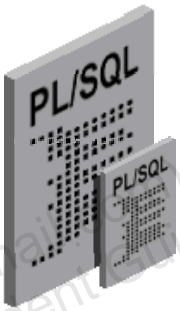
ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Procedures and Functions

- Are named PL/SQL blocks
- Are called PL/SQL subprograms
- Have block structures similar to anonymous blocks:
  - Optional declarative section (without the `DECLARE` keyword)
  - Mandatory executable section
  - Optional section to handle exceptions



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Procedures and Functions

Up to this point, anonymous blocks were the only examples of PL/SQL code covered in this course. As the name indicates, *anonymous* blocks are unnamed executable PL/SQL blocks. Because they are unnamed, they can be neither reused nor stored for later use.

Procedures and functions are named PL/SQL blocks that are also known as *subprograms*. These subprograms are compiled and stored in the database. The block structure of the subprograms is similar to the structure of anonymous blocks. Subprograms can be declared not only at the schema level but also within any other PL/SQL block. A subprogram contains the following sections:

- **Declarative section:** Subprograms can have an optional declarative section. However, unlike anonymous blocks, the declarative section of a subprogram does not start with the `DECLARE` keyword. The optional declarative section follows the `IS` or `AS` keyword in the subprogram declaration.
- **Executable section:** This is the mandatory section of the subprogram, which contains the implementation of the business logic. Looking at the code in this section, you can easily determine the business functionality of the subprogram. This section begins and ends with the `BEGIN` and `END` keywords, respectively.
- **Exception section:** This is an optional section that is included to handle exceptions.

Differences Between Anonymous Blocks and Subprograms

Anonymous Blocks	Subprograms
Unnamed PL/SQL blocks	Named PL/SQL blocks
Compiled every time	Compiled only once
Not stored in the database	Stored in the database
Cannot be invoked by other applications	Named and, therefore, can be invoked by other applications
Do not return values	If functions, must return values
Cannot take parameters	Can take parameters

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Differences Between Anonymous Blocks and Subprograms

The table in the slide not only shows the differences between anonymous blocks and subprograms, but also highlights the general benefits of subprograms.

Anonymous blocks are not persistent database objects. They are compiled every time they are to be executed. They are not stored in the database for reuse. If you want to reuse them, you must rerun the script that creates the anonymous block, which causes recompilation and execution.

Procedures and functions are compiled and stored in the database in a compiled form. They are recompiled only when they are modified. Because they are stored in the database, any application can make use of these subprograms based on appropriate permissions. The calling application can pass parameters to the procedures if the procedure is designed to accept parameters. Similarly, a calling application can retrieve a value if it invokes a function or a procedure.

Agenda

- Introducing procedures and functions
- Previewing procedures
- Previewing functions

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Procedure: Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
 [(argument1 [mode1] datatype1,
  argument2 [mode2] datatype2,
  . . .)]
IS|AS
procedure_body;
```

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Procedure: Syntax

The slide shows the syntax for creating procedures. In the syntax:

procedure_name	Is the name of the procedure to be created
argument	Is the name given to the procedure parameter. Every argument is associated with a mode and data type. You can have any number of arguments separated by commas.

mode	Mode of argument:
	`IN` (default)
	`OUT`
	`IN OUT`

datatype	Is the data type of the associated parameter. The data type of parameters cannot have explicit size; instead, use `%TYPE`.
Procedure_body	Is the PL/SQL block that makes up the code

The argument list is optional in a procedure declaration. You learn about procedures in detail in the course titled *Oracle Database 11g: Develop PL/SQL Program Units*.

Creating a Procedure

```
...
CREATE TABLE dept AS SELECT * FROM departments;
CREATE PROCEDURE add_dept IS
 v_dept_id dept.department_id%TYPE;
 v_dept_name dept.department_name%TYPE;
BEGIN
 v_dept_id:=280;
 v_dept_name:='ST-Curriculum';
 INSERT INTO dept(department_id,department_name)
 VALUES(v_dept_id,v_dept_name);
 DBMS_OUTPUT.PUT_LINE(' Inserted '|| SQL%ROWCOUNT
||' row ');
END;
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating a Procedure

In the code example, the `add_dept` procedure inserts a new department with department ID `280` and department name `ST-Curriculum`.

In addition, the example shows the following:

- The declarative section of a procedure starts immediately after the procedure declaration and does not begin with the `DECLARE` keyword.
- The procedure declares two variables, `dept_id` and `dept_name`.
- The procedure uses the implicit cursor attribute or the `SQL%ROWCOUNT` SQL attribute to verify that the row was successfully inserted. A value of 1 should be returned in this case.

Note: See the following page for more notes on the example.

Procedure: Example

Note

- When you create any object, the entries are made to the `user_objects` table. When the code in the slide is executed successfully, you can check the `user_objects` table for the new objects by issuing the following command:

```
SELECT object_name,object_type FROM user_objects;
```

Results | Script Output | Explain | Autotrace

Results:

	OBJECT_NAME	OBJECT_TYPE
41	COPY_EMP	TABLE
42	DEPT	TABLE
43	GREET	PROCEDURE
44	ADD_DEPT	PROCEDURE
45	MY_SEQ	SEQUENCE

- The source of the procedure is stored in the `user_source` table. You can check the source for the procedure by issuing the following command:

```
SELECT * FROM user_source WHERE name='ADD_DEPT';
```

Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

Results:

	NAME	TYPE	LINE	TEXT
1	ADD_DEPT	PROCEDURE	1	PROCEDURE add_dept IS
2	ADD_DEPT	PROCEDURE	2	v_dept_id dept.department_id%TYPE;
3	ADD_DEPT	PROCEDURE	3	v_dept_name dept.department_name%TYPE;
4	ADD_DEPT	PROCEDURE	4	BEGIN
5	ADD_DEPT	PROCEDURE	5	v_dept_id:=280;
6	ADD_DEPT	PROCEDURE	6	v_dept_name:='ST-Curriculum';
7	ADD_DEPT	PROCEDURE	7	INSERT INTO dept(department_id,department_name)
8	ADD_DEPT	PROCEDURE	8	VALUES(v_dept_id,v_dept_name);
9	ADD_DEPT	PROCEDURE	9	DBMS_OUTPUT.PUT_LINE(' Inserted '|| SQL%ROWCOUNT ||' row');
10	ADD_DEPT	PROCEDURE	10	END;

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur.34@gmail.com) has a non-transferable license to use this Student Guide.

Invoking a Procedure

```
...
BEGIN
 add_dept;
END;
/
SELECT department_id, department_name FROM dept
WHERE department_id=280;
```

Results | Script Output | Explain | Autotrace

```
anonymous block completed
 Inserted 1 row

DEPARTMENT_ID          DEPARTMENT_NAME
---------------------- --------------------
280                    ST-Curriculum

1 rows selected
```

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Invoking the Procedure

The slide shows how to invoke a procedure from an anonymous block. You must include the call to the procedure in the executable section of the anonymous block. Similarly, you can invoke the procedure from any application, such as a Forms application or a Java application. The `SELECT` statement in the code checks to see whether the row was successfully inserted.

You can also invoke a procedure with the SQL statement `CALL <procedure_name>`.

Agenda

- Introducing procedures and functions
- Previewing procedures
- Previewing functions

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Function: Syntax

```
CREATE [OR REPLACE] FUNCTION function_name
 [(argument1 [mode1] datatype1,
  argument2 [mode2] datatype2,
  . . .)]
RETURN datatype
IS|AS
function_body;
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Function: Syntax

The slide shows the syntax for creating a function. In the syntax:

function_name	Is the name of the function to be created
argument	Is the name given to the function parameter (Every argument is associated with a mode and data type. You can have any number of arguments separated by a comma. You pass the argument when you invoke the function.)
mode	Is the type of parameter (Only `IN` parameters should be declared.)
datatype	Is the data type of the associated parameter
`RETURN` *datatype*	Is the data type of the value returned by the function
function_body	Is the PL/SQL block that makes up the function code

The argument list is optional in the function declaration. The difference between a procedure and a function is that a function must return a value to the calling program. Therefore, the syntax contains *return_type*, which specifies the data type of the value that the function returns. A procedure may return a value via an `OUT` or `IN OUT` parameter.

Creating a Function

```
CREATE FUNCTION check_sal RETURN Boolean IS
v_dept_id employees.department_id%TYPE;
 v_empno   employees.employee_id%TYPE;
 v_sal     employees.salary%TYPE;
 v_avg_sal employees.salary%TYPE;
BEGIN
 v_empno:=205;
 SELECT salary,department_id INTO v_sal,v_dept_id FROM
employees
 WHERE employee_id= v_empno;
 SELECT avg(salary) INTO v_avg_sal FROM employees WHERE
department_id=v_dept_id;
 IF v_sal > v_avg_sal THEN
  RETURN TRUE;
 ELSE
  RETURN FALSE;
 END IF;
EXCEPTION
  WHEN NO_DATA_FOUND THEN
   RETURN NULL;
END;
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

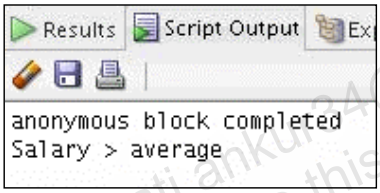
Function: Example

The `check_sal` function is written to determine whether the salary of a particular employee is greater than or less than the average salary of all employees working in the same department. The function returns `TRUE` if the salary of the employee is greater than the average salary of the employees in the department; if not, it returns `FALSE`. The function returns `NULL` if a `NO_DATA_FOUND` exception is thrown.

Note that the function checks for the employee with the employee ID 205. The function is hard-coded to check only for this employee ID. If you want to check for any other employees, you must modify the function itself. You can solve this problem by declaring the function such that it accepts an argument. You can then pass the employee ID as parameter.

Invoking a Function

```
BEGIN
 IF (check_sal IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal) THEN
 DBMS_OUTPUT.PUT_LINE('Salary > average');
 ELSE
 DBMS_OUTPUT.PUT_LINE('Salary < average');
 END IF;
END;
/
```



ORACLE
Copyright © 2009, Oracle. All rights reserved.

Invoking the Function

You include the call to the function in the executable section of the anonymous block. The function is invoked as a part of a statement. Remember that the `check_sal` function returns `Boolean` or `NULL`. Thus the call to the function is included as the conditional expression for the `IF` block.

Note: You can use the `DESCRIBE` command to check the arguments and return type of the function, as in the following example:

```
DESCRIBE check_sal;
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Passing a Parameter to the Function

```
DROP FUNCTION check_sal;
CREATE FUNCTION check_sal(p_empno employees.employee_id%TYPE)
RETURN Boolean IS
 v_dept_id employees.department_id%TYPE;
 v_sal     employees.salary%TYPE;
 v_avg_sal employees.salary%TYPE;
BEGIN
 SELECT salary,department_id INTO v_sal,v_dept_id FROM employees
   WHERE employee_id=p_empno;
 SELECT avg(salary) INTO v_avg_sal FROM employees
   WHERE department_id=v_dept_id;
 IF v_sal > v_avg_sal THEN
  RETURN TRUE;
 ELSE
  RETURN FALSE;
 END IF;
EXCEPTION
  ...
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

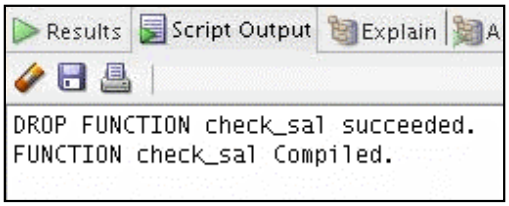
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Passing a Parameter to the Function

Remember that the function was hard-coded to check the salary of the employee with employee ID 205. The code shown in the slide removes that constraint because it is rewritten to accept the employee number as a parameter. You can now pass different employee numbers and check for the employee's salary.

You learn more about functions in the course titled *Oracle Database 11g: Develop PL/SQL Program Units*.

The output of the code example in the slide is as follows:



Invoking the Function with a Parameter

```
BEGIN
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 205');
 IF (check_sal(205) IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal(205)) THEN
 DBMS_OUTPUT.PUT_LINE('Salary > average');
 ELSE
 DBMS_OUTPUT.PUT_LINE('Salary < average');
 END IF;
DBMS_OUTPUT.PUT_LINE('Checking for employee with id 70');
 IF (check_sal(70) IS NULL) THEN
 DBMS_OUTPUT.PUT_LINE('The function returned
  NULL due to exception');
 ELSIF (check_sal(70)) THEN
 ...
 END IF;
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Invoking the Function with a Parameter

The code in the slide invokes the function twice by passing parameters. The output of the code is as follows:

Results | Script Output | Explain | Autotrace

```
anonymous block completed
Checking for employee with id 205
Salary > average
Checking for employee with id 70
The function returned NULL due to exception
```

Quiz

Subprograms:

1. Are named PL/SQL blocks and can be invoked by other applications
2. Are compiled only once
3. Are stored in the database
4. Do not have to return values if they are functions
5. Can take parameters

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Answer: 1, 2, 3, 5

Ankur M. Prajapati (rajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Summary

In this lesson, you should have learned to:

- Create a simple procedure
- Invoke the procedure from an anonymous block
- Create a simple function
- Create a simple function that accepts parameters
- Invoke the function from an anonymous block

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transforable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Summary

You can use anonymous blocks to design any functionality in PL/SQL. However, the major constraint with anonymous blocks is that they are not stored and, therefore, cannot be reused.

Instead of creating anonymous blocks, you can create PL/SQL subprograms. Procedures and functions are called subprograms, which are named PL/SQL blocks. Subprograms express reusable logic by virtue of parameterization. The structure of a procedure or function is similar to the structure of an anonymous block. These subprograms are stored in the database and are, therefore, reusable.

Practice 9: Overview

This practice covers the following topics:

- Converting an existing anonymous block to a procedure
- Modifying the procedure to accept a parameter
- Writing an anonymous block to invoke the procedure

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Appendix A
Practices and Solutions

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Table of Contents

Practices and Solutions for Lesson I 3
Practice I-1: Accessing SQL Developer Resources 3
Practice I-2: Getting Started 3
Solution I-1: Accessing SQL Developer Resources 6
Solution I-2: Getting Started 7
Practices and Solutions for Lesson 1 14
Practice 1: Introduction to PL/SQL 14
Solution 1: Introduction to PL/SQL 15
Practices and Solutions for Lesson 2 16
Practice 2: Declaring PL/SQL Variables 16
Solution 2: Declaring PL/SQL Variables 18
Practices and Solutions for Lesson 3 21
Practice 3: Writing Executable Statements 21
Solution 3: Writing Executable Statements 24
Practices and Solutions for Lesson 4 28
Practice 4: Interacting with the Oracle Server 28
Solution 4: Interacting with the Oracle Server 30
Practices and Solutions for Lesson 5 33
Practice 5: Writing Control Structures 33
Solution 5: Writing Control Structures 35
Practices and Solutions for Lesson 6 38
Practice 6: Working with Composite Data Types 38
Solution 6: Working with Composite Data Types 40
Practices and Solutions for Lesson 7 45
Practice 7-1: Using Explicit Cursors 45
Practice 7-2: Using Explicit Cursors – Optional 48
Solution 7-1: Using Explicit Cursors 49
Solution 7-2: Using Explicit Cursors – Optional 54
Practices and Solutions for Lesson 8 56
Practice 8-1: Handling Predefined Exceptions 56
Practice 8-2: Handling Standard Oracle Server Exceptions 58
Solution 8-1: Handling Predefined Exceptions 59
Solution 8-2: Handling Standard Oracle Server Exceptions 61
Practices and Solutions for Lesson 9 62
Practice 9: Creating and Using Stored Procedures 62
Solution 9: Creating and Using Stored Procedures 64

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur Nahalani (ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practices and Solutions for Lesson I

In these practices, you identify information resources for SQL Developer, execute SQL statements using SQL Developer, and examine data in the class schema. Specifically, you:

- Start SQL Developer
- Create a new database connection
- Browse the schema tables
- Set a SQL Developer preference

Note: All written practices use SQL Developer as the development environment. Although it is recommended that you use SQL Developer, you can also use the SQL*Plus or JDeveloper environments that are available in this course.

Practice I-1: Accessing SQL Developer Resources

In this practice, you navigate to the SQL Developer home page and browse helpful information on the tool.

1) Access the SQL Developer home page.
   a) Access the online SQL Developer Home Page, which is available at:
      http://www.oracle.com/technology/products/database/sql_developer/index.html
   b) Bookmark the page for easier access in future.
2) Access the SQL Developer tutorial, which is available online at http://st-curriculum.oracle.com/tutorial/SQLDeveloper/index.htm. Then review the following sections and associated demonstrations:
   a) What to Do First
   b) Working with Database Objects
   c) Accessing Data

Practice I-2: Getting Started

1) Start SQL Developer.
2) Create a database connection by using the following information (**Hint:** Select the Save Password check box):
   a) Connection Name: MyConnection
   b) Username: `ora41`
   c) Password: `ora41`
   d) Hostname: localhost
   e) Port: 1521

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Patel (oracle.ankur84@gmail.com) has a non-transferable license to use this Student Guide.

Practice I-2: Getting Started (continued)

f) SID: `orcl`

3) Test the new connection. If the Status is Success, connect to the database using this new connection.
   a) In the Database Connection window, click the Test button.
      Note: The connection status appears in the lower-left corner of the window.
   b) If the status is Success, click the Connect button.

4) Browse the structure of the `EMPLOYEES` table and display its data.
   a) Expand the MyConnection connection by clicking the plus symbol next to it.
   b) Expand the Tables icon by clicking the plus symbol next to it.
   c) Display the structure of the `EMPLOYEES` table.

5) Use the EMPLOYEES tab to view data in the `EMPLOYEES` table.

6) Use the SQL Worksheet to select the last names and salaries of all employees whose annual salary is greater than $10,000. Use both the Execute Statement (F9) and the Run Script (F5) icons to execute the `SELECT` statement. Review the results of both methods of executing the `SELECT` statements on the appropriate tabs.
   Note: Take a few minutes to familiarize yourself with the data, or consult Appendix B, which provides the description and data for all the tables in the HR schema that you will use in this course.

7) From the SQL Developer menu, select Tools > Preferences. The Preferences window appears.

8) Select Database > Worksheet Parameters. In the "Select default path to look for scripts" text box, use the Browse button to select the `/home/oracle/labs/plsf` folder. This folder contains the code example scripts, lab scripts, and practice solution scripts that are used in this course. Then, in the Preferences window, click OK to save the Worksheet Parameter setting.

9) Familiarize yourself with the structure of the `/home/oracle/labs/plsf` folder.
   a) Select File > Open. The Open window automatically selects the …`/plsf` folder as your starting location. This folder contains three subfolders:
      - The `/code_ex` folder contains the code examples found in the course materials. Each `.sql` script is associated with a particular page in the lesson.
      - The `/labs` folder contains the code that is used in certain lesson practices. You are instructed to run the required script in the appropriate practice.
      - The `/soln` folder contains the solutions for each practice. Each `.sql` script is numbered with the associated practice_exercise reference.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Anita M. Prajapati (prajapati.anita@gmail.com) has a non-transferable license to use this Student Guide.

Practice I-2: Getting Started (continued)

b) You can also use the Files tab to navigate through folders to open the script files.

c) Using the Open window, and the Files tab, navigate through the folders and open a script file without executing the code.

d) Close the SQL Worksheet.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

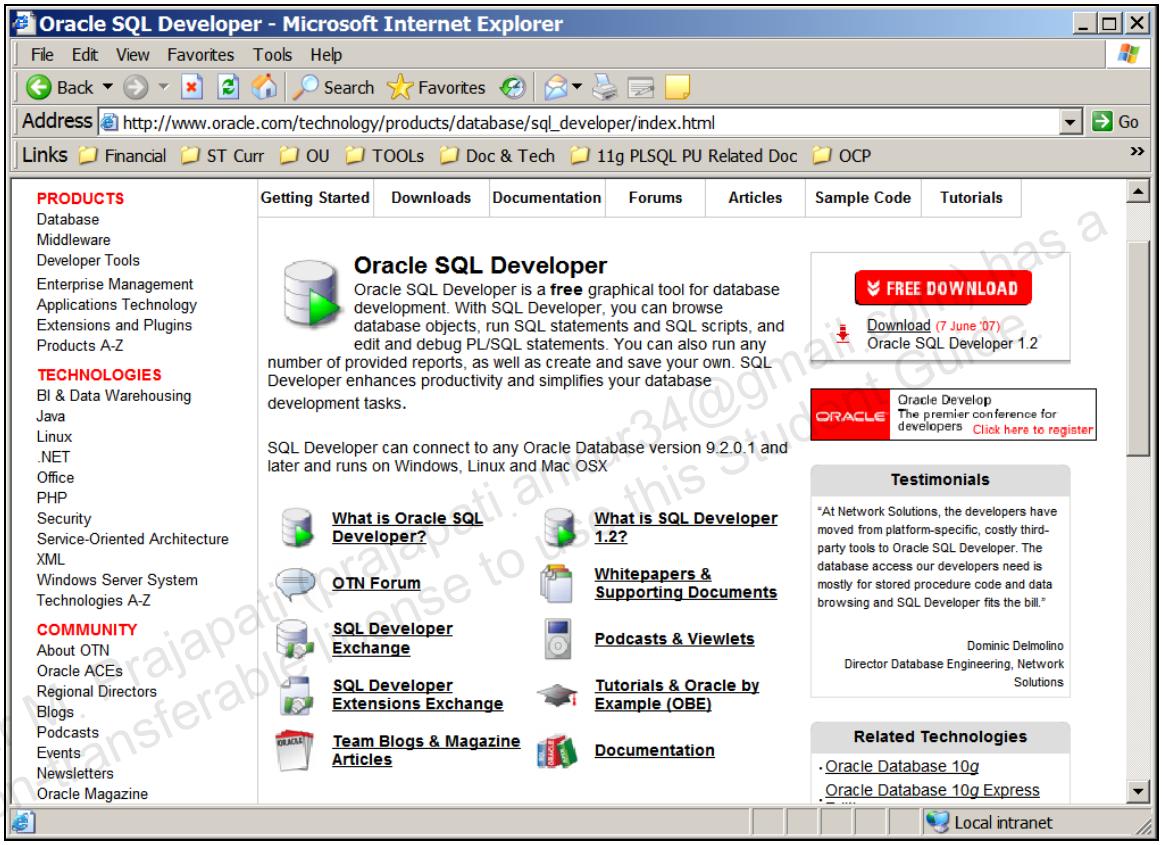
Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution I-1: Accessing SQL Developer Resources

1) Access the SQL Developer home page.

   a) Access the online SQL Developer Home Page, which is available at: http://www.oracle.com/technology/products/database/sql_developer/index.html

   The SQL Developer home page is displayed as follows:



   b) Bookmark the page for easier access in future.

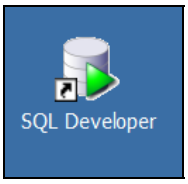
2) Access the SQL Developer tutorial, which is available online at http://st-curriculum.oracle.com/tutorial/SQLDeveloper/index.htm. Then review the following sections and associated demos:

   a) What to Do First
   b) Working with Database Objects
   c) Accessing Data

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution I-2: Getting Started

1) Start SQL Developer.
   Click the SQL Developer icon on your desktop.

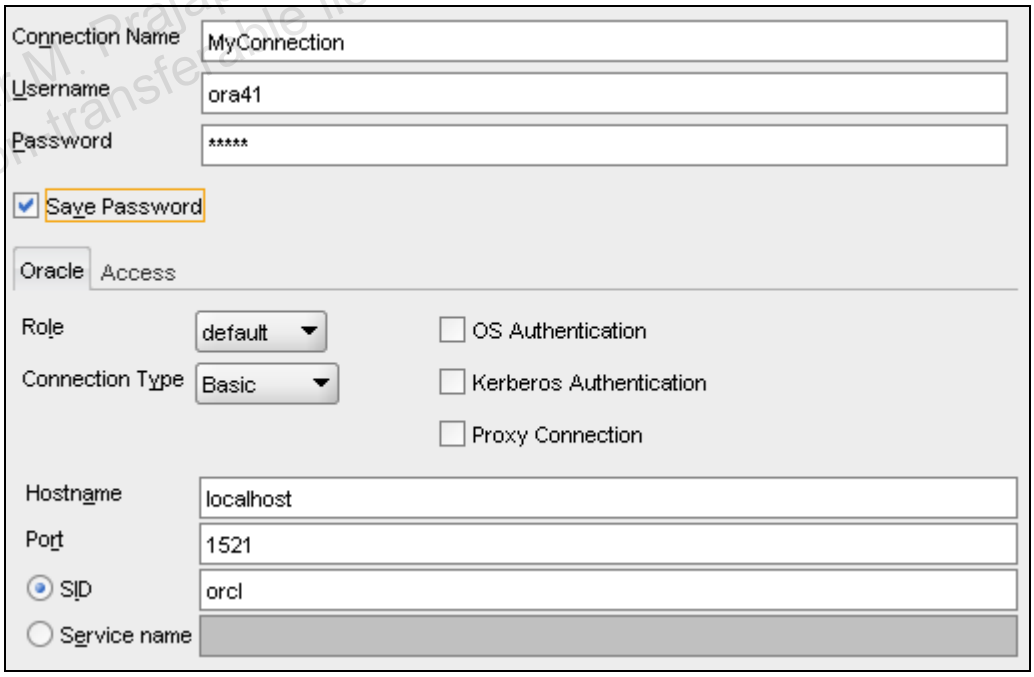


2) Create a database connection by using the following information (**Hint:** Select the Save Password check box):
   a) Connection Name: MyConnection
   b) Username: `ora41`
   c) Password: `ora41`
   d) Hostname: localhost
   e) Port: 1521
   f) SID: `orcl`

   Right-click the Connections node on the Connections tabbed page and select New Database Connection from the shortcut menu. Result: The New/Select Database Connection window appears.

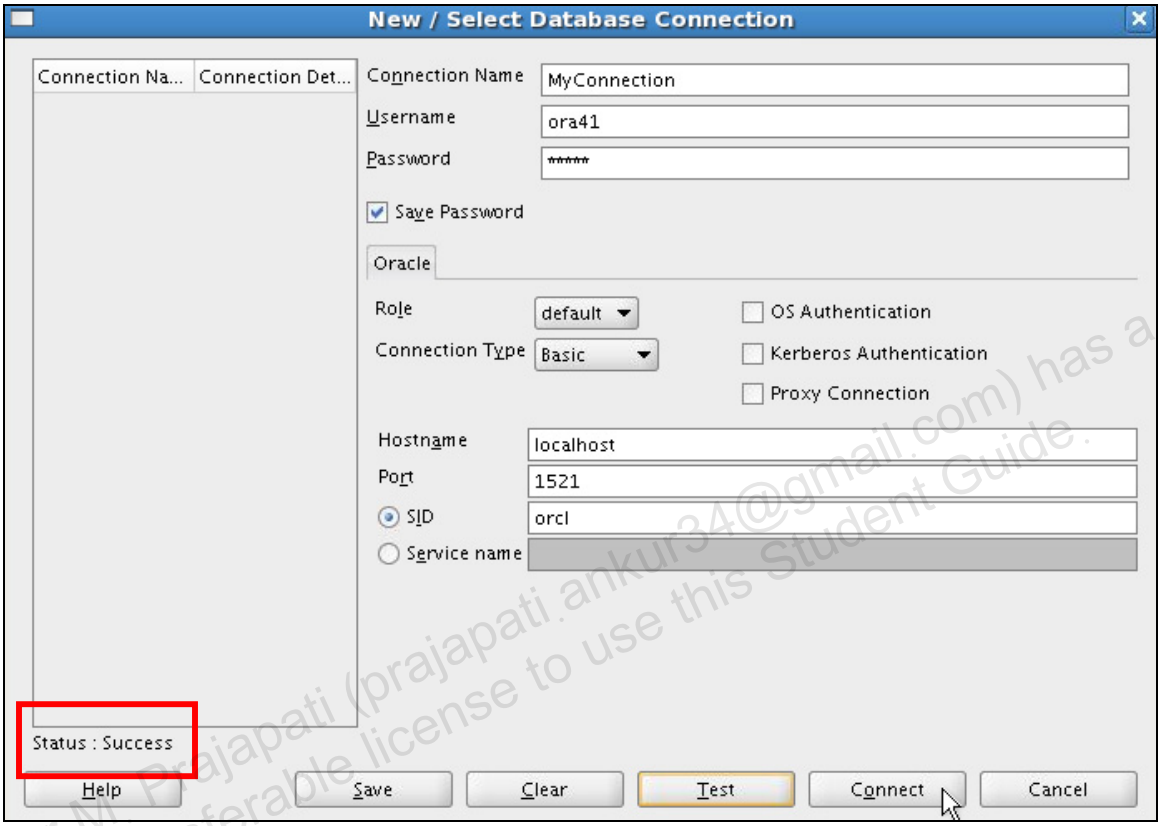
   Use the preceding information to create the new database connection. In addition, select the Save Password check box. For example:

Ankur M. Prajapati (prajapatiankur.34@gmail.com) has a non-transferable license to use this Student Guide.



Solution I-2: Getting Started (continued)

3) Test the new connection. If the Status is Success, connect to the database using this new connection.
   a) In the Database Connection window, click the Test button.
      Note: The connection status appears in the lower-left corner of the window.
   b) If the status is Success, click the Connect button.



Note: To display the properties of an existing connection, right-click the connection name on the Connections tab and select Properties from the shortcut menu.

4) Browse the structure of the `EMPLOYEES` table and display its data.
   a) Expand the MyConnection connection by clicking the plus symbol next to it.
   b) Expand Tables by clicking the plus symbol next to it.
   c) Display the structure of the `EMPLOYEES` table.

      Drill down on the `EMPLOYEES` table by clicking the plus symbol next to it.

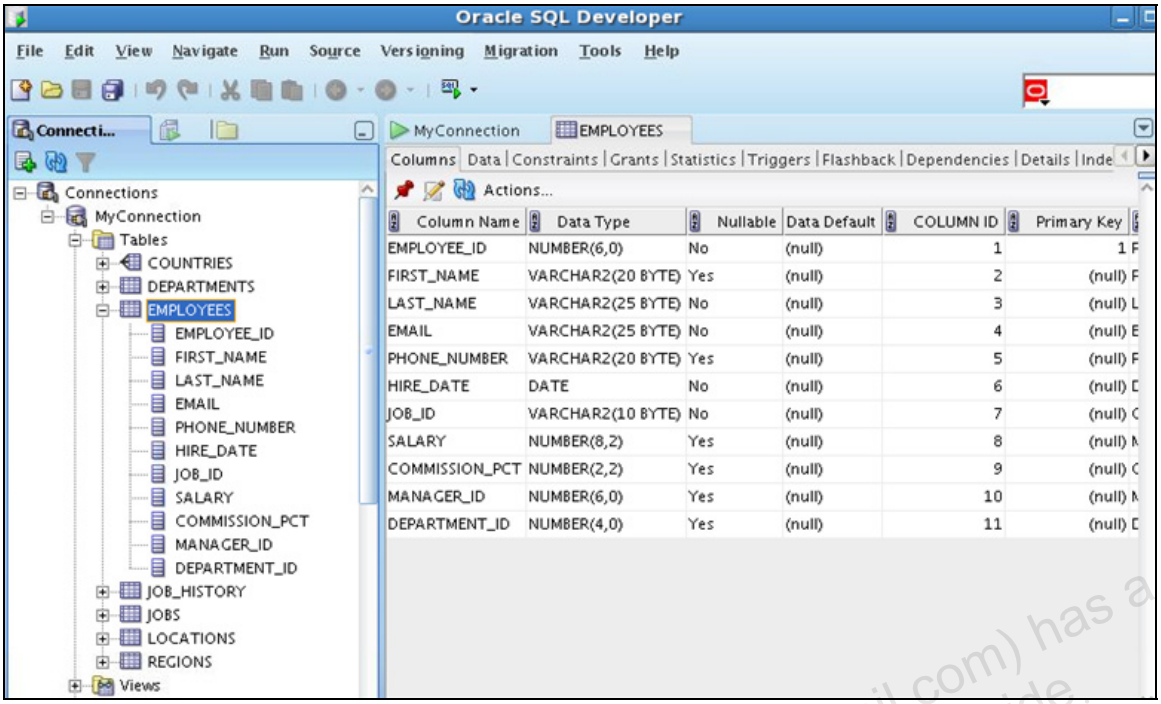
      Click the `EMPLOYEES` table.

      Result: The Columns tab displays the columns in the `EMPLOYEES` table as follows:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non transferable license to use this Student Guide.

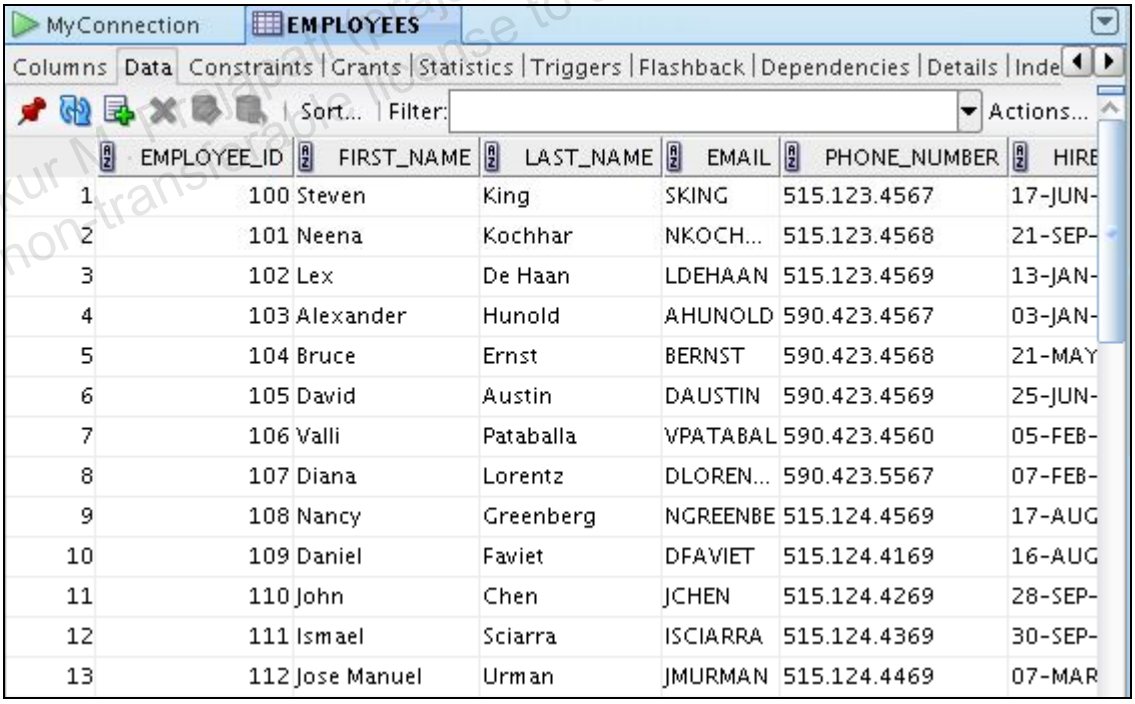
Solution I-2: Getting Started (continued)



5) Use the EMPLOYEES tab to view the data in the `EMPLOYEES` table.

To display employees' data, click the Data tab.

Result: The `EMPLOYEES` table data is displayed as follows:



	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE
1	100	Steven	King	SKING	515.123.4567	17-JUN-
2	101	Neena	Kochhar	NKOCH...	515.123.4568	21-SEP-
3	102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-
4	103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-
5	104	Bruce	Ernst	BERNST	590.423.4568	21-MAY
6	105	David	Austin	DAUSTIN	590.423.4569	25-JUN-
7	106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-
8	107	Diana	Lorentz	DLOREN...	590.423.5567	07-FEB-
9	108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG
10	109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG
11	110	John	Chen	JCHEN	515.124.4269	28-SEP-
12	111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-
13	112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution I-2: Getting Started (continued)

6) Use the SQL Worksheet to select the last names and salaries of all employees whose annual salary is greater than $10,000. Use both the Execute Statement (F9) and Run Script (F5) icons to execute the `SELECT` statement. Review the results of both methods of executing the `SELECT` statements on the appropriate tabs.

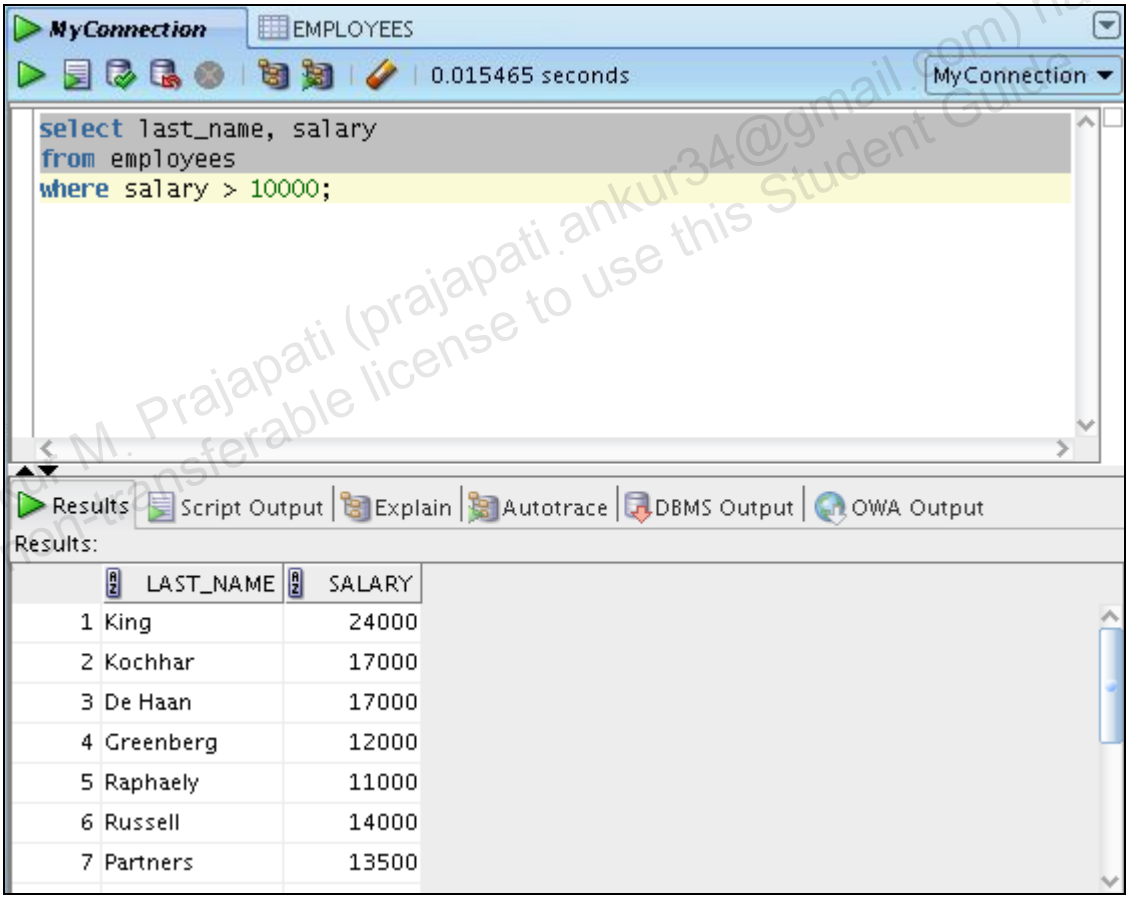
   Note: Take a few minutes to familiarize yourself with the data, or consult Appendix B, which provides the description and data for all the tables in the HR schema that you will use in this course.

   To display the SQL Worksheet, click the MyConnection tab.

   Note: This tab was opened previously when you drilled down on your database connection.

   Enter the appropriate `SELECT` statement. Press F9 to execute the query and F5 to execute the query using the Run Script method.

   For example, when you press F9, the results appear similar to the following:

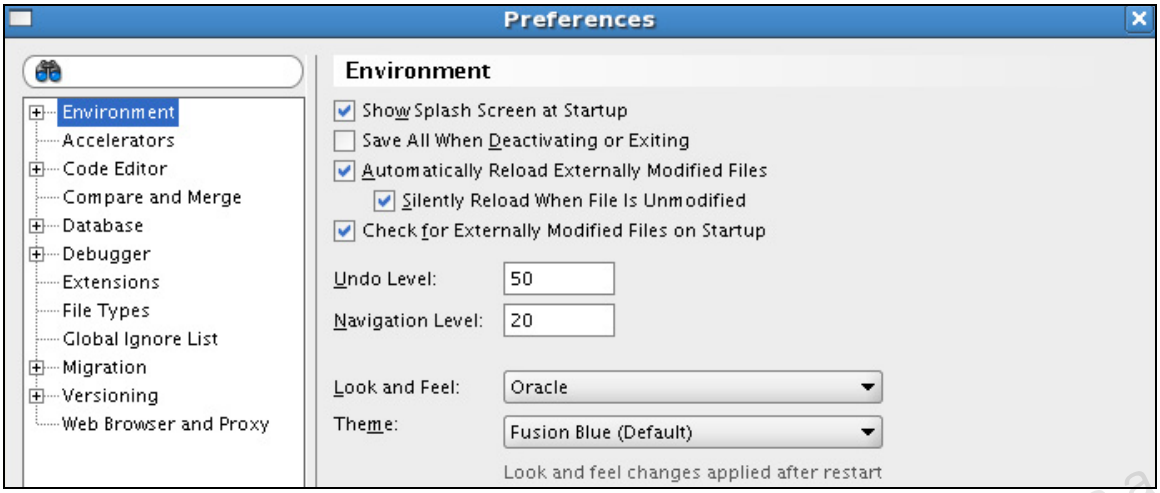


Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

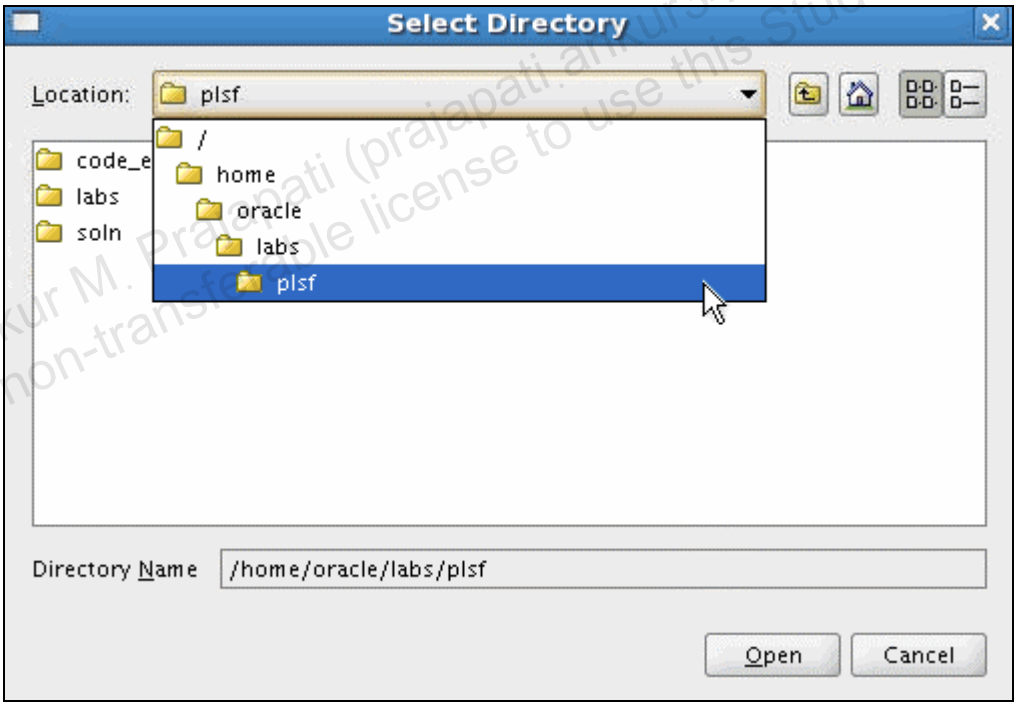
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide

Solution I-2: Getting Started (continued)

7) From the SQL Developer menu, select Tools > Preferences. The Preferences window appears.



8) Select Database > Worksheet Parameters. In the "Select default path to look for scripts" text box, use the Browse button to select the `/home/oracle/labs/plsf` folder.



This folder contains the code example scripts, lab scripts, and practice solution scripts that are used in this course.

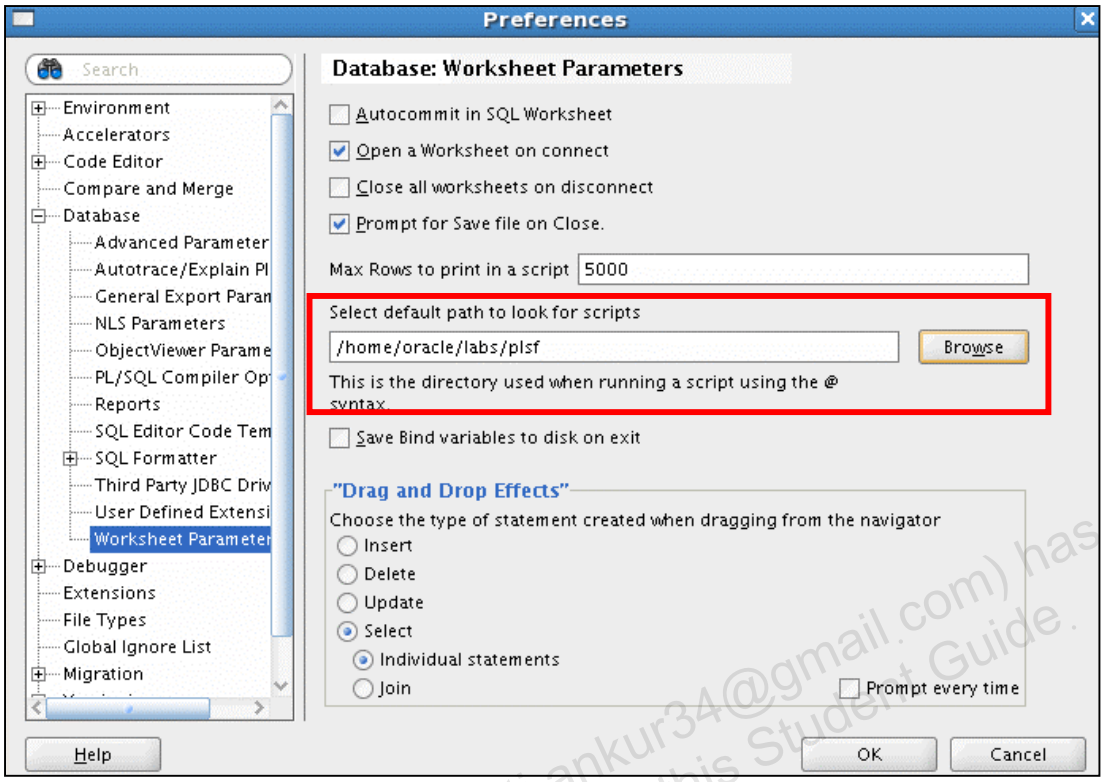
Click Open to select the folder.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide

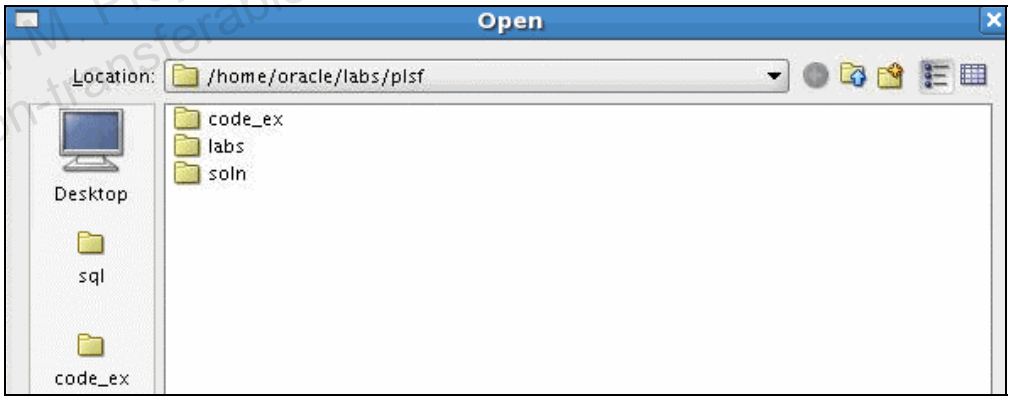
Solution I-2: Getting Started (continued)

Then, in the Preferences window, click OK to save the Worksheet Parameter setting.



9) Familiarize yourself with the structure of the `/home/oracle/labs/plsf` folder.

   a) Select File > Open. The Open window automatically selects the …`/plsf` folder as your starting location. This folder contains three subfolders:



- The `/code_ex` folder contains the code examples found in the course materials. Each `.sql` script is associated with a particular page in the lesson.
- The `/labs` folder contains the code that is used in certain lesson practices. You are instructed to run the required script in the appropriate practice.

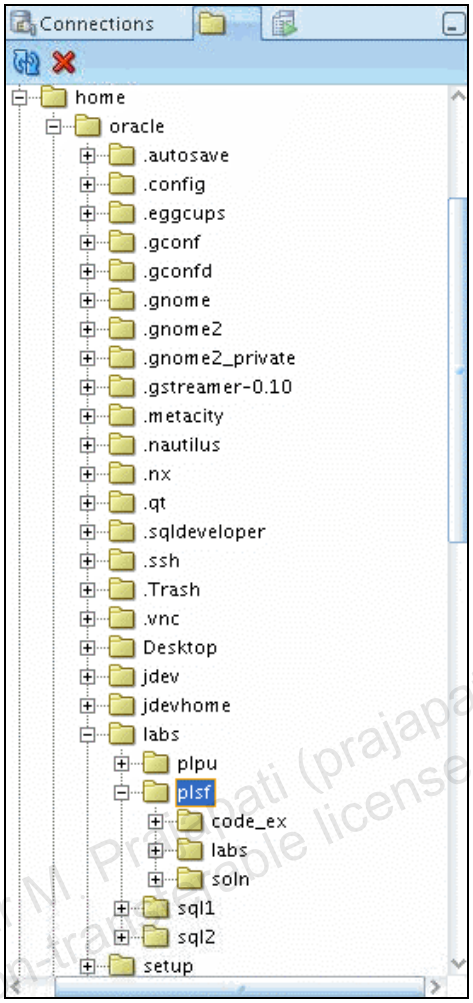
Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution I-2: Getting Started (continued)

- The `/soln` folder contains the solutions for each practice. Each `.sql` script is numbered with the associated practice_exercise reference.

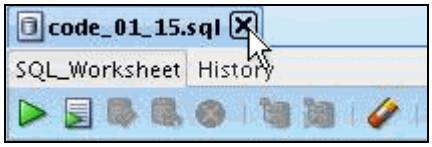
b) You can also use the Files tab to navigate through folders to open script files.



c) Using the Open window, and the Files tab, navigate through the folders and open a script file without executing the code.

d) Close the SQL Worksheet.

To close any SQL Worksheet tab, click X on the tab, as shown here:



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practices and Solutions for Lesson 1

The `/home/oracle/labs` folder is the working directory where you save the scripts that you create.
The solutions for all the practices are in the `/home/oracle/labs/plsf/soln` folder.

Practice 1: Introduction to PL/SQL

1) Which of the following PL/SQL blocks execute successfully?

   a)
   ```
   BEGIN
   END;
   ```

   b)
   ```
   DECLARE
   v_amount INTEGER(10);
   END;
   ```

   c)
   ```
   DECLARE
   BEGIN
   END;
   ```

   d)
   ```
   DECLARE
   v_amount INTEGER(10);
   BEGIN
   DBMS_OUTPUT.PUT_LINE(amount);
   END;
   ```

2) Create and execute a simple anonymous block that outputs “Hello World.” Execute and save this script as `lab_01_02_soln.sql`.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution 1: Introduction to PL/SQL

1) Which of the following PL/SQL blocks execute successfully?

a)
```
BEGIN
END;
```

b)
```
DECLARE
v_amount INTEGER(10);
END;
```

c)
```
DECLARE
BEGIN
END;
```

d)
```
DECLARE
v_amount INTEGER(10);
BEGIN
DBMS_OUTPUT.PUT_LINE(amount);
END;
```

The block in a does not execute. It has no executable statements.
The block in b does not have the mandatory executable section that starts with the `BEGIN` keyword.
The block in c has all the necessary parts, but no executable statements.
The block in d executes succesfully.

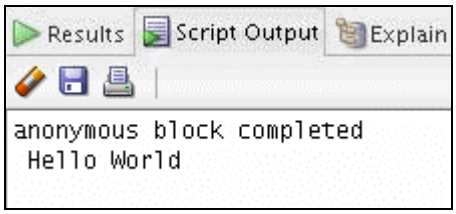
Anwar M. Prajapati (prajapati.anwar4@gmail.com) has a non-transferable license to use this Student Guide.

2) Create and execute a simple anonymous block that outputs "Hello World." Execute and save this script as `lab_01_02_soln.sql`.

Enter the following code in the workspace, and then press F5.

```
SET SERVEROUTPUT ON
BEGIN
DBMS_OUTPUT.PUT_LINE(' Hello World ');
END;
```

You should see the following output on the Script Output tab:



Click the Save button. Select the folder in which you want to save the file. Enter `lab_01_02_soln.sql` as the file name and click Save.

Practices and Solutions for Lesson 2

Practice 2: Declaring PL/SQL Variables

In this practice, you declare PL/SQL variables.

1) Identify valid and invalid identifiers:

   a) `today`
   b) `last_name`
   c) `today's_date`
   d) `Number_of_days_in_February_this_year`
   e) `Isleap$year`
   f) `#number`
   g) `NUMBER#`
   h) `number1to7`

2) Identify valid and invalid variable declaration and initialization:

   a) `number_of_copies PLS_INTEGER;`
   b) `PRINTER_NAME` constant `VARCHAR2(10);`
   c) `deliver_to VARCHAR2(10)`:=Johnson;
   d) `by_when DATE`:= `CURRENT_DATE`+1;

3) Examine the following anonymous block, and then select a statement from the following that is true.

```
DECLARE
  v_fname VARCHAR2(20);
  v_lname VARCHAR2(15) DEFAULT 'fernandez';
BEGIN
  DBMS_OUTPUT.PUT_LINE(v_fname ||' ' ||v_lname);
END;
```

   a) The block executes successfully and prints "fernandez."
   b) The block produces an error because the `fname` variable is used without initializing.
   c) The block executes successfully and prints "null fernandez."
   d) The block produces an error because you cannot use the `DEFAULT` keyword to initialize a variable of type `VARCHAR2`.
   e) The block produces an error because the `v_fname` variable is not declared.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Anupam Prajapati (prajapati.anupam@gmail.com) has a non-transferable license to use this Student Guide.

Practice 2: Declaring PL/SQL Variables (continued)

4) Modify an existing anonymous block and save it as a new script.

   a) Open the `lab_01_02_soln.sql` script, which you created in Practice 1.

   b) In this PL/SQL block, declare the following variables:

      1. `v_today` of type `DATE`. Initialize `today` with `SYSDATE`.
      2. `v_tomorrow` of type `today`. Use the `%TYPE` attribute to declare this variable.

   c) In the executable section:

      1. Initialize the `v_tomorrow` variable with an expression, which calculates tomorrow's date (add one to the value in `today`)
      2. Print the value of `v_today` and `tomorrow` after printing "Hello World"

   d) Save your script as `lab_02_04_soln.sql`, and then execute.

      The sample output is as follows (the values of `v_today` and `v_tomorrow` will be different to reflect your current today's and tomorrow's date):

```
anonymous block completed
 Hello World
TODAY IS : 05-JUN-09
TOMORROW IS : 06-JUN-09
```

5) Edit the `lab_02_04_soln.sql` script.

   a) Add code to create two bind variables, named `b_basic_percent` and `b_pf_percent`. Both bind variables are of type `NUMBER`.

   b) In the executable section of the PL/SQL block, assign the values 45 and 12 to `b_basic_percent` and `b_pf_percent`, respectively.

   c) Terminate the PL/SQL block with "/" and display the value of the bind variables by using the `PRINT` command.

   d) Execute and save your script as `lab_02_05_soln.sql`. The sample output is as follows:

```
anonymous block completed
b_basic_percent
--
45

b_pf_percent
--
12
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution 2: Declaring PL/SQL Variables

1) Identify valid and invalid identifiers:

   a) `today` **Valid**
   b) `last_name` **Valid**
   c) `today's_date` **Invalid** – character "`'`" not allowed
   d) `Number_of_days_in_February_this_year` **Invalid** – Too long
   e) `Isleap$year` **Valid**
   f) `#number` **Invalid –** Cannot start with "#"
   g) `NUMBER#` **Valid**
   h) `number1to7` **Valid**

2) Identify valid and invalid variable declaration and initialization:

   a) `number_of_copies PLS_INTEGER;` **Valid**
   b) `PRINTER_NAME` constant `VARCHAR2(10);` **Invalid**
   c) `deliver_to VARCHAR2(10)`:=Johnson; **Invalid**
   d) `by_when DATE`:= `CURRENT_DATE`+1; **Valid**

   *The declaration in **b** is invalid because constant variables must be initialized during declaration.*

   *The declaration in **c** is invalid because string literals should be enclosed within single quotation marks.*

3) Examine the following anonymous block, and then select a statement from the following that is true.

```
DECLARE
  v_fname VARCHAR2(20);
  v_lname VARCHAR2(15) DEFAULT 'fernandez';
BEGIN
  DBMS_OUTPUT.PUT_LINE(v_fname ||' ' ||v_lname);
END;
```

   a) The block executes successfully and prints "fernandez."
   b) The block produces an error because the `fname` variable is used without initializing.
   c) The block executes successfully and prints "null fernandez."
   d) The block produces an error because you cannot use the `DEFAULT` keyword to initialize a variable of type `VARCHAR2`.
   e) The block produces an error because the `v_fname` variable is not declared.

   a. The block will execute successfully and print "fernandez."

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 2: Declaring PL/SQL Variables (continued)

4) Modify an existing anonymous block and save it as a new script.

   a) Open the `lab_01_02_soln.sql` script, which you created in Practice 1.

   b) In the PL/SQL block, declare the following variables:

      1. Variable `v_today` of type `DATE`. Initialize `today` with `SYSDATE`.

```
DECLARE
   v_today DATE:=SYSDATE;
```

      2. Variable `v_tomorrow` of type `today`. Use the `%TYPE` attribute to declare this variable.

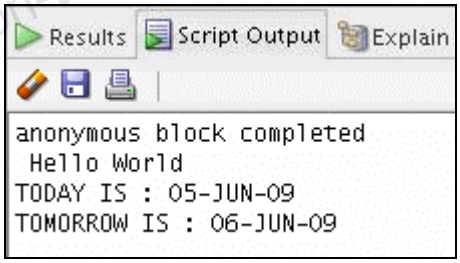
```
   v_tomorrow v_today%TYPE;
```

   c) In the executable section:
      1. Initialize the `v_tomorrow` variable with an expression, which calculates tomorrow's date (add one to the value in `v_today`)
      2. Print the value of `v_today` and `v_tomorrow` after printing "Hello World"

```
BEGIN
   v_tomorrow:=v_today +1;
   DBMS_OUTPUT.PUT_LINE(' Hello World ');
   DBMS_OUTPUT.PUT_LINE('TODAY IS : '||v_today);
   DBMS_OUTPUT.PUT_LINE('TOMORROW IS : ' || v_tomorrow);
END;
```

   d) Save your script as `lab_02_04_soln.sql`, and then execute.

      The sample output is as follows (the values of `v_today` and `v_tomorrow` will be different to reflect your current today's and tomorrow's date):



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 2: Declaring PL/SQL Variables (continued)

5) Edit the `lab_02_04_soln.sql` script.

   a) Add the code to create two bind variables, named `b_basic_percent` and `b_pf_percent`. Both bind variables are of type `NUMBER`.

```
VARIABLE b_basic_percent NUMBER
VARIABLE b_pf_percent NUMBER
```

   b) In the executable section of the PL/SQL block, assign the values `45` and `12` to `b_basic_percent` and `b_pf_percent`, respectively.

```
    :b_basic_percent:=45;
    :b_pf_percent:=12;
```

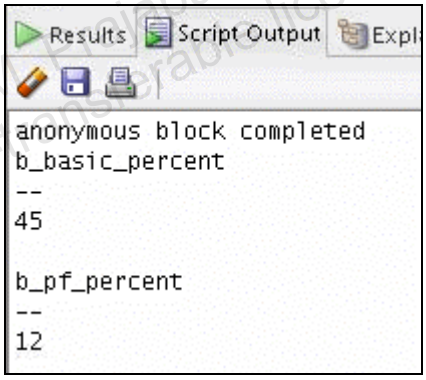
   c) Terminate the PL/SQL block with "/" and display the value of the bind variables by using the `PRINT` command.

```
/
PRINT b_basic_percent
PRINT b_pf_percent
```

OR

```
PRINT
```

   d) Execute and save your script as `lab_02_05_soln.sql`. The sample output is as follows:



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practices and Solutions for Lesson 3

Practice 3: Writing Executable Statements

In this practice, you examine and write executable statements.

```
DECLARE
 v_weight    NUMBER(3) := 600;
 v_message   VARCHAR2(255) := 'Product 10012';
BEGIN
  DECLARE
   v_weight    NUMBER(3) := 1;
   v_message   VARCHAR2(255) := 'Product 11001';
   v_new_locn  VARCHAR2(50) := 'Europe';
  BEGIN
   v_weight := v_weight + 1;
   v_new_locn := 'Western ' || v_new_locn;
1 →
  END;
 v_weight := v_weight + 1;
 v_message := v_message || ' is in stock';
 v_new_locn := 'Western ' || v_new_locn;
2 →
END;
/
```

1) Evaluate the preceding PL/SQL block and determine the data type and value of each of the following variables, according to the rules of scoping.

   a) The value of `v_weight` at position 1 is:

   b) The value of `v_new_locn` at position 1 is:

   c) The value of `v_weight` at position 2 is:

   d) The value of `v_message` at position 2 is:

   e) The value of `v_new_locn` at position 2 is:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practice 3: Writing Executable Statements (continued)

```
DECLARE
   v_customer     VARCHAR2(50) := 'Womansport';
   v_credit_rating   VARCHAR2(50) := 'EXCELLENT';
BEGIN
    DECLARE
       v_customer  NUMBER(7) := 201;
       v_name VARCHAR2(25) := 'Unisports';
    BEGIN
       v_credit_rating :='GOOD';
       …
    END;
  …
END;
```

2) In the preceding PL/SQL block, determine the values and data types for each of the following cases:

   a) The value of `v_customer` in the nested block is:

   b) The value of `v_name` in the nested block is:

   c) The value of `v_credit_rating` in the nested block is:

   d) The value of `v_customer` in the main block is:

   e) The value of `v_name` in the main block is:

   f) The value of `v_credit_rating` in the main block is:

3) Use the same session that you used to execute the practices in the lesson titled "Declaring PL/SQL Variables." If you have opened a new session, execute `lab_02_05_soln.sql`. Then, edit `lab_02_05_soln.sql` as follows:

   a) Use single-line comment syntax to comment the lines that create the bind variables, and turn on `SERVEROUTPUT`.

   b) Use multiple-line comments in the executable section to comment the lines that assign values to the bind variables.

   c) In the declaration section:
      1. Declare and initialize two temporary variables to replace the commented out bind variables
      2. Declare two additional variables: `v_fname` of type `VARCHAR2` and size 15, and `v_emp_sal` of type `NUMBER` and size 10

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practice 3: Writing Executable Statements (continued)

d) Include the following SQL statement in the executable section:

```
SELECT first_name, salary INTO v_fname, v_emp_sal
  FROM employees WHERE employee_id=110;
```

e) Change the line that prints "Hello World" to print "Hello" and the first name. Then, comment the lines that display the dates and print the bind variables.

f) Calculate the contribution of the employee towards provident fund (PF). PF is 12% of the basic salary, and the basic salary is 45% of the salary. Use local variables for the calculation. Try to use only one expression to calculate the PF. Print the employee's salary and his or her contribution toward PF.

g) Execute and save your script as `lab_03_03_soln.sql`. The sample output is as follows:

```
anonymous block completed
 Hello John
YOUR SALARY IS : 8200
YOUR CONTRIBUTION TOWARDS PF:
     442.8
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 3: Writing Executable Statements

In this practice, you examine and write executable statements.

```
DECLARE
 v_weight    NUMBER(3) := 600;
 v_message   VARCHAR2(255) := 'Product 10012';
BEGIN
  DECLARE
   v_weight    NUMBER(3) := 1;
   v_message   VARCHAR2(255) := 'Product 11001';
   v_new_locn  VARCHAR2(50) := 'Europe';
  BEGIN
   v_weight := v_weight + 1;
   v_new_locn := 'Western ' || v_new_locn;
1 →
  END;
 v_weight := v_weight + 1;
 v_message := v_message || ' is in stock';
 v_new_locn := 'Western ' || v_new_locn;
2 →
END;
/
```

1) Evaluate the preceding PL/SQL block and determine the data type and value of each of the following variables, according to the rules of scoping.

   a) The value of `v_weight` at position 1 is:
      2
      The data type is `NUMBER`.

   b) The value of `v_new_locn` at position 1 is:
      Western Europe
      The data type is `VARCHAR2`.

   c) The value of `v_weight` at position 2 is:
      601
      The data type is `NUMBER`.

   d) The value of `v_message` at position 2 is:
      Product 10012 is in stock
      The data type is `VARCHAR2`.

   e) The value of `v_new_locn` at position 2 is:
      Illegal because `v_new_locn` is not visible outside the subblock

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 3: Writing Executable Statements (continued)

```
DECLARE
   v_customer     VARCHAR2(50) := 'Womansport';
   v_credit_rating   VARCHAR2(50) := 'EXCELLENT';
BEGIN
    DECLARE
       v_customer  NUMBER(7) := 201;
       v_name VARCHAR2(25) := 'Unisports';
    BEGIN
       v_credit_rating :='GOOD';
       …
    END;
  …
END;
```

2) In the preceding PL/SQL block, determine the values and data types for each of the following cases:

   a) The value of `v_customer` in the nested block is:
      201
      The data type is `NUMBER`.
   b) The value of `v_name` in the nested block is:
      Unisports
      The data type is `VARCHAR2`.
   c) The value of `v_credit_rating` in the nested block is:
      GOOD
      The data type is `VARCHAR2`.
   d) The value of `v_customer` in the main block is:
      Womansport
      The data type is `VARCHAR2`.
   e) The value of `v_name` in the main block is:
      Null. **`name` is not visible in the main block and you would see an error**.
   f) The value of `v_credit_rating` in the main block is:
      EXCELLENT
      The data type is `VARCHAR2`.

3) Use the same session that you used to execute the practices in the lesson titled "Declaring PL/SQL Variables." If you have opened a new session, execute `lab_02_05_soln.sql`. Then, edit `lab_02_05_soln.sql` as follows:

   a) Use single-line comment syntax to comment the lines that create the bind variables, and turn on `SERVEROUTPUT`.

```
-- VARIABLE b_basic_percent NUMBER
-- VARIABLE b_pf_percent NUMBER
SET SERVEROUTPUT ON
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 3: Writing Executable Statements (continued)

b) Use multiple-line comments in the executable section to comment the lines that assign values to the bind variables.

```
   /*:b_basic_percent:=45;
   :b_pf_percent:=12;*/
```

c) In the declaration section:
   1. Declare and initialize two temporary variables to replace the commented out bind variables
   2. Declare two additional variables: `v_fname` of type `VARCHAR2` and size 15, and `v_emp_sal` of type `NUMBER` and size 10

```
DECLARE
   v_basic_percent NUMBER:=45;
   v_pf_percent NUMBER:=12;
   v_fname VARCHAR2(15);
   v_emp_sal NUMBER(10);
```

d) Include the following SQL statement in the executable section:

```
   SELECT first_name, salary INTO v_fname, v_emp_sal
     FROM employees WHERE employee_id=110;
```

e) Change the line that prints "Hello World" to print "Hello" and the first name. Then, comment the lines that display the dates and print the bind variables.

```
DBMS_OUTPUT.PUT_LINE(' Hello '|| v_fname);
/*   DBMS_OUTPUT.PUT_LINE('TODAY IS : '|| v_today);
DBMS_OUTPUT.PUT_LINE('TOMORROW IS : ' || v_tomorrow);*/
...
...

/
--PRINT b_basic_percent
--PRINT b_basic_percent
```

f) Calculate the contribution of the employee towards provident fund (PF). PF is 12% of the basic salary, and the basic salary is 45% of the salary. Use local variables for the calculation. Try to use only one expression to calculate the PF. Print the employee's salary and his or her contribution toward PF.

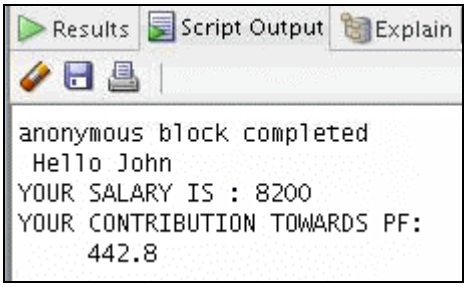
```
   DBMS_OUTPUT.PUT_LINE('YOUR SALARY IS : '||v_emp_sal);
   DBMS_OUTPUT.PUT_LINE('YOUR CONTRIBUTION TOWARDS PF:
     '||v_emp_sal*v_basic_percent/100*v_pf_percent/100);
END;
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 3: Writing Executable Statements (continued)

g) Execute and save your script as `lab_03_03_soln.sql`. The sample output is as follows:



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practices and Solutions for Lesson 4

Practice 4: Interacting with the Oracle Server

In this practice, you use PL/SQL code to interact with the Oracle Server.

1) Create a PL/SQL block that selects the maximum department ID in the `departments` table and stores it in the `v_max_deptno` variable. Display the maximum department ID.

   a) Declare a variable `v_max_deptno` of type `NUMBER` in the declarative section.

   b) Start the executable section with the `BEGIN` keyword and include a `SELECT` statement to retrieve the maximum `department_id` from the `departments` table.

   c) Display `v_max_deptno` and end the executable block.

   d) Execute and save your script as `lab_04_01_soln.sql`. The sample output is as follows:

```
anonymous block completed
The maximum department_id is : 270
```

2) Modify the PL/SQL block that you created in step 1 to insert a new department into the `departments` table.

   a) Load the `lab_04_01_soln.sql` script. Declare two variables: `v_dept_name` of type `departments.department_name` and `v_dept_id` of type `NUMBER`
   Assign 'Education' to `v_dept_name` in the declarative section.

   b) You have already retrieved the current maximum department number from the `departments` table. Add 10 to it and assign the result to `v_dept_id`.

   c) Include an `INSERT` statement to insert data into the `department_name`, `department_id`, and `location_id` columns of the `departments` table. Use values in `dept_name` and `dept_id` for `department_name` and `department_id`, respectively, and use `NULL` for `location_id`.

   d) Use the SQL attribute `SQL%ROWCOUNT` to display the number of rows that are affected.

   e) Execute a `SELECT` statement to check whether the new department is inserted. You can terminate the PL/SQL block with "/" and include the `SELECT` statement in your script.

   f) Execute and save your script as `lab_04_02_soln.sql`. The sample output is as follows:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practice 4: Interacting with the Oracle Server (continued)

```
anonymous block completed
The maximum department_id is : 270
 SQL%ROWCOUNT gives 1

DEPARTMENT_ID          DEPARTMENT_NAME                MANAGER_ID             LOCATION_ID
---------------------- ------------------------------ ---------------------- -----------
280                    Education

1 rows selected
```

3) In step 2, you set `location_id` to `NULL`. Create a PL/SQL block that updates the `location_id` to `3000` for the new department.
Note: If you successfully completed step 2, continue with step 3a. If not, first execute the solution script `/soln/sol_04_02.sql`.

   a) Start the executable block with the `BEGIN` keyword. Include the `UPDATE` statement to set the `location_id` to `3000` for the new department (`dept_id` =280).

   b) End the executable block with the `END` keyword. Terminate the PL/SQL block with "/" and include a `SELECT` statement to display the department that you updated.

   c) Include a `DELETE` statement to delete the department that you added.

   d) Execute and save your script as `lab_04_03_soln.sql`. The sample output is as follows:

```
anonymous block completed
DEPARTMENT_ID          DEPARTMENT_NAME                MANAGER_ID             LOCATION_ID
---------------------- ------------------------------ ---------------------- -----------
280                    Education                                             3000

1 rows selected

1 rows deleted
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution 4: Interacting with the Oracle Server

In this practice, you use PL/SQL code to interact with the Oracle Server.

1) Create a PL/SQL block that selects the maximum department ID in the `departments` table and stores it in the `v_max_deptno` variable. Display the maximum department ID.

   a) Declare a variable `v_max_deptno` of type `NUMBER` in the declarative section.

```
DECLARE
   v_max_deptno  NUMBER;
```

   b) Start the executable section with the `BEGIN` keyword and include a `SELECT` statement to retrieve the maximum `department_id` from the `departments` table.

```
BEGIN
   SELECT MAX(department_id)  INTO v_max_deptno  FROM
      departments;
```

   c) Display `v_max_deptno` and end the executable block.

```
DBMS_OUTPUT.PUT_LINE('The maximum department_id is : ' ||
v_max_deptno);
END;
```

   d) Execute and save your script as `lab_04_01_soln.sql`. The sample output is as follows:

```
anonymous block completed
The maximum department_id is : 270
```

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

2) Modify the PL/SQL block that you created in step 1 to insert a new department into the `departments` table.

   a) Load the `lab_04_01_soln.sql` script. Declare two variables:
   `v_dept_name` of type `departments.department_name` and
   `v_dept_id` of type `NUMBER`
   Assign ‘Education’ to `v_dept_name` in the declarative section.

```
v_dept_name departments.department_name%TYPE:= 'Education';
v_dept_id NUMBER;
```

Solution 4: Interacting with the Oracle Server (continued)

b) You have already retrieved the current maximum department number from the `departments` table. Add 10 to it and assign the result to `v_dept_id`.

```
v_dept_id := 10 + v_max_deptno;
```

c) Include an `INSERT` statement to insert data into the `department_name`, `department_id`, and `location_id` columns of the `departments` table. Use values in `dept_name` and `dept_id` for `department_name` and `department_id`, respectively, and use `NULL` for `location_id`.

```
…
INSERT INTO departments (department_id, department_name,
location_id)
VALUES (v_dept_id, v_dept_name, NULL);
```

d) Use the SQL attribute `SQL%ROWCOUNT` to display the number of rows that are affected.

```
DBMS_OUTPUT.PUT_LINE (' SQL%ROWCOUNT gives ' || SQL%ROWCOUNT);
…
```

e) Execute a `SELECT` statement to check whether the new department is inserted. You can terminate the PL/SQL block with "/" and include the `SELECT` statement in your script.

```
…
/
SELECT * FROM  departments  WHERE  department_id= 280;
```

f) Execute and save your script as `lab_04_02_soln.sql`. The sample output is as follows:

```
anonymous block completed
The maximum department_id is : 270
 SQL%ROWCOUNT gives 1

DEPARTMENT_ID          DEPARTMENT_NAME                MANAGER_ID             LOCATION_ID
---------------------- ------------------------------ ---------------------- -----------
280                    Education

1 rows selected
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 4: Interacting with the Oracle Server (continued)

3) In step 2, you set `location_id` to `NULL`. Create a PL/SQL block that updates the `location_id` to `3000` for the new department.
Note: If you successfully completed step 2, continue with step 3a. If not, first execute the solution script `/soln/sol_04_02.sql`.

   a) Start the executable block with the `BEGIN` keyword. Include the `UPDATE` statement to set `location_id` to `3000` for the new department (`dept_id` =280).

```
BEGIN
UPDATE departments SET location_id=3000 WHERE
department_id=280;
```

   b) End the executable block with the `END` keyword. Terminate the PL/SQL block with "/" and include a `SELECT` statement to display the department that you updated.

```
END;
/
SELECT * FROM departments WHERE department_id=280;
```

   c) Include a `DELETE` statement to delete the department that you added.

```
DELETE FROM departments WHERE department_id=280;
```

   d) Execute and save your script as `lab_04_03_soln.sql`. The sample output is as follows:

```
anonymous block completed
DEPARTMENT_ID          DEPARTMENT_NAME                MANAGER_ID             LOCATION_ID
---------------------- ------------------------------ ---------------------- -----------
280                    Education                                             3000

1 rows selected

1 rows deleted
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

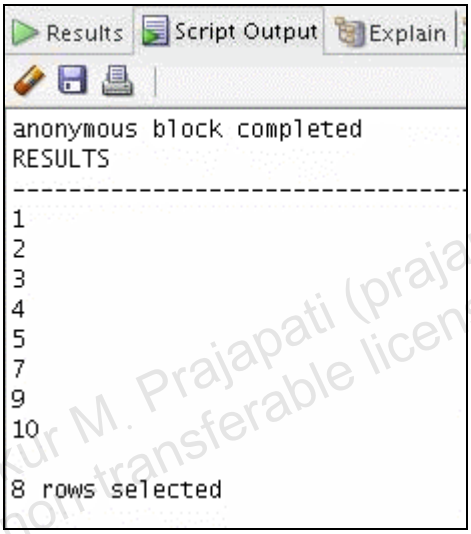
Practices and Solutions for Lesson 5

Practice 5: Writing Control Structures

In this practice, you create PL/SQL blocks that incorporate loops and conditional control structures. This practice tests your understanding of various `IF` statements and `LOOP` constructs.

1) Execute the command in the `lab_05_01.sql` file to create the `messages` table. Write a PL/SQL block to insert numbers into the `messages` table.
   a) Insert the numbers 1 through 10, excluding 6 and 8.
   b) Commit before the end of the block.
   c) Execute a `SELECT` statement to verify that your PL/SQL block worked.

   Result: You should see the following output:



2) Execute the `lab_05_02.sql` script. This script creates an `emp` table that is a replica of the `employees` table. It alters the `emp` table to add a new column, `stars`, of `VARCHAR2` data type and size 50. Create a PL/SQL block that inserts an asterisk in the `stars` column for every `$1000` of an employee's salary. Save your script as `lab_05_02_soln.sql`.
   a) In the declarative section of the block, declare a variable `v_empno` of type `emp.employee_id` and initialize it to 176. Declare a variable `v_asterisk` of type `emp.stars` and initialize it to `NULL`. Create a variable `v_sal` of type `emp.salary`.
   b) In the executable section, write logic to append an asterisk (`*`) to the string for every $1,000 of the salary. For example, if the employee earns $8,000, the string

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practice 5: Writing Control Structures (continued)

of asterisks should contain eight asterisks. If the employee earns $12,500, the string of asterisks should contain 13 asterisks.

c) Update the `stars` column for the employee with the string of asterisks. Commit before the end of the block.

d) Display the row from the `emp` table to verify whether your PL/SQL block has executed successfully.

e) Execute and save your script as `lab_05_02_soln.sql`. The output is as follows:

```
Results | Script Output | Explain | Autotrace | DBMS Output

anonymous block completed
EMPLOYEE_ID          SALARY               STARS
-------------------- -------------------- --------------
176                  8600                 *********

1 rows selected
```

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution 5: Writing Control Structures

1) Execute the command in the `lab_05_01.sql` file to create the `messages` table. Write a PL/SQL block to insert numbers into the `messages` table.

   a) Insert the numbers 1 through 10, excluding 6 and 8.

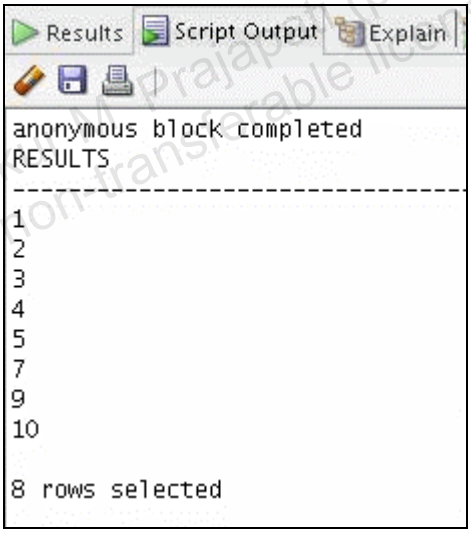
   b) Commit before the end of the block.

```
BEGIN
FOR i in 1..10 LOOP
  IF i = 6 or i = 8 THEN
    null;
  ELSE
    INSERT INTO messages(results)
    VALUES (i);
 END IF;
END LOOP;
COMMIT;
END;
/
```

   c) Execute a `SELECT` statement to verify that your PL/SQL block worked.

```
SELECT * FROM messages;
```

Result: You should see the following output:



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 5: Writing Control Structures (continued)

2) Execute the `lab_05_02.sql` script. This script creates an `emp` table that is a replica of the `employees` table. It alters the `emp` table to add a new column, `stars`, of `VARCHAR2` data type and size `50`. Create a PL/SQL block that inserts an asterisk in the `stars` column for every `$1000` of the employee's salary. Save your script as `lab_05_02_soln.sql`.

   a) In the declarative section of the block, declare a variable `v_empno` of type `emp.employee_id` and initialize it to 176. Declare a variable `v_asterisk` of type `emp.stars` and initialize it to `NULL`. Create a variable `v_sal` of type `emp.salary`.

```
DECLARE
  v_empno        emp.employee_id%TYPE := 176;
  v_asterisk     emp.stars%TYPE := NULL;
  v_sal           emp.salary%TYPE;
```

   b) In the executable section, write logic to append an asterisk (`*`) to the string for every $1,000 of the salary. For example, if the employee earns $8,000, the string of asterisks should contain eight asterisks. If the employee earns $12,500, the string of asterisks should contain 13 asterisks.

```
BEGIN
    SELECT NVL(ROUND(salary/1000), 0)  INTO v_sal
    FROM emp  WHERE employee_id = v_empno;

    FOR i IN 1..v_sal
       LOOP
        v_asterisk := v_asterisk ||'*';
    END LOOP;
```

   c) Update the `stars` column for the employee with the string of asterisks. Commit before the end of the block.

```
   UPDATE emp SET stars = v_asterisk
   WHERE employee_id = v_empno;
   COMMIT;
END;
/
```

   d) Display the row from the `emp` table to verify whether your PL/SQL block has executed successfully.

```
SELECT employee_id,salary, stars
FROM emp WHERE employee_id =176;
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 5: Writing Control Structures (continued)

e) Execute and save your script as `lab_05_02_soln.sql`. The output is as follows:

```
Results | Script Output | Explain | Autotrace | DBMS Output

anonymous block completed
EMPLOYEE_ID          SALARY               STARS
-------------------- -------------------- --------------
176                  8600                 *********

1 rows selected
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practices and Solutions for Lesson 6

Practice 6: Working with Composite Data Types

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

1) Write a PL/SQL block to print information about a given country.
   a) Declare a PL/SQL record based on the structure of the `countries` table.
   b) Declare a variable `v_countryid`. Assign `CA` to `v_countryid`.
   c) In the declarative section, use the `%ROWTYPE` attribute and declare the `v_country_record` variable of type `countries`.
   d) In the executable section, get all the information from the `countries` table by using `v_countryid`. Display selected information about the country. The sample output is as follows:

```
anonymous block completed
Country Id: CA Country Name: Canada Region: 2
```

   e) You may want to execute and test the PL/SQL block for countries with the IDs DE, UK, and US.

Ankurbhai Prajapati (prajapatiankur24@gmail.com) has a non-transferable license to use this Student Guide.

2) Create a PL/SQL block to retrieve the names of some departments from the `departments` table and print each department name on the screen, incorporating an associative array. Save the script as `lab_06_02_soln.sql`.
   a) Declare an `INDEX BY` table `dept_table_type` of type `departments.department_name`. Declare a variable `my_dept_table` of type `dept_table_type` to temporarily store the names of the departments.
   b) Declare two variables: `f_loop_count` and `v_deptno` of type `NUMBER`. Assign 10 to `f_loop_count` and 0 to `v_deptno`.
   c) Using a loop, retrieve the names of 10 departments and store the names in the associative array. Start with `department_id` 10. Increase `v_deptno` by 10 for every loop iteration. The following table shows the `department_id` for which you should retrieve the `department_name`.

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
30	Purchasing
40	Human Resources
50	Shipping
60	IT
70	Public Relations
80	Sales
90	Executive
100	Finance

Practice 6: Working with Composite Data Types (continued)

d) Using another loop, retrieve the department names from the associative array and display them.
e) Execute and save your script as `lab_06_02_soln.sql`. The output is as follows:

```
anonymous block completed
Administration
Marketing
Purchasing
Human Resources
Shipping
IT
Public Relations
Sales
Executive
Finance
```

3) Modify the block that you created in Practice 2 to retrieve all information about each department from the `departments` table and display the information. Use an associative array with the `INDEX BY` table of records method.

a) Load the `lab_06_02_soln.sql` script.
b) You have declared the associative array to be of type `departments.department_name`. Modify the declaration of the associative array to temporarily store the number, name, and location of all the departments. Use the `%ROWTYPE` attribute.
c) Modify the `SELECT` statement to retrieve all department information currently in the `departments` table and store it in the associative array.
d) Using another loop, retrieve the department information from the associative array and display the information.

The sample output is as follows:

```
anonymous block completed
Department Number: 10 Department Name: Administration Manager Id: 200 Location Id: 1700
Department Number: 20 Department Name: Marketing Manager Id: 201 Location Id: 1800
Department Number: 30 Department Name: Purchasing Manager Id: 114 Location Id: 1700
Department Number: 40 Department Name: Human Resources Manager Id: 203 Location Id: 2400
Department Number: 50 Department Name: Shipping Manager Id: 121 Location Id: 1500
Department Number: 60 Department Name: IT Manager Id: 103 Location Id: 1400
Department Number: 70 Department Name: Public Relations Manager Id: 204 Location Id: 2700
Department Number: 80 Department Name: Sales Manager Id: 145 Location Id: 2500
Department Number: 90 Department Name: Executive Manager Id: 100 Location Id: 1700
Department Number: 100 Department Name: Finance Manager Id: 108 Location Id: 1700
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Patel (placati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution 6: Working with Composite Data Types

1) Write a PL/SQL block to print information about a given country.

   a) Declare a PL/SQL record based on the structure of the `countries` table.
   b) Declare a variable `v_countryid`. Assign `CA` to `v_countryid`.

```
SET SERVEROUTPUT ON

SET VERIFY OFF
DECLARE
  v_countryid varchar2(20):= 'CA';
```

   c) In the declarative section, use the `%ROWTYPE` attribute and declare the `v_country_record` variable of type `countries`.

```
 v_country_record countries%ROWTYPE;
```

   d) In the executable section, get all the information from the `countries` table by using `v_countryid`. Display selected information about the country. The sample output is as follows:

```
BEGIN
 SELECT *
 INTO   v_country_record
 FROM   countries
 WHERE country_id = UPPER(v_countryid);

 DBMS_OUTPUT.PUT_LINE ('Country Id: ' ||
     v_country_record.country_id ||
     ' Country Name: ' || v_country_record.country_name
     || ' Region: ' || v_country_record.region_id);

END;
```

```
anonymous block completed
Country Id: CA Country Name: Canada Region: 2
```

   e) You may want to execute and test the PL/SQL block for countries with the IDs DE, UK, and US.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 6: Working with Composite Data Types (continued)

2) Create a PL/SQL block to retrieve the names of some departments from the `departments` table and print each department name on the screen, incorporating an associative array. Save the script as `lab_06_02_soln.sql`.

   a) Declare an `INDEX BY` table `dept_table_type` of type `departments.department_name`. Declare a variable `my_dept_table` of type `dept_table_type` to temporarily store the names of the departments.

```
SET SERVEROUTPUT ON

DECLARE
   TYPE dept_table_type is table of
     departments.department_name%TYPE
   INDEX BY PLS_INTEGER;
   my_dept_table   dept_table_type;
```

   b) Declare two variables: `f_loop_count` and `v_deptno` of type `NUMBER`. Assign 10 to `f_loop_count` and 0 to `v_deptno`.

```
   loop_count NUMBER (2):=10;
   deptno          NUMBER (4):=0;
```

   c) Using a loop, retrieve the names of 10 departments and store the names in the associative array. Start with `department_id` 10. Increase `v_deptno` by 10 for every iteration of the loop. The following table shows the `department_id` for which you should retrieve the `department_name` and store in the associative array.

DEPARTMENT_ID	DEPARTMENT_NAME
10	Administration
20	Marketing
30	Purchasing
40	Human Resources
50	Shipping
60	IT
70	Public Relations
80	Sales
90	Executive
100	Finance

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 6: Working with Composite Data Types (continued)

```
BEGIN
  FOR i IN 1..f_loop_count
  LOOP
        v_deptno:=v_deptno+10;
   SELECT department_name
   INTO my_dept_table(i)
   FROM departments
   WHERE department_id = v_deptno;
  END LOOP;
```

d) Using another loop, retrieve the department names from the associative array and display them.

```
FOR i IN 1..f_loop_count
  LOOP
    DBMS_OUTPUT.PUT_LINE (my_dept_table(i));
  END LOOP;
END;
```

e) Execute and save your script as `lab_06_02_soln.sql`. The output is as follows:

```
anonymous block completed
Administration
Marketing
Purchasing
Human Resources
Shipping
IT
Public Relations
Sales
Executive
Finance
```

3) Modify the block that you created in Practice 2 to retrieve all information about each department from the `departments` table and display the information. Use an associative array with the `INDEX BY` table of records method.

a) Load the `lab_06_02_soln.sql` script.

b) You have declared the associative array to be of the `departments.department_name` type. Modify the declaration of the associative array to temporarily store the number, name, and location of all the departments. Use the `%ROWTYPE` attribute.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 6: Working with Composite Data Types (continued)

```
SET SERVEROUTPUT ON

DECLARE
   TYPE dept_table_type is table of departments%ROWTYPE
    INDEX BY PLS_INTEGER;
    my_dept_table   dept_table_type;
    f_loop_count       NUMBER (2):=10;
    v_deptno          NUMBER (4):=0;
```

c) Modify the `SELECT` statement to retrieve all department information currently in the `departments` table and store it in the associative array.

```
BEGIN
  FOR i IN 1..f_loop_count
  LOOP
   v_deptno := v_deptno + 10;
   SELECT *
   INTO my_dept_table(i)
   FROM departments
   WHERE department_id = v_deptno;
  END LOOP;
```

d) Using another loop, retrieve the department information from the associative array and display the information.

```
FOR i IN 1..f_loop_count
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Department Number: ' ||
my_dept_table(i).department_id
     || ' Department Name: ' ||
my_dept_table(i).department_name
     || ' Manager Id: '||  my_dept_table(i).manager_id
     || ' Location Id: ' || my_dept_table(i).location_id);
  END LOOP;
END;
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 6: Working with Composite Data Types (continued)

The sample output is as follows:

```
anonymous block completed
Department Number: 10 Department Name: Administration Manager Id: 200 Location Id: 1700
Department Number: 20 Department Name: Marketing Manager Id: 201 Location Id: 1800
Department Number: 30 Department Name: Purchasing Manager Id: 114 Location Id: 1700
Department Number: 40 Department Name: Human Resources Manager Id: 203 Location Id: 2400
Department Number: 50 Department Name: Shipping Manager Id: 121 Location Id: 1500
Department Number: 60 Department Name: IT Manager Id: 103 Location Id: 1400
Department Number: 70 Department Name: Public Relations Manager Id: 204 Location Id: 2700
Department Number: 80 Department Name: Sales Manager Id: 145 Location Id: 2500
Department Number: 90 Department Name: Executive Manager Id: 100 Location Id: 1700
Department Number: 100 Department Name: Finance Manager Id: 108 Location Id: 1700
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practices and Solutions for Lesson 7

Practice 7-1: Using Explicit Cursors

In this practice, you perform two exercises:

- First, you use an explicit cursor to process a number of rows from a table and populate another table with the results using a cursor `FOR` loop.
- Second, you write a PL/SQL block that processes information with two cursors, including one that uses a parameter.

1) Create a PL/SQL block to perform the following:

   a) In the declarative section, declare and initialize a variable named `v_deptno` of type `NUMBER`. Assign a valid department ID value (see table in step d for values).

   b) Declare a cursor named `c_emp_cursor`, which retrieves the `last_name`, `salary`, and `manager_id` of employees working in the department specified in `v_deptno`.

   c) In the executable section, use the cursor `FOR` loop to operate on the data retrieved. If the salary of the employee is less than 5,000 and if the manager ID is either 101 or 124, display the message "*<<last_name>>* Due for a raise." Otherwise, display the message "*<<last_name>>* Not Due for a raise."

   d) Test the PL/SQL block for the following cases:

Department ID	Message
10	Whalen Due for a raise
20	Hartstein Not Due for a raise Fay Not Due for a raise
50	Weiss Not Due for a raise Fripp Not Due for a raise Kaufling Not Due for a raise Vollman Not Due for a raise. . . . . . OConnell Due for a raise Grant Due for a raise
80	Russell Not Due for a raise Partners Not Due for a raise Errazuriz Not Due for a raise Cambrault Not Due for a raise . . . Livingston Not Due for a raise Johnson Not Due for a raise

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur@gmail.com) has a non-transferable license to use this Student Guide.

Practice 7-1: Using Explicit Cursors (continued)

2) Next, write a PL/SQL block that declares and uses two cursors—one without a parameter and one with a parameter. The first cursor retrieves the department number and the department name from the `departments` table for all departments whose ID number is less than 100. The second cursor receives the department number as a parameter, and retrieves employee details for those who work in that department and whose `employee_id` is less than 120.

   a) Declare a cursor `c_dept_cursor` to retrieve `department_id` and `department_name` for those departments with `department_id` less than 100. Order by `department_id`.

   b) Declare another cursor `c_emp_cursor` that takes the department number as parameter and retrieves the following data from the `employees` table: `last_name`, `job_id`, `hire_date`, and `salary` of those employees who work in that department, with `employee_id` less than 120.

   c) Declare variables to hold the values retrieved from each cursor. Use the `%TYPE` attribute while declaring variables.

   d) Open `c_dept_cursor` and use a simple loop to fetch values into the variables declared. Display the department number and department name. Use the appropriate cursor attribute to exit the loop.

   e) Open `c_emp_cursor` by passing the current department number as a parameter. Start another loop and fetch the values of `emp_cursor` into variables, and print all the details retrieved from the `employees` table.

      Note
      - Check whether `c_emp_cursor` is already open before opening the cursor.
      - Use the appropriate cursor attribute for the exit condition.
      - When the loop completes, print a line after you have displayed the details of each department, and close `c_emp_cursor`.

   f) End the first loop and close `c_dept_cursor`. Then end the executable section.

   g) Execute the script. The sample output is as follows:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practice 7-1: Using Explicit Cursors (continued)

```
anonymous block completed
Department Number : 10  Department Name : Administration
-----------------------------------------------------------------
Department Number : 20  Department Name : Marketing
-----------------------------------------------------------------
Department Number : 30  Department Name : Purchasing
Raphaely    PU_MAN   07-DEC-94   11000
Khoo    PU_CLERK   18-MAY-95   3100
Baida    PU_CLERK   24-DEC-97   2900
Tobias    PU_CLERK   24-JUL-97   2800
Himuro    PU_CLERK   15-NOV-98   2600
Colmenares    PU_CLERK   10-AUG-99   2500
-----------------------------------------------------------------
Department Number : 40  Department Name : Human Resources
-----------------------------------------------------------------
Department Number : 50  Department Name : Shipping
-----------------------------------------------------------------
Department Number : 60  Department Name : IT
Hunold    IT_PROG   03-JAN-90   9000
Ernst    IT_PROG   21-MAY-91   6000
Austin    IT_PROG   25-JUN-97   4800
Pataballa    IT_PROG   05-FEB-98   4800
Lorentz    IT_PROG   07-FEB-99   4200
-----------------------------------------------------------------
Department Number : 70  Department Name : Public Relations
-----------------------------------------------------------------
Department Number : 80  Department Name : Sales
-----------------------------------------------------------------
Department Number : 90  Department Name : Executive
King    AD_PRES   17-JUN-87   24000
Kochhar    AD_VP   21-SEP-89   17000
De Haan    AD_VP   13-JAN-93   17000
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Practice 7-2: Using Explicit Cursors – Optional

If you have time, complete the following optional practice. Here, create a PL/SQL block that uses an explicit cursor to determine the top *n* salaries of employees.

1) Run the `lab_07-2.sql` script to create the `top_salaries` table for storing the salaries of the employees.

2) In the declarative section, declare the `v_num` variable of the `NUMBER` type that holds a number `n`, representing the number of top *n* earners from the `employees` table. For example, to view the top five salaries, enter `5`. Declare another variable `sal` of type `employees.salary`. Declare a cursor, `c_emp_cursor`, which retrieves the salaries of employees in descending order. Remember that the salaries should not be duplicated.

3) In the executable section, open the loop and fetch the top *n* salaries, and then insert them into the `top_salaries` table. You can use a simple loop to operate on the data. Also, try and use the `%ROWCOUNT` and `%FOUND` attributes for the exit condition.

   Note: Make sure that you add an exit condition to avoid having an infinite loop.

4) After inserting data into the `top_salaries` table, display the rows with a `SELECT` statement. The output shown represents the five highest salaries in the `employees` table.

```
SALARY
----------------------
24000
17000
17000
14000
13500
```

5) Test a variety of special cases such as `v_num = 0` or where `v_num` is greater than the number of employees in the `employees` table. Empty the `top_salaries` table after each test.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution 7-1: Using Explicit Cursors

In this practice, you perform two exercises:

- First, you use an explicit cursor to process a number of rows from a table and populate another table with the results using a cursor `FOR` loop.
- Second, you write a PL/SQL block that processes information with two cursors, including one that uses a parameter.

1) Create a PL/SQL block to perform the following:

   a) In the declarative section, declare and initialize a variable named `v_deptno` of the `NUMBER` type. Assign a valid department ID value (see table in step d for values).

```
DECLARE
v_deptno NUMBER := 10;
```

   b) Declare a cursor named `c_emp_cursor`, which retrieves the `last_name`, `salary`, and `manager_id` of employees working in the department specified in `v_deptno`.

```
CURSOR  c_emp_cursor IS
   SELECT     last_name, salary,manager_id
   FROM       employees
   WHERE      department_id = v_deptno;
```

   c) In the executable section, use the cursor `FOR` loop to operate on the data retrieved. If the salary of the employee is less than 5,000 and if the manager ID is either 101 or 124, display the message "*<<last_name>>* Due for a raise." Otherwise, display the message "*<<last_name>>* Not Due for a raise."

```
BEGIN
 FOR emp_record IN c_emp_cursor
  LOOP
   IF emp_record.salary < 5000 AND (emp_record.manager_id=101
OR emp_record.manager_id=124) THEN
      DBMS_OUTPUT.PUT_LINE (emp_record.last_name || ' Due for
a raise');
   ELSE
      DBMS_OUTPUT.PUT_LINE (emp_record.last_name || ' Not Due
for a raise');
   END IF;
  END LOOP;
END;
```

   d) Test the PL/SQL block for the following cases:

Ankur N. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 7-1: Using Explicit Cursors (continued)

Department ID	Message
10	Whalen Due for a raise
20	Hartstein Not Due for a raise Fay Not Due for a raise
50	Weiss Not Due for a raise Fripp Not Due for a raise Kaufling Not Due for a raise Vollman Not Due for a raise. . . . . . OConnell Due for a raise Grant Due for a raise
80	Russell Not Due for a raise Partners Not Due for a raise Errazuriz Not Due for a raise Cambrault Not Due for a raise . . . Livingston Not Due for a raise Johnson Not Due for a raise

2) Next, write a PL/SQL block that declares and uses two cursors—one without a parameter and one with a parameter. The first cursor retrieves the department number and the department name from the `departments` table for all departments whose ID number is less than 100. The second cursor receives the department number as a parameter, and retrieves employee details for those who work in that department and whose `employee_id` is less than 120.

   a) Declare a cursor `c_dept_cursor` to retrieve `department_id` and `department_name` for those departments with `department_id` less than 100. Order by `department_id`.

```
DECLARE
  CURSOR c_dept_cursor IS
        SELECT department_id,department_name
        FROM  departments
        WHERE department_id < 100
        ORDER BY    department_id;
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 7-1: Using Explicit Cursors (continued)

b) Declare another cursor `c_emp_cursor` that takes the department number as parameter and retrieves the following data from the `employees` table: `last_name`, `job_id`, `hire_date`, and `salary` of those employees who work in that department, with `employee_id` less than 120.

```
CURSOR c_emp_cursor(v_deptno NUMBER) IS
        SELECT last_name,job_id,hire_date,salary
        FROM  employees
        WHERE department_id = v_deptno
        AND employee_id < 120;
```

c) Declare variables to hold the values retrieved from each cursor. Use the `%TYPE` attribute while declaring variables.

```
  v_current_deptno departments.department_id%TYPE;
  v_current_dname  departments.department_name%TYPE;
  v_ename employees.last_name%TYPE;
  v_job employees.job_id%TYPE;
  v_hiredate employees.hire_date%TYPE;
  v_sal employees.salary%TYPE;
```

d) Open `c_dept_cursor` and use a simple loop to fetch values into the variables declared. Display the department number and department name. Use the appropriate cursor attribute to exit the loop.

```
BEGIN
   OPEN c_dept_cursor;
      LOOP
          FETCH c_dept_cursor INTO v_current_deptno,
            v_current_dname;
          EXIT WHEN c_dept_cursor%NOTFOUND;
          DBMS_OUTPUT.PUT_LINE ('Department Number : ' ||
            v_current_deptno || '  Department Name : ' ||
            v_current_dname);
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (rajpati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 7-1: Using Explicit Cursors (continued)

e) Open `c_emp_cursor` by passing the current department number as a parameter. Start another loop and fetch the values of `emp_cursor` into variables, and print all the details retrieved from the `employees` table.

Note

- Check whether `c_emp_cursor` is already open before opening the cursor.
- Use the appropriate cursor attribute for the exit condition.
- When the loop completes, print a line after you have displayed the details of each department, and close `c_emp_cursor`.

```
IF c_emp_cursor%ISOPEN THEN
  CLOSE c_emp_cursor;
END IF;
OPEN c_emp_cursor (v_current_deptno);
  LOOP
    FETCH c_emp_cursor INTO  v_ename,v_job,v_hiredate,v_sal;
    EXIT WHEN c_emp_cursor%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (v_ename || '    ' ||  v_job
                     || '   '  ||  v_hiredate || '   ' ||
v_sal);
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('-----------------------------------------
-----------------------------------------------------');
         CLOSE c_emp_cursor;
```

f) End the first loop and close `c_dept_cursor`. Then end the executable section.

```
    END LOOP;
  CLOSE c_dept_cursor;
END;
```

g) Execute the script. The sample output is as follows:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 7-1: Using Explicit Cursors (continued)

```
anonymous block completed
Department Number : 10  Department Name : Administration
-----------------------------------------------------------------
Department Number : 20  Department Name : Marketing
-----------------------------------------------------------------
Department Number : 30  Department Name : Purchasing
Raphaely    PU_MAN   07-DEC-94   11000
Khoo    PU_CLERK   18-MAY-95   3100
Baida    PU_CLERK   24-DEC-97   2900
Tobias    PU_CLERK   24-JUL-97   2800
Himuro    PU_CLERK   15-NOV-98   2600
Colmenares    PU_CLERK   10-AUG-99   2500
-----------------------------------------------------------------
Department Number : 40  Department Name : Human Resources
-----------------------------------------------------------------
Department Number : 50  Department Name : Shipping
-----------------------------------------------------------------
Department Number : 60  Department Name : IT
Hunold    IT_PROG   03-JAN-90   9000
Ernst    IT_PROG   21-MAY-91   6000
Austin    IT_PROG   25-JUN-97   4800
Pataballa    IT_PROG   05-FEB-98   4800
Lorentz    IT_PROG   07-FEB-99   4200
-----------------------------------------------------------------
Department Number : 70  Department Name : Public Relations
-----------------------------------------------------------------
Department Number : 80  Department Name : Sales
-----------------------------------------------------------------
Department Number : 90  Department Name : Executive
King    AD_PRES   17-JUN-87   24000
Kochhar    AD_VP   21-SEP-89   17000
De Haan    AD_VP   13-JAN-93   17000
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution 7-2: Using Explicit Cursors – Optional

If you have time, complete the following optional exercise. Here, create a PL/SQL block that uses an explicit cursor to determine the top *n* salaries of employees.

1) Execute the `lab_07-02.sql` script to create a new table, `top_salaries`, for storing the salaries of the employees.

2) In the declarative section, declare a variable `v_num` of type `NUMBER` that holds a number `n`, representing the number of top *n* earners from the `employees` table. For example, to view the top five salaries, enter `5`. Declare another variable `sal` of type `employees.salary`. Declare a cursor, `c_emp_cursor`, which retrieves the salaries of employees in descending order. Remember that the salaries should not be duplicated.

```
DECLARE
  v_num         NUMBER(3) := 5;
  v_sal         employees.salary%TYPE;
  CURSOR        c_emp_cursor IS
    SELECT      salary
    FROM        employees
    ORDER BY    salary DESC;
```

3) In the executable section, open the loop and fetch the top *n* salaries, and then insert them into the `top_salaries` table. You can use a simple loop to operate on the data. Also, try and use the `%ROWCOUNT` and `%FOUND` attributes for the exit condition. **Note:** Make sure that you add an exit condition to avoid having an infinite loop.

```
BEGIN
  OPEN c_emp_cursor;
  FETCH c_emp_cursor INTO  v_sal;
  WHILE c_emp_cursor%ROWCOUNT <= v_num AND c_emp_cursor%FOUND
LOOP
              INSERT INTO top_salaries (salary)
              VALUES (v_sal);
    FETCH c_emp_cursor INTO v_sal;
  END LOOP;
  CLOSE c_emp_cursor;
END;
```

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 7-2: Using Explicit Cursors – Optional (continued)

4) After inserting data into the `top_salaries` table, display the rows with a `SELECT` statement. The output shown represents the five highest salaries in the `employees` table.

```
/
SELECT * FROM top_salaries;
```

The sample output is as follows:

```
SALARY
----------------------
24000
17000
17000
14000
13500
```

5) Test a variety of special cases such as `v_num = 0` or where `v_num` is greater than the number of employees in the `employees` table. Empty the `top_salaries` table after each test.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practices and Solutions for Lesson 8

Practice 8-1: Handling Predefined Exceptions

In this practice, you write a PL/SQL block that applies a predefined exception in order to process only one record at a time. The PL/SQL block selects the name of the employee with a given salary value.

1) Execute the command in the `lab_05_01.sql` file to re-create the `messages` table.

2) In the declarative section, declare two variables: `v_ename` of type `employees.last_name` and `v_emp_sal` of type `employees.salary`. Initialize the latter to 6000.

3) In the executable section, retrieve the last names of employees whose salaries are equal to the value in `v_emp_sal`. If the salary entered returns only one row, insert into the `messages` table the employee's name and the salary amount.
Note: Do not use explicit cursors.

4) If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the `messages` table the message "No employee with a salary of <*salary*>."

5) If the salary entered returns multiple rows, handle the exception with an appropriate exception handler and insert into the `messages` table the message "More than one employee with a salary of <*salary*>."

6) Handle any other exception with an appropriate exception handler and insert into the `messages` table the message "Some other error occurred."

7) Display the rows from the `messages` table to check whether the PL/SQL block has executed successfully. The output is as follows:

```
RESULTS
-----------------------------------------------
More than one employee with a salary of 6000

1 rows selected
```

8) Change the initialized value of `v_emp_sal` to 2000 and re-execute. Output is as follows:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankurm@gmail.com) has a non-transferable license to use this Student Guide.

Practice 8-1: Handling Predefined Exceptions (continued)

```
RESULTS
-----------------------------------------------
More than one employee with a salary of 6000
No employee with a salary of 2000

2 rows selected
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practice 8-2: Handling Standard Oracle Server Exceptions

In this practice, you write a PL/SQL block that declares an exception for the Oracle Server error `ORA-02292 (integrity constraint violated - child record found)`. The block tests for the exception and outputs the error message.

1) In the declarative section, declare an exception `e_childrecord_exists`. Associate the declared exception with the standard Oracle Server error `-02292`.

2) In the executable section, display "Deleting department 40...." Include a `DELETE` statement to delete the department with the `department_id` 40.

3) Include an exception section to handle the `e_childrecord_exists` exception and display the appropriate message.

   The sample output is as follows:

```
anonymous block completed
 Deleting department 40........
 Cannot delete this department. There are employees in this department (child records exist.)
```

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution 8-1: Handling Predefined Exceptions

In this practice, you write a PL/SQL block that applies a predefined exception in order to process only one record at a time. The PL/SQL block selects the name of the employee with a given salary value.

1) Execute the command in the `lab_05_01.sql` file to recreate the `messages` table.

2) In the declarative section, declare two variables: `v_ename` of type `employees.last_name` and `v_emp_sal` of type `employees.salary`. Initialize the latter to 6000.

```
DECLARE
  v_ename     employees.last_name%TYPE;
  v_emp_sal   employees.salary%TYPE := 6000;
```

3) In the executable section, retrieve the last names of employees whose salaries are equal to the value in `v_emp_sal`. If the salary entered returns only one row, insert into the `messages` table the employee's name and the salary amount.
Note: Do not use explicit cursors.

```
BEGIN
  SELECTlast_name
  INTO        v_ename
  FROM        employees
  WHERE       salary = v_emp_sal;
  INSERT INTO messages (results)
  VALUES (v_ename || ' - ' || v_emp_sal);
```

4) If the salary entered does not return any rows, handle the exception with an appropriate exception handler and insert into the `messages` table the message "No employee with a salary of <*salary*>."

```
EXCEPTION
  WHEN no_data_found THEN
    INSERT INTO messages (results)
    VALUES ('No employee with a salary of '||
            TO_CHAR(v_emp_sal));
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur Jyotiprajapati (ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 8-1: Handling Predefined Exceptions (continued)

5) If the salary entered returns multiple rows, handle the exception with an appropriate exception handler and insert into the `messages` table the message "More than one employee with a salary of <*salary*>."

```
WHEN too_many_rows THEN
    INSERT INTO messages (results)
    VALUES ('More than one employee with a salary of '||
             TO_CHAR(v_emp_sal));
```

6) Handle any other exception with an appropriate exception handler and insert into the `messages` table the message "Some other error occurred."

```
WHEN others THEN
    INSERT INTO messages (results)
    VALUES ('Some other error occurred.');
END;
```

7) Display the rows from the `messages` table to check whether the PL/SQL block has executed successfully.

```
/
SELECT * FROM messages;
```

The output is as follows:

```
RESULTS
--------------------------------------------
More than one employee with a salary of 6000

1 rows selected
```

8) Change the initialized value of `v_emp_sal` to 2000 and re-execute. The output is as follows:

```
RESULTS
--------------------------------------------
More than one employee with a salary of 6000
No employee with a salary of 2000

2 rows selected
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 8-2: Handling Standard Oracle Server Exceptions

In this practice, you write a PL/SQL block that declares an exception for the Oracle Server error `ORA-02292 (integrity constraint violated - child record found)`. The block tests for the exception and outputs the error message.

1) In the declarative section, declare an exception `e_childrecord_exists`. Associate the declared exception with the standard Oracle Server error `-02292`.

```
SET SERVEROUTPUT ON
DECLARE
   e_childrecord_exists EXCEPTION;
   PRAGMA EXCEPTION_INIT(e_childrecord_exists, -02292);
```

2) In the executable section, display "Deleting department 40...." Include a `DELETE` statement to delete the department with `department_id` 40.

```
BEGIN
    DBMS_OUTPUT.PUT_LINE(' Deleting department 40........');
   delete from departments where department_id=40;
```

3) Include an exception section to handle the `e_childrecord_exists` exception and display the appropriate message.

```
EXCEPTION
   WHEN e_childrecord_exists THEN
   DBMS_OUTPUT.PUT_LINE(' Cannot delete this department. There
are employees in this department (child records exist.) ');
END;
```

The sample output is as follows:

```
anonymous block completed
 Deleting department 40........
 Cannot delete this department. There are employees in this department (child records exist.)
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practices and Solutions for Lesson 9

Practice 9: Creating and Using Stored Procedures

In this practice, you modify existing scripts to create and use stored procedures.

1) Load the `sol_02_04.sql` script from the /home/oracle/plsf/soln/ folder.

   a) Modify the script to convert the anonymous block to a procedure called `greet`. (**Hint:** Also remove the `SET SERVEROUTPUT ON` command.)

   b) Execute the script to create the procedure. The output results should be as follows:

```
Results | Script Output | Exp
PROCEDURE greet Compiled.
```

   c) Save this script as `lab_09_01_soln.sql`.

   d) Click the Clear button to clear the workspace.

   e) Create and execute an anonymous block to invoke the `greet` procedure. (**Hint:** Ensure that you enable `SERVEROUTPUT` at the beginning of the block.)

   The output should be similar to the following:

```
Results | Script Output | Explain
anonymous block completed
 Hello World
TODAY IS : 10-JUL-09
TOMORROW IS : 11-JUL-09
```

2) Modify the `lab_09_01_soln.sql` script as follows:

   a) Drop the `greet` procedure by issuing the following command:

```
DROP PROCEDURE greet;
```

   b) Modify the procedure to accept an argument of type `VARCHAR2`. Call the argument `p_name`.

   c) Print `Hello` <*name*> (that is, the contents of the argument) instead of printing `Hello World`.

   d) Save your script as `lab_09_02_soln.sql`.

   e) Execute the script to create the procedure. The output results should be as follows:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Practice 9: Creating and Using Stored Procedures (continued)

```
Results | Script Output | Explain
DROP PROCEDURE greet succeeded.
PROCEDURE greet Compiled.
```

f) Create and execute an anonymous block to invoke the `greet` procedure with a parameter value. The block should also produce the output.

The sample output should be similar to the following:

```
Results | Script Output | Ex
anonymous block completed
 Hello Nancy
TODAY IS : 10-JUL-09
TOMORROW IS : 11-JUL-09
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 9: Creating and Using Stored Procedures

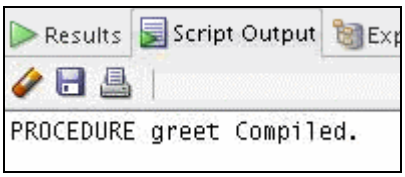
In this practice, you modify existing scripts to create and use stored procedures.

1) Load the `sol_02_04.sql` script from the /home/oracle/plsf/soln/ folder.

   a) Modify the script to convert the anonymous block to a procedure called `greet`. (**Hint:** Also remove the `SET SERVEROUTPUT ON` command.)

```
CREATE PROCEDURE greet IS
   V_today DATE:=SYSDATE;
   V_tomorrow today%TYPE;
...
```

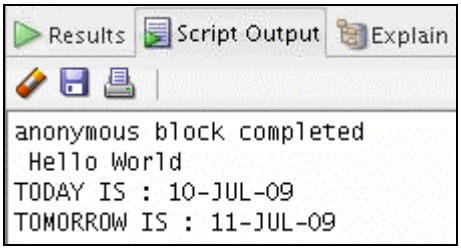
   b) Execute the script to create the procedure. The output results should be as follows:



   c) Save this script as `lab_09_01_soln.sql`.
   d) Click the Clear button to clear the workspace.
   e) Create and execute an anonymous block to invoke the `greet` procedure. (**Hint:** Ensure that you enable `SERVEROUTPUT` at the beginning of the block.)

```
SET SERVEROUTPUT ON

BEGIN
 greet;
END;
```

The output should be similar to the following:



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 9: Creating and Using Stored Procedures (continued)

2) Modify the `lab_09_01_soln.sql` script as follows:

   a) Drop the `greet` procedure by issuing the following command:

```
DROP PROCEDURE greet;
```

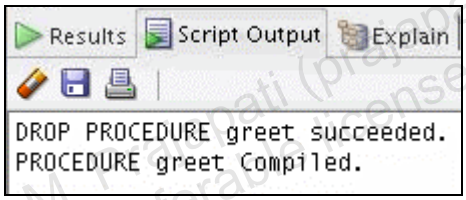
   b) Modify the procedure to accept an argument of type `VARCHAR2`. Call the argument `p_name`.

```
CREATE PROCEDURE greet(p_name VARCHAR2) IS
   V_today DATE:=SYSDATE;
   V_tomorrow today%TYPE;
```

   c) Print `Hello` <*name*> instead of printing `Hello World`.

```
BEGIN
   V_tomorrow:=v_today +1;
   DBMS_OUTPUT.PUT_LINE(' Hello '|| p_name);
...
```

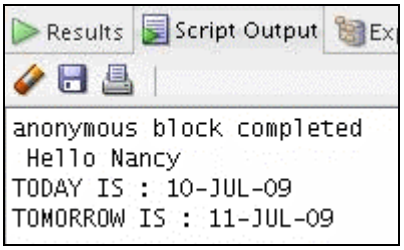
   d) Save your script as `lab_09_02_soln.sql`.
   e) Execute the script to create the procedure. The output results should be as follows:



   f) Create and execute an anonymous block to invoke the `greet` procedure with a parameter value. The block should also produce the output.

```
SET SERVEROUTPUT ON;
BEGIN
 greet('Nancy');
END;
```

The sample output should be similar to the following:



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

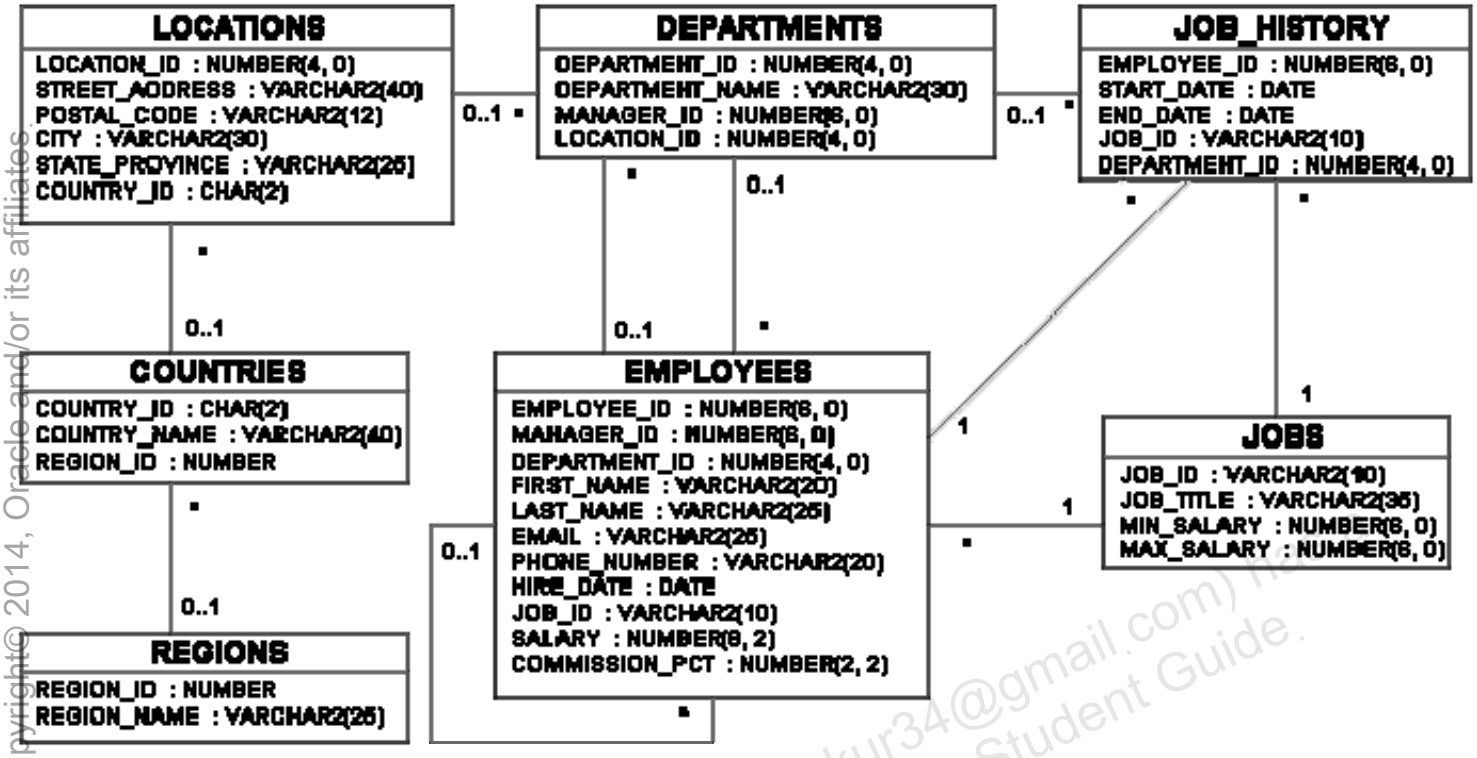
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

B
Table Descriptions and Data

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

ENTITY RELATIONSHIP DIAGRAM



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has non-transferable license to use this Student Guide.

Tables in the Schema

```
SELECT * FROM tab;
```

TNAME	TABTYPE	CLUSTERID
COUNTRIES	TABLE	
DEPARTMENTS	TABLE	
EMPLOYEES	TABLE	
EMP_DETAILS_VIEW	VIEW	
JOBS	TABLE	
JOB_HISTORY	TABLE	
LOCATIONS	TABLE	
REGIONS	TABLE	

8 rows selected.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

`regions` Table

```
DESCRIBE regions
```

Name	Null?	Type
REGION_ID	NOT NULL	NUMBER
REGION_NAME		VARCHAR2(25)

```
SELECT * FROM regions;
```

REGION_ID	REGION_NAME
1	Europe
2	Americas
3	Asia
4	Middle East and Africa

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

`countries` Table

```
DESCRIBE countries
```

Name	Null?	Type
COUNTRY_ID	NOT NULL	CHAR(2)
COUNTRY_NAME		VARCHAR2(40)
REGION_ID		NUMBER

```
SELECT * FROM countries;
```

CO	COUNTRY_NAME	REGION_ID
AR	Argentina	2
AU	Australia	3
BE	Belgium	1
BR	Brazil	2
CA	Canada	2
CH	Switzerland	1
CN	China	3
DE	Germany	1
DK	Denmark	1
EG	Egypt	4
FR	France	1
HK	HongKong	3
IL	Israel	4
IN	India	3
CO	COUNTRY_NAME	REGION_ID
IT	Italy	1
JP	Japan	3
KW	Kuwait	4
MX	Mexico	2
NG	Nigeria	4
NL	Netherlands	1
SG	Singapore	3
UK	United Kingdom	1
US	United States of America	2
ZM	Zambia	4
ZW	Zimbabwe	4

25 rows selected.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

`locations` Table

```
DESCRIBE locations;
```

Name	Null?	Type
LOCATION_ID	NOT NULL	NUMBER(4)
STREET_ADDRESS		VARCHAR2(40)
POSTAL_CODE		VARCHAR2(12)
CITY	NOT NULL	VARCHAR2(30)
STATE_PROVINCE		VARCHAR2(25)
COUNTRY_ID		CHAR(2)

```
SELECT * FROM locations;
```

LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	CO
1000	1297 Via Cola di Rie	00989	Roma		IT
1100	93091 Calle della Testa	10934	Venice		IT
1200	2017 Shinjuku-ku	1689	Tokyo	Tokyo Prefecture	JP
1300	9450 Kamiya-cho	6823	Hiroshima		JP
1400	2014 Jabberwocky Rd	26192	Southlake	Texas	US
1500	2011 Interiors Blvd	99236	South San Francisco	California	US
1600	2007 Zagora St	50090	South Brunswick	New Jersey	US
1700	2004 Charade Rd	98199	Seattle	Washington	US
1800	147 Spadina Ave	M5V 2L7	Toronto	Ontario	CA
1900	6092 Boxwood St	YSW 9T2	Whitehorse	Yukon	CA
2000	40-5-12 Laogianggen	190518	Beijing		CN
2100	1298 Vileparle (E)	490231	Bombay	Maharashtra	IN
LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	CO
2400	8204 Arthur St		London		UK
2500	Magdalen Centre, The Oxford Science Park	OX9 9ZB	Oxford	Oxford	UK
2600	9702 Chester Road	09629850293	Stretford	Manchester	UK
2700	Schwanthalerstr. 7031	80925	Munich	Bavaria	DE
2800	Rua Frei Caneca 1360	01307-002	Sao Paulo	Sao Paulo	BR
2900	20 Rue des Corps-Saints	1730	Geneva	Geneve	CH
3000	Murtenstrasse 921	3095	Bern	BE	CH
3100	Pieter Breughelstraat 837	3029SK	Utrecht	Utrecht	NL
3200	Mariano Escobedo 9991	11932	Mexico City	Distrito Federal,	MX

23 rows selected.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur Pratapati (pratapati.ankur@gmail.com) has a non-transferable license to use this Student Guide.

`departments` Table

```
DESCRIBE departments
```

Name	Null?	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

```
SELECT * FROM departments;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
150	Shareholder Services		1700
160	Benefits		1700
170	Manufacturing		1700
180	Construction		1700
190	Contracting		1700
200	Operations		1700
210	IT Support		1700
220	NOC		1700
230	IT Helpdesk		1700
240	Government Sales		1700
250	Retail Sales		1700
260	Recruiting		1700
270	Payroll		1700

27 rows selected.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has non-transferable license to use this Student Guide.

`jobs` Table

```
DESCRIBE jobs
```

Name	Null?	Type
JOB_ID	NOT NULL	VARCHAR2(10)
JOB_TITLE	NOT NULL	VARCHAR2(35)
MIN_SALARY		NUMBER(6)
MAX_SALARY		NUMBER(6)

```
SELECT * FROM jobs;
```

JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
AD_PRES	President	20000	40000
AD_VP	Administration Vice President	15000	30000
AD_ASST	Administration Assistant	3000	6000
FI_MGR	Finance Manager	8200	16000
FI_ACCOUNT	Accountant	4200	9000
AC_MGR	Accounting Manager	8200	16000
AC_ACCOUNT	Public Accountant	4200	9000
SA_MAN	Sales Manager	10000	20000
SA_REP	Sales Representative	6000	12000
PU_MAN	Purchasing Manager	8000	15000
PU_CLERK	Purchasing Clerk	2500	5500
ST_MAN	Stock Manager	5500	8500
ST_CLERK	Stock Clerk	2000	5000
SH_CLERK	Shipping Clerk	2500	5500
JOB_ID	JOB_TITLE	MIN_SALARY	MAX_SALARY
IT_PROG	Programmer	4000	10000
MK_MAN	Marketing Manager	9000	15000
MK_REP	Marketing Representative	4000	9000
HR_REP	Human Resources Representative	4000	9000
PR_REP	Public Relations Representative	4500	10500

19 rows selected.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

`employees` Table

```
DESCRIBE employees
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

`employees` Table (continued)

The headings for the `commission_pct`, `manager_id`, and `department_id` columns are set to `comm`, `mgrid`, and `deptid`, respectively, in the following screenshot to fit the table values across the page.

```
SELECT * FROM employees;
```

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
100	Steven	King	SKING	515.123.4567	17-JUN-87	AD_PRES	24000			90
101	Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-89	AD_VP	17000		100	90
102	Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-93	AD_VP	17000		100	90
103	Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-90	IT_PROG	9000		102	60
104	Bruce	Ernst	BERNST	590.423.4568	21-MAY-91	IT_PROG	6000		103	60
105	David	Austin	DAUSTIN	590.423.4569	25-JUN-97	IT_PROG	4800		103	60
106	Valli	Pataballa	VPATABAL	590.423.4560	05-FEB-98	IT_PROG	4800		103	60
107	Diana	Lorentz	DLORENTZ	590.423.5567	07-FEB-99	IT_PROG	4200		103	60
108	Nancy	Greenberg	NGREENBE	515.124.4569	17-AUG-94	FI_MGR	12000		101	100
109	Daniel	Faviet	DFAVIET	515.124.4169	16-AUG-94	FI_ACCOUNT	9000		108	100
110	John	Chen	JCHEN	515.124.4269	28-SEP-97	FI_ACCOUNT	8200		108	100
111	Ismael	Sciarra	ISCIARRA	515.124.4369	30-SEP-97	FI_ACCOUNT	7700		108	100
112	Jose Manuel	Urman	JMURMAN	515.124.4469	07-MAR-98	FI_ACCOUNT	7800		108	100
113	Luis	Popp	LPOPP	515.124.4567	07-DEC-99	FI_ACCOUNT	6900		108	100
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
114	Den	Raphaely	DRAPHEAL	515.127.4561	07-DEC-94	PU_MAN	11000		100	30
115	Alexander	Khoo	AKHOO	515.127.4562	18-MAY-95	PU_CLERK	3100		114	30
116	Shelli	Baida	SBAIDA	515.127.4563	24-DEC-97	PU_CLERK	2900		114	30
117	Sigal	Tobias	STOBIAS	515.127.4564	24-JUL-97	PU_CLERK	2800		114	30
118	Guy	Himuro	GHIMURO	515.127.4565	15-NOV-98	PU_CLERK	2600		114	30
119	Karen	Colmenares	KCOLMENA	515.127.4566	10-AUG-99	PU_CLERK	2500		114	30
120	Matthew	Weiss	MWEISS	650.123.1234	18-JUL-96	ST_MAN	8000		100	50
121	Adam	Fripp	AFRIPP	650.123.2234	10-APR-97	ST_MAN	8200		100	50
122	Payam	Kaufling	PKAUFLIN	650.123.3234	01-MAY-95	ST_MAN	7900		100	50
123	Shanta	Vollman	SVOLLMAN	650.123.4234	10-OCT-97	ST_MAN	6500		100	50
124	Kevin	Mourgos	KMOURGOS	650.123.5234	16-NOV-99	ST_MAN	5800		100	50
125	Julia	Nayer	JNAYER	650.124.1214	16-JUL-97	ST_CLERK	3200		120	50
126	Irene	Mikkilineni	IMIKKILI	650.124.1224	28-SEP-98	ST_CLERK	2700		120	50
127	James	Landry	JLANDRY	650.124.1334	14-JAN-99	ST_CLERK	2400		120	50

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajap0ankur@humber.com) has a non-transferable license to use this Student Guide

`employees` Table (continued)

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
128	Steven	Markle	SMARKLE	650.124.1434	08-MAR-00	ST_CLERK	2200		120	50
129	Laura	Bissot	LBISSOT	650.124.5234	20-AUG-97	ST_CLERK	3300		121	50
130	Mozhe	Atkinson	MATKINSO	650.124.6234	30-OCT-97	ST_CLERK	2800		121	50
131	James	Marlow	JAMRLOW	650.124.7234	16-FEB-97	ST_CLERK	2500		121	50
132	TJ	Olson	TJOLSON	650.124.8234	10-APR-99	ST_CLERK	2100		121	50
133	Jason	Mallin	JMALLIN	650.127.1934	14-JUN-96	ST_CLERK	3300		122	50
134	Michael	Rogers	MROGERS	650.127.1834	26-AUG-98	ST_CLERK	2900		122	50
135	Ki	Gee	KGEE	650.127.1734	12-DEC-99	ST_CLERK	2400		122	50
136	Hazel	Philtanker	HPHILTAN	650.127.1634	06-FEB-00	ST_CLERK	2200		122	50
137	Renske	Ladwig	RLADWIG	650.121.1234	14-JUL-95	ST_CLERK	3600		123	50
138	Stephen	Stiles	SSTILES	650.121.2034	26-OCT-97	ST_CLERK	3200		123	50
139	John	Seo	JSEO	650.121.2019	12-FEB-98	ST_CLERK	2700		123	50
140	Joshua	Patel	JPATEL	650.121.1834	06-APR-98	ST_CLERK	2500		123	50
141	Trenna	Rajs	TRAJS	650.121.8009	17-OCT-95	ST_CLERK	3500		124	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
142	Curtis	Davies	CDAVIES	650.121.2994	29-JAN-97	ST_CLERK	3100		124	50
143	Randall	Matos	RMATOS	650.121.2874	15-MAR-98	ST_CLERK	2600		124	50
144	Peter	Vargas	PVARGAS	650.121.2004	09-JUL-98	ST_CLERK	2500		124	50
145	John	Russell	JRUSSEL	011.44.1344.429268	01-OCT-96	SA_MAN	14000	.4	100	80
146	Karen	Partners	KPARTNER	011.44.1344.467268	05-JAN-97	SA_MAN	13500	.3	100	80
147	Alberto	Errazuriz	AERRAZUR	011.44.1344.429278	10-MAR-97	SA_MAN	12000	.3	100	80
148	Gerald	Cambrault	GCAMBRAU	011.44.1344.619268	15-OCT-99	SA_MAN	11000	.3	100	80
149	Eleni	Zlotkey	EZLOTKEY	011.44.1344.429018	29-JAN-00	SA_MAN	10500	.2	100	80
150	Peter	Tucker	PTUCKER	011.44.1344.129268	30-JAN-97	SA_REP	10000	.3	145	80
151	David	Bernstein	DBERNSTE	011.44.1344.345268	24-MAR-97	SA_REP	9500	.25	145	80
152	Peter	Hall	PHALL	011.44.1344.478968	20-AUG-97	SA_REP	9000	.25	145	80
153	Christopher	Olsen	COLSEN	011.44.1344.498718	30-MAR-98	SA_REP	8000	.2	145	80
154	Nanette	Cambrault	NCAMBRAU	011.44.1344.987668	09-DEC-98	SA_REP	7500	.2	145	80
155	Oliver	Tuvault	OTUVAULT	011.44.1344.486508	23-NOV-99	SA_REP	7000	.15	145	80
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
156	Janette	King	JKING	011.44.1345.429268	30-JAN-96	SA_REP	10000	.35	146	80
157	Patrick	Sully	PSULLY	011.44.1345.929268	04-MAR-96	SA_REP	9500	.35	146	80
158	Allan	McEwen	AMCEWEN	011.44.1345.829268	01-AUG-96	SA_REP	9000	.35	146	80
159	Lindsey	Smith	LSMITH	011.44.1345.729268	10-MAR-97	SA_REP	8000	.3	146	80
160	Louise	Doran	LDORAN	011.44.1345.629268	15-DEC-97	SA_REP	7500	.3	146	80
161	Sarath	Sewall	SSEWALL	011.44.1345.529268	03-NOV-98	SA_REP	7000	.25	146	80
162	Clara	Vishney	CVISHNEY	011.44.1346.129268	11-NOV-97	SA_REP	10500	.25	147	80
163	Danielle	Greene	DGREENE	011.44.1346.229268	19-MAR-99	SA_REP	9500	.15	147	80
164	Mattea	Marvins	MMARVINS	011.44.1346.329268	24-JAN-00	SA_REP	7200	.1	147	80
165	David	Lee	DLEE	011.44.1346.529268	23-FEB-00	SA_REP	6800	.1	147	80
166	Sundar	Ande	SANDE	011.44.1346.629268	24-MAR-00	SA_REP	6400	.1	147	80
167	Amit	Banda	ABANDA	011.44.1346.729268	21-APR-00	SA_REP	6200	.1	147	80
168	Lisa	Ozer	LOZER	011.44.1343.929268	11-MAR-97	SA_REP	11500	.25	148	80
169	Harrison	Bloom	HBLOOM	011.44.1343.829268	23-MAR-98	SA_REP	10000	.2	148	80

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Anil M. Prajapati (oracle.training.humber@gmail.com) has a non-transferable license to use this Student Guide.

`employees` Table (continued)

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
170	Tayler	Fox	TFOX	011.44.1343.729268	24-JAN-98	SA_REP	9600	.2	148	80
171	William	Smith	WSMITH	011.44.1343.629268	23-FEB-99	SA_REP	7400	.15	148	80
172	Elizabeth	Bates	EBATES	011.44.1343.529268	24-MAR-99	SA_REP	7300	.15	148	80
173	Sundita	Kumar	SKUMAR	011.44.1343.329268	21-APR-00	SA_REP	6100	.1	148	80
174	Ellen	Abel	EABEL	011.44.1644.429267	11-MAY-96	SA_REP	11000	.3	149	80
175	Alyssa	Hutton	AHUTTON	011.44.1644.429266	19-MAR-97	SA_REP	8800	.25	149	80
176	Jonathon	Taylor	JTAYLOR	011.44.1644.429265	24-MAR-98	SA_REP	8600	.2	149	80
177	Jack	Livingston	JLIVINGS	011.44.1644.429264	23-APR-98	SA_REP	8400	.2	149	80
178	Kimberely	Grant	KGRANT	011.44.1644.429263	24-MAY-99	SA_REP	7000	.15	149	
179	Charles	Johnson	CJOHNSON	011.44.1644.429262	04-JAN-00	SA_REP	6200	.1	149	80
180	Winston	Taylor	WTAYLOR	650.507.9876	24-JAN-98	SH_CLERK	3200		120	50
181	Jean	Fleaur	JFLEAUR	650.507.9877	23-FEB-98	SH_CLERK	3100		120	50
182	Martha	Sullivan	MSULLIVA	650.507.9878	21-JUN-99	SH_CLERK	2500		120	50
183	Girard	Geoni	GGEONI	650.507.9879	03-FEB-00	SH_CLERK	2800		120	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
184	Nandita	Sarchand	NSARCHAN	650.509.1876	27-JAN-96	SH_CLERK	4200		121	50
185	Alexis	Bull	ABULL	650.509.2876	20-FEB-97	SH_CLERK	4100		121	50
186	Julia	Dellinger	JDELLING	650.509.3876	24-JUN-98	SH_CLERK	3400		121	50
187	Anthony	Cabrio	ACABRIO	650.509.4876	07-FEB-99	SH_CLERK	3000		121	50
188	Kelly	Chung	KCHUNG	650.505.1876	14-JUN-97	SH_CLERK	3800		122	50
189	Jennifer	Dilly	JDILLY	650.505.2876	13-AUG-97	SH_CLERK	3600		122	50
190	Timothy	Gates	TGATES	650.505.3876	11-JUL-98	SH_CLERK	2900		122	50
191	Randall	Perkins	RPERKINS	650.505.4876	19-DEC-99	SH_CLERK	2500		122	50
192	Sarah	Bell	SBELL	650.501.1876	04-FEB-96	SH_CLERK	4000		123	50
193	Britney	Everett	BEVERETT	650.501.2876	03-MAR-97	SH_CLERK	3900		123	50
194	Samuel	McCain	SMCCAIN	650.501.3876	01-JUL-98	SH_CLERK	3200		123	50
195	Vance	Jones	VJONES	650.501.4876	17-MAR-99	SH_CLERK	2800		123	50
196	Alana	Walsh	AWALSH	650.507.9811	24-APR-98	SH_CLERK	3100		124	50
197	Kevin	Feeney	KFEENEY	650.507.9822	23-MAY-98	SH_CLERK	3000		124	50
EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	comm	mgrid	deptid
198	Donald	OConnell	DOCONNEL	650.507.9833	21-JUN-99	SH_CLERK	2600		124	50
199	Douglas	Grant	DGRANT	650.507.9844	13-JAN-00	SH_CLERK	2600		124	50
200	Jennifer	Whalen	JWHALEN	515.123.4444	17-SEP-87	AD_ASST	4400		101	10
201	Michael	Hartstein	MHARTSTE	515.123.5555	17-FEB-96	MK_MAN	13000		100	20
202	Pat	Fay	PFAY	603.123.6666	17-AUG-97	MK_REP	6000		201	20
203	Susan	Mavris	SMAVRIS	515.123.7777	07-JUN-94	HR_REP	6500		101	40
204	Hermann	Baer	HBAER	515.123.8888	07-JUN-94	PR_REP	10000		101	70
205	Shelley	Higgins	SHIGGINS	515.123.8080	07-JUN-94	AC_MGR	12000		101	110
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	8300		205	110

107 rows selected.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Anwar M. Rajapathi (prajapathi@humber.ca) has a non-transferable license to use this student guide.

`job_history` Table

```
DESCRIBE job_history
```

Name	Null?	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
START_DATE	NOT NULL	DATE
END_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
DEPARTMENT_ID		NUMBER(4)

```
SELECT * FROM job_history;
```

EMPLOYEE_ID	START_DAT	END_DATE	JOB_ID	deptid
102	13-JAN-93	24-JUL-98	IT_PROG	60
101	21-SEP-89	27-OCT-93	AC_ACCOUNT	110
101	28-OCT-93	15-MAR-97	AC_MGR	110
201	17-FEB-96	19-DEC-99	MK_REP	20
114	24-MAR-98	31-DEC-99	ST_CLERK	50
122	01-JAN-99	31-DEC-99	ST_CLERK	50
200	17-SEP-87	17-JUN-93	AD_ASST	90
176	24-MAR-98	31-DEC-98	SA_REP	80
176	01-JAN-99	31-DEC-99	SA_MAN	80
200	01-JUL-94	31-DEC-98	AC_ACCOUNT	90

10 rows selected.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Humber College

C

Using SQL Developer

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

After completing this appendix, you should be able to do the following:

- List the key features of Oracle SQL Developer
- Identify the menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use SQL Worksheet
- Save and run SQL scripts
- Create and save reports

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

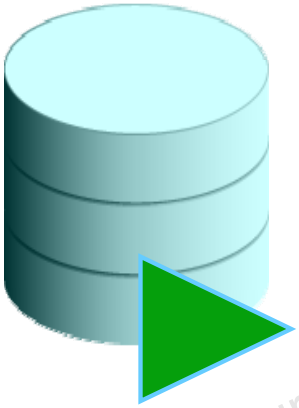
Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

In this appendix, you are introduced to the graphical tool called SQL Developer. You learn how to use SQL Developer for your database development tasks. You learn how to use SQL Worksheet to execute SQL statements and SQL scripts.

What Is Oracle SQL Developer?

- Oracle SQL Developer is a graphical tool that enhances productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema by using standard Oracle database authentication.



SQL Developer

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

What Is Oracle SQL Developer?

Oracle SQL Developer is a free graphical tool that is designed to improve your productivity and simplify the development of everyday database tasks. With just a few clicks, you can easily create and debug stored procedures, test SQL statements, and view optimizer plans.

SQL Developer, the visual tool for database development, simplifies the following tasks:

- Browsing and managing database objects
- Executing SQL statements and scripts
- Editing and debugging PL/SQL statements
- Creating reports

You can connect to any target Oracle database schema by using standard Oracle database authentication. When connected, you can perform operations on objects in the database.

The SQL Developer 1.2 release tightly integrates with *Developer Migration Workbench* that provides users with a single point to browse database objects and data in third-party databases, and to migrate from these databases to Oracle. You can also connect to schemas for selected third-party (non-Oracle) databases such as MySQL, Microsoft SQL Server, and Microsoft Access, and you can view the metadata and data in these databases.

Additionally, SQL Developer includes support for Oracle Application Express 3.0.1 (Oracle APEX).

Specifications of SQL Developer

- Is shipped along with Oracle Database 11*g* Release 2
- Is developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Provides default connectivity by using the JDBC Thin driver
- Connects to Oracle Database version 9.2.0.1 and later
- Is freely downloadable from the following link:
  - http://www.oracle.com/technology/products/database/sql_developer/index.html

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Specifications of SQL Developer

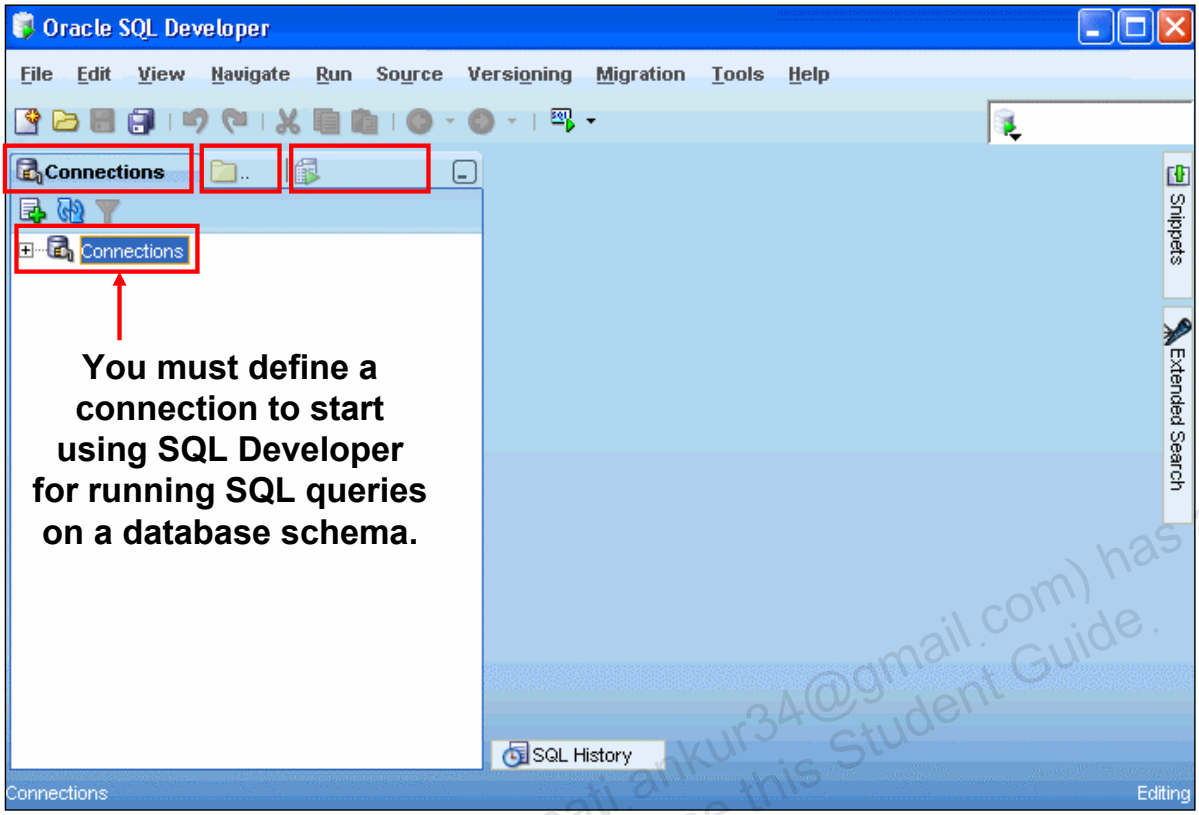
Oracle SQL Developer 1.5 is shipped along with Oracle Database 11*g* Release 2. SQL Developer is developed in Java and leverages the Oracle JDeveloper integrated development environment (IDE). Therefore, it is a cross-platform tool. The tool runs on the Windows, Linux, and Mac operating system (OS) X platforms.

Default connectivity to the database is through the Java Database Connectivity (JDBC) Thin driver, and therefore, no Oracle Home is required. SQL Developer does not require an installer. You only need to unzip the downloaded file. With SQL Developer, users can connect to Oracle Databases 9.2.0.1 and later, and all Oracle database editions, including Express Edition.

Note

- For Oracle Database versions before Oracle Database 11*g* Release 2, you will have to download and install SQL Developer. SQL Developer 1.5 is freely downloadable from the following link:
  - http://www.oracle.com/technology/products/database/sql_developer/index.html
- For instructions on how to install SQL Developer, you can visit the following link:
  - http://download.oracle.com/docs/cd/E12151_01/index.htm

SQL Developer 1.5 Interface



Copyright © 2009, Oracle. All rights reserved.

SQL Developer 1.5 Interface

The SQL Developer 1.5 interface contains three main navigation tabs, from left to right:

- **Connections tab:** By using this tab, you can browse database objects and users to which you have access.
- **Files tab:** Identified by the Files folder icon, this tab enables you to access files from your local machine without having to use the File > Open menu.
- **Reports tab:** Identified by the Reports icon, this tab enables you to run predefined reports or create and add your own reports.

General Navigation and Use

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about selected objects. You can customize many aspects of the appearance and behavior of SQL Developer by setting preferences.

Note: You need to define at least one connection to be able to connect to a database schema and issue SQL queries or run procedures and functions.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL Developer 1.5 Interface (continued)

Menus

The following menus contain standard entries, as well as entries for features that are specific to SQL Developer:

- **View:** Contains options that affect what is displayed in the SQL Developer interface
- **Navigate:** Contains options for navigating to panes and in the execution of subprograms
- **Run:** Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected, and also debugging options
- **Source:** Contains options for use when you edit functions and procedures
- **Versioning:** Provides integrated support for the following versioning and source control systems: Concurrent Versions System (CVS) and Subversion
- **Migration:** Contains options related to migrating third-party databases to Oracle
- **Tools:** Invokes SQL Developer tools such as SQL*Plus, Preferences, and SQL Worksheet

Note: The Run menu also contains options that are relevant when a function or procedure is selected for debugging. These are the same options found in the Debug menu in version 1.2.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating a Database Connection

- You must have at least one database connection to use SQL Developer.
- You can create and test connections for:
  - Multiple databases
  - Multiple schemas
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an Extensible Markup Language (XML) file.
- Each additional database connection created is listed in the Connections Navigator hierarchy.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating a Database Connection

A connection is a SQL Developer object that specifies the necessary information for connecting to a specific database as a specific user of that database. To use SQL Developer, you must have at least one database connection, which may be existing, created, or imported.

You can create and test connections for multiple databases and for multiple schemas.

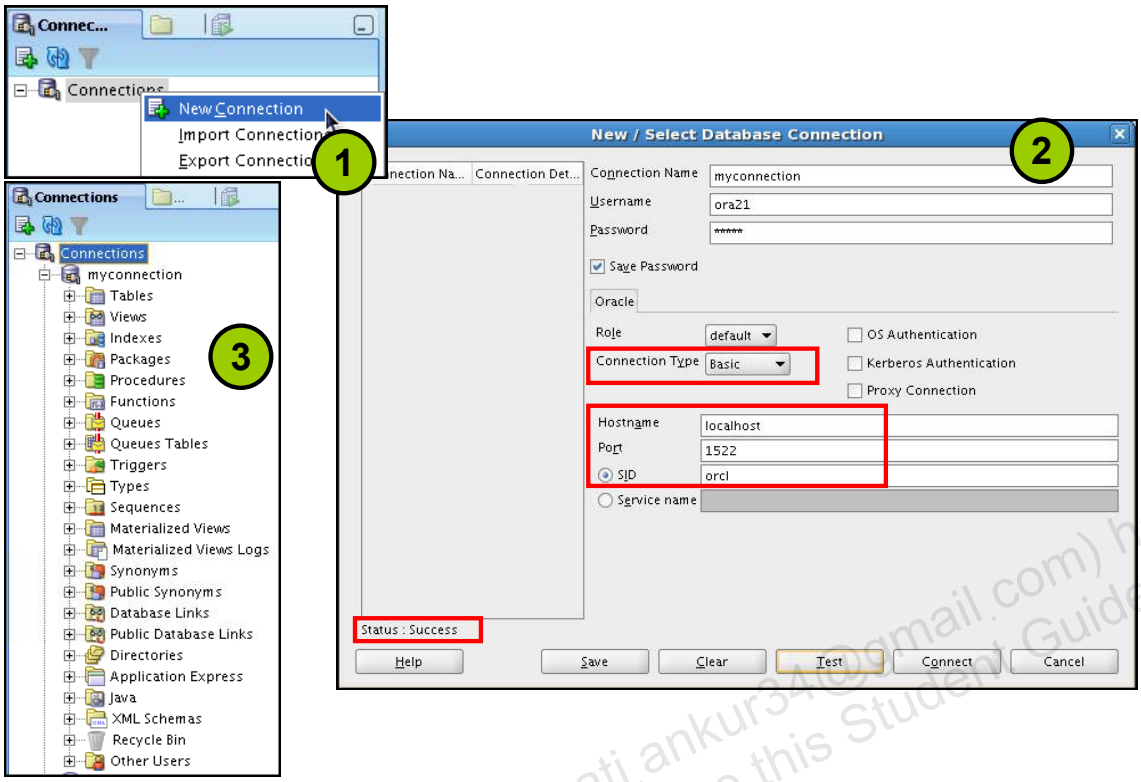
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory, but it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and display the Database Connections dialog box, SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.

Note: On Windows, if the `tnsnames.ora` file exists but its connections are not being used by SQL Developer, define `TNS_ADMIN` as a system environment variable.

You can export connections to an XML file so that you can reuse it later.

You can create additional connections as different users to the same database or to connect to different databases.

Creating a Database Connection



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Creating a Database Connection (continued)

To create a database connection, perform the following steps:

1. On the Connections tabbed page, right-click **Connections** and select **New Connection**.
2. In the New/Select Database Connection window, enter the connection name. Enter the username and password of the schema that you want to connect to.
   a) From the Role drop-down box, you can select either *default* or `SYSDBA` (you choose `SYSDBA` for the `sys` user or any user with database administrator privileges).
   b) You can select the connection type as:
      - **Basic:** In this type, enter the host name and SID for the database that you want to connect to. Port is already set to 1521. Or you can also choose to enter the Service name directly if you use a remote database connection.
      - **TNS:** You can select any one of the database aliases imported from the `tnsnames.ora` file.
      - **LDAP:** You can look up database services in Oracle Internet Directory, which is a component of Oracle Identity Management.
      - **Advanced:** You can define a custom Java Database Connectivity (JDBC) URL to connect to the database.
   c) Click **Test** to ensure that the connection has been set correctly.
   d) Click **Connect**.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating a Database Connection (continued)

If you select the Save Password check box, the password is saved to an XML file. So, after you close the SQL Developer connection and open it again, you are not prompted for the password.

3. The connection gets added in the Connections Navigator. You can expand the connection to view the database objects and object definitions, for example, dependencies, details, statistics, and so on.

Note: From the same New/Select Database Connection window, you can define connections to non-Oracle data sources using the Access, MySQL, and SQL Server tabs. However, these connections are read-only connections that enable you to browse objects and data in that data source.

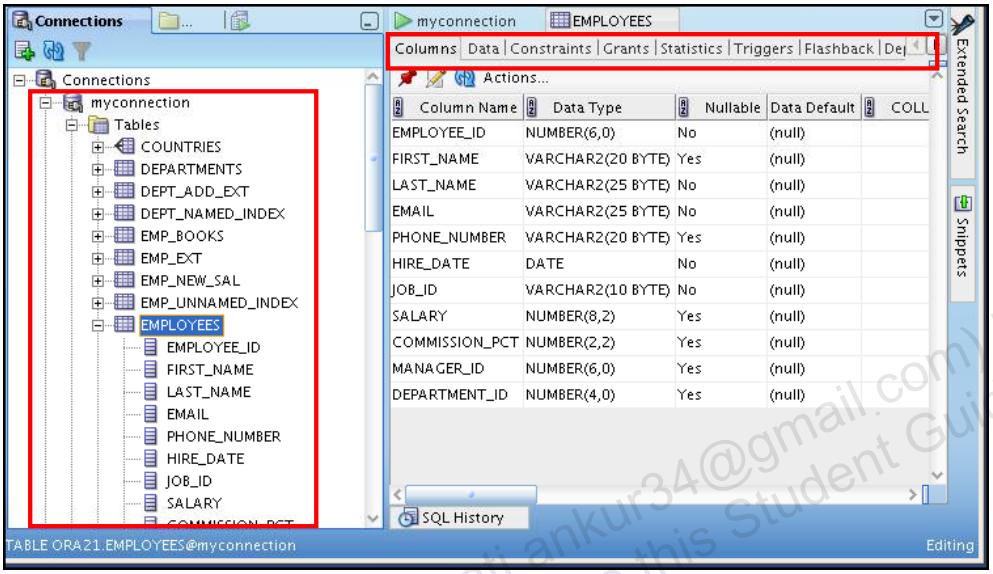
Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Browsing Database Objects

Use the Connections Navigator to:

- Browse through many objects in a database schema
- Review the definitions of objects at a glance



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Browsing Database Objects

After you create a database connection, you can use the Connections Navigator to browse through many objects in a database schema, including Tables, Views, Indexes, Packages, Procedures, Triggers, and Types.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about the selected objects. You can customize many aspects of the appearance of SQL Developer by setting preferences.

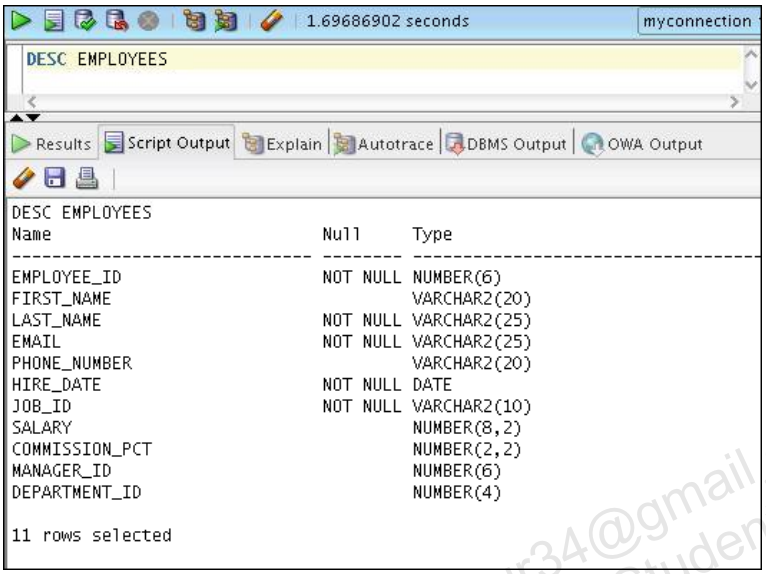
You can see the definition of objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, the details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

If you want to see the definition of the `EMPLOYEES` table as shown in the slide, perform the following steps:

1. Expand the Connections node in the Connections Navigator.
2. Expand Tables.
3. Click `EMPLOYEES`. By default, the Columns tab is selected. It shows the column description of the table. Using the Data tab, you can view the table data and also enter new rows, update data, and commit these changes to the database.

Displaying the Table Structure

Use the `DESCRIBE` command to display the structure of a table:



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

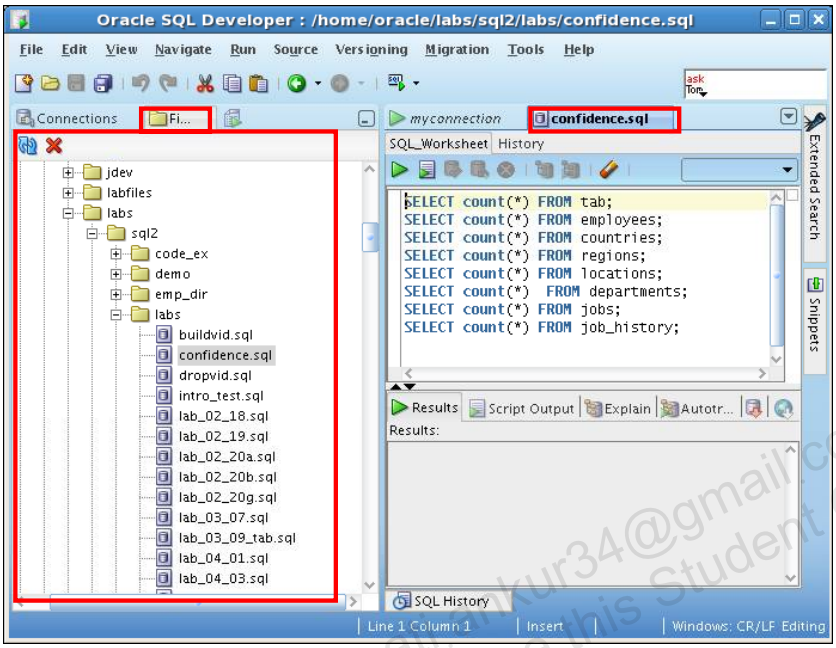
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Displaying the Table Structure

In SQL Developer, you can also display the structure of a table using the `DESCRIBE` command. The result of the command is a display of column names and data types, as well as an indication whether a column must contain data.

Browsing Files

Use the Files Navigator to explore the file system and open system files.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Browsing Database Objects

You can use the Files Navigator to browse and open system files.

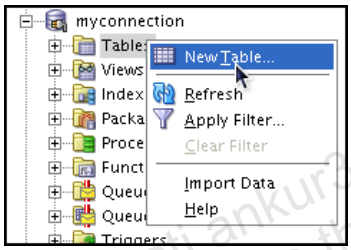
- To view the Files Navigator, click the Files tab, or select View > Files.
- To view the contents of a file, double-click a file name to display its contents in the SQL Worksheet area.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapatiankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating a Schema Object

- SQL Developer supports the creation of any schema object by:
  - Executing a SQL statement in SQL Worksheet
  - Using the context menu
- Edit the objects by using an edit dialog box or one of the many context-sensitive menus.
- View the data definition language (DDL) for adjustments such as creating a new object or editing an existing schema object.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (ankur34@gmail.com) has a non-transferable license to use this Student Guide.

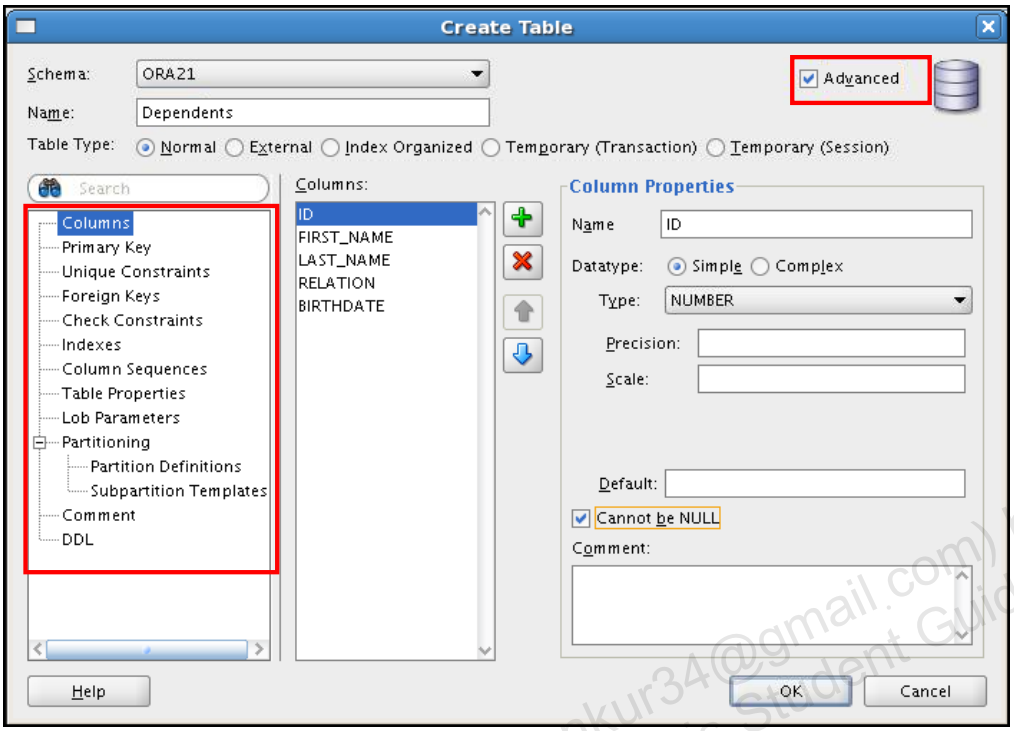
Creating a Schema Object

SQL Developer supports the creation of any schema object by executing a SQL statement in SQL Worksheet. Alternatively, you can create objects by using the context menu. When created, you can edit the objects by using an edit dialog box or one of the many context-sensitive menus.

As new objects are created or existing objects are edited, the DDL for those adjustments is available for review. An Export DDL option is available if you want to create the full DDL for one or more objects in the schema.

The slide shows how to create a table by using the context menu. To open a dialog box for creating a new table, right-click Tables and select New Table. The dialog boxes to create and edit database objects have multiple tabs, each reflecting a logical grouping of properties for that type of object.

Creating a New Table: Example



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Creating a New Table: Example

In the Create Table dialog box, if you do not select the Advanced check box, you can create a table quickly by specifying the columns and some frequently used features.

If you select the Advanced check box, the Create Table dialog box changes to one with multiple options, in which you can specify an extended set of features while you create the table.

The example in the slide shows how to create the `DEPENDENTS` table by selecting the Advanced check box.

To create a new table, perform the following steps:

1. In the Connections Navigator, right-click Tables.
2. Select New Table.
3. In the Create Table dialog box, select Advanced.
4. Specify column information.
5. Click OK.

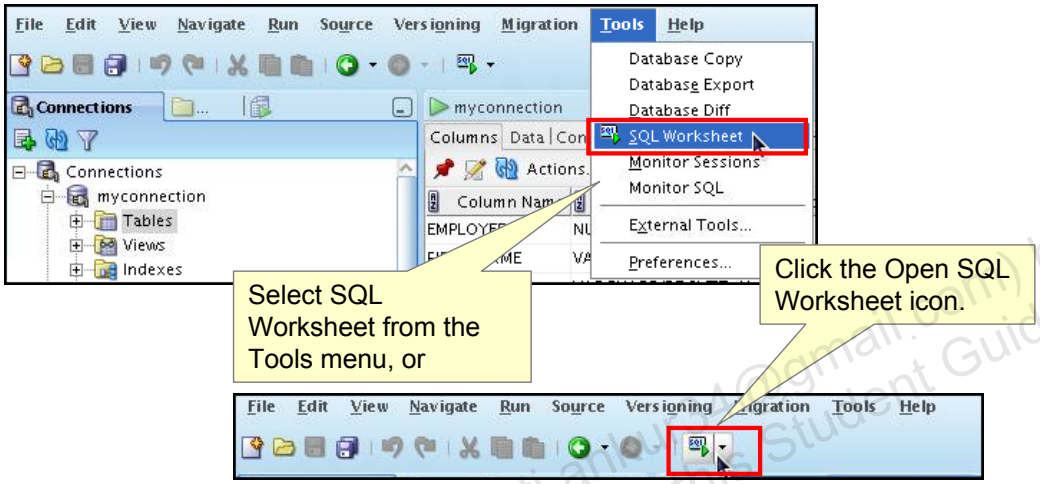
Although it is not required, you should also specify a primary key by using the Primary Key tab in the dialog box. Sometimes, you may want to edit the table that you created. To do so, right-click the table in the Connections Navigator and select Edit.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.



ORACLE
Copyright © 2009, Oracle. All rights reserved.

Using the SQL Worksheet

When you connect to a database, a SQL Worksheet window for that connection automatically opens. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. The SQL Worksheet supports SQL*Plus statements to a certain extent. However, SQL*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database.

You can specify the actions that can be processed by the database connection associated with the worksheet, such as:

- Creating a table
- Inserting data
- Creating and editing a trigger
- Selecting data from a table
- Saving the selected data to a file

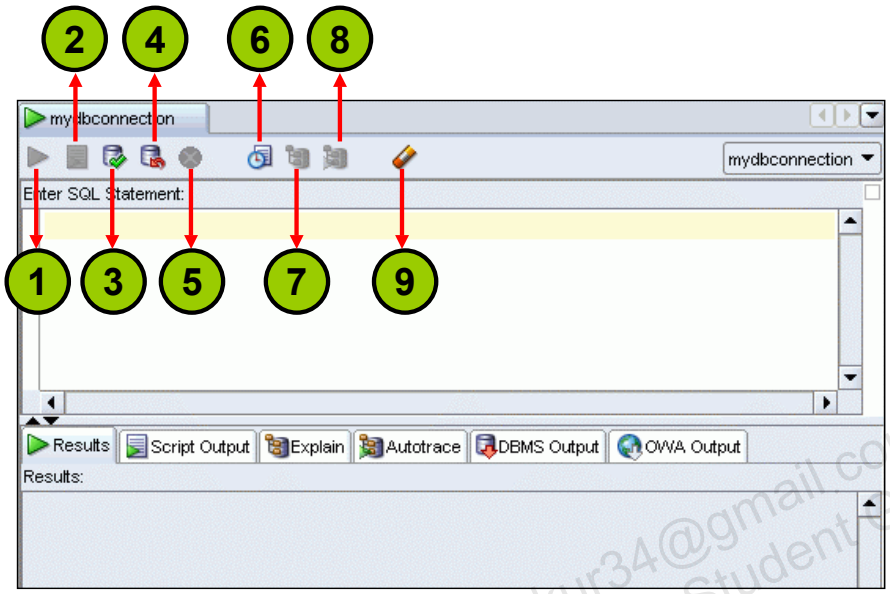
You can display a SQL Worksheet by using one of the following:

- Select Tools > SQL Worksheet.
- Click the Open SQL Worksheet icon.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (ankurm1@gmail.com) has a non-transferable license to use this Student Guide.

Using the SQL Worksheet



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

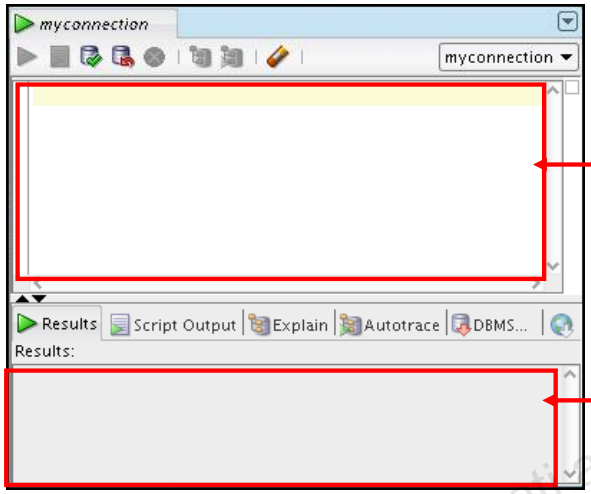
Using the SQL Worksheet (continued)

You may want to use the shortcut keys or icons to perform certain tasks such as executing a SQL statement, running a script, and viewing the history of SQL statements that you executed. You can use the SQL Worksheet toolbar that contains icons to perform the following tasks:

1. **Execute Statement:** Executes the statement where the cursor is located in the Enter SQL Statement window. You can use bind variables in the SQL statements, but not substitution variables.
2. **Run Script:** Executes all statements in the Enter SQL Statement window by using the Script Runner. You can use substitution variables in the SQL statements, but not bind variables.
3. **Commit:** Writes any changes to the database and ends the transaction
4. **Rollback:** Discards any changes to the database, without writing them to the database, and ends the transaction
5. **Cancel:** Stops the execution of any statements currently being executed
6. **SQL History:** Displays a dialog box with information about the SQL statements that you have executed
7. **Execute Explain Plan:** Generates the execution plan, which you can see by clicking the Explain tab
8. **Autotrace:** Generates trace information for the statement
9. **Clear:** Erases the statement or statements in the Enter SQL Statement window

Using the SQL Worksheet

- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.



ORACLE
Copyright © 2009, Oracle. All rights reserved.

Using the SQL Worksheet (continued)

When you connect to a database, a SQL Worksheet window for that connection automatically opens. You can use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. All SQL and PL/SQL commands are supported as they are passed directly from the SQL Worksheet to the Oracle database. But the SQL*Plus commands used in SQL Developer have to be interpreted by the SQL Worksheet before being passed to the database.

The SQL Worksheet currently supports a number of SQL*Plus commands. However, commands that are not supported by the SQL Worksheet are ignored and are not sent to the Oracle database. Through the SQL Worksheet, you can execute SQL statements and some of the SQL*Plus commands.

You can display a SQL Worksheet by using either of the following two options:

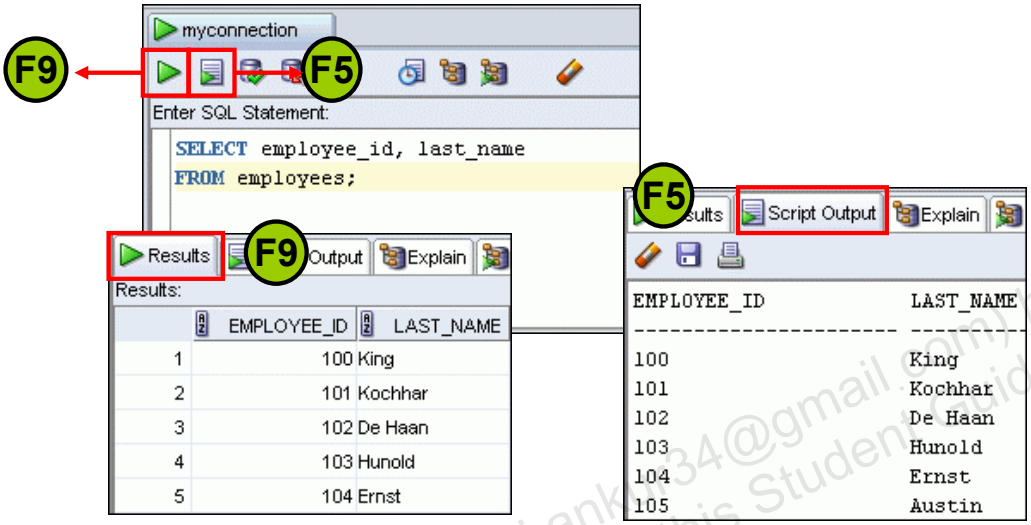
- Select Tools > SQL Worksheet.
- Click the Open SQL Worksheet icon.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapatiankur04@gmail.com) has a non-transferable license to use this Student Guide.

Executing SQL Statements

Use the Enter SQL Statement window to enter single or multiple SQL statements.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

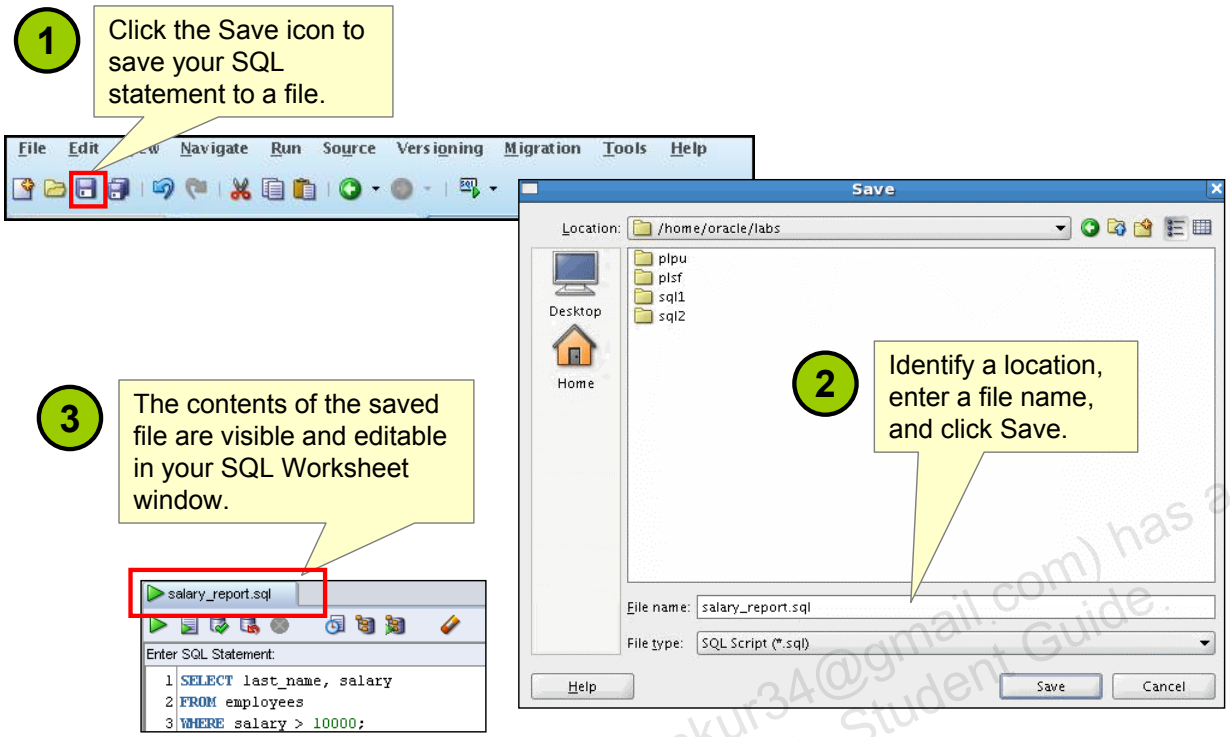
Executing SQL Statements

The example in the slide shows the difference in output for the same query when the F9 key or Execute Statement is used versus when F5 or Run Script is used.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Saving SQL Scripts



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Saving SQL Scripts

You can save your SQL statements from the SQL Worksheet into a text file. To save the contents of the Enter SQL Statement window, perform the following steps:

1. Click the Save icon or use the File > Save menu option.
2. In the Windows Save dialog box, enter a file name and the location where you want the file saved.
3. Click Save.

After you save the contents to a file, the Enter SQL Statement window displays a tabbed page of your file contents. You can have multiple files open at the same time. Each file displays as a tabbed page.

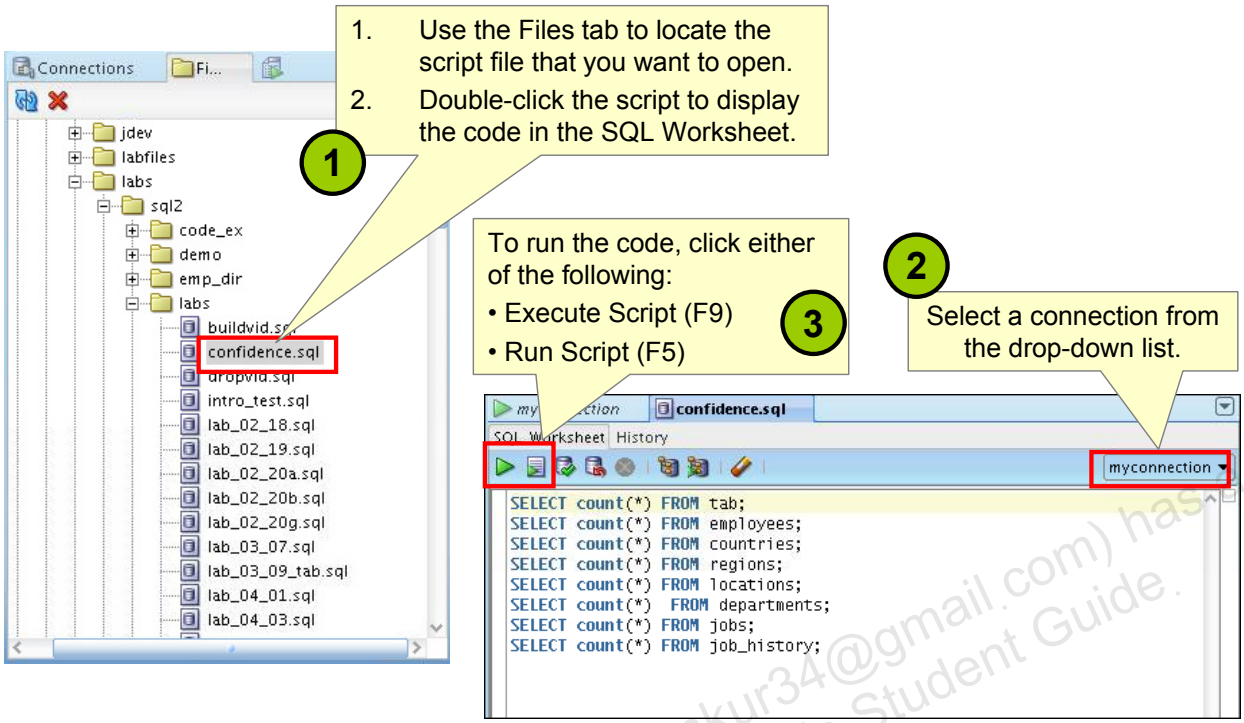
Script Pathing

You can select a default path to look for scripts and to save scripts. Under Tools > Preferences > Database > Worksheet Parameters, enter a value in the "Select default path to look for scripts" field.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Executing Saved Script Files: Method 1



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Executing Saved Script Files: Method 1

To open a script file and display the code in the SQL Worksheet area, perform the following:

1. In the files navigator, select (or navigate to) the script file that you want to open.
2. Double-click to open. The code of the script file is displayed in the SQL Worksheet area.
3. Select a connection from the connection drop-down list.
4. To run the code, click the Run Script (F5) icon on the SQL Worksheet toolbar. If you have not selected a connection from the connection drop-down list, a connection dialog box will appear. Select the connection that you want to use for the script execution.

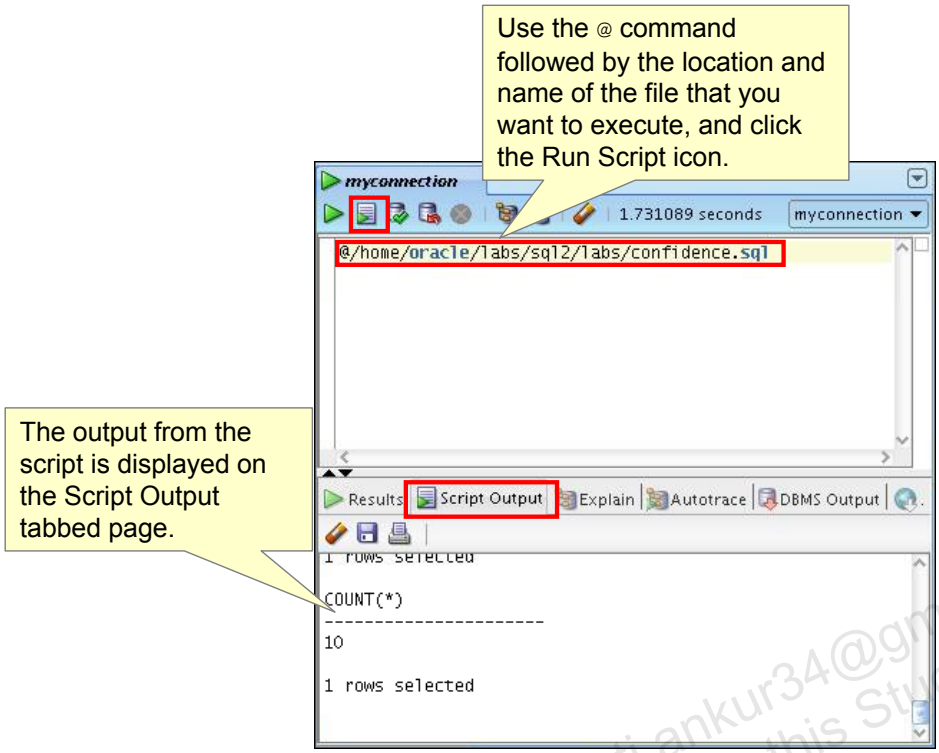
Alternatively, you can also perform the following:

1. Select File > Open. The Open dialog box appears.
2. In the Open dialog box, select (or navigate to) the script file that you want to open.
3. Click Open. The code of the script file is displayed in the SQL Worksheet area.
4. Select a connection from the connection drop-down list.
5. To run the code, click the Run Script (F5) icon on the SQL Worksheet toolbar. If you have not selected a connection from the connection drop-down list, a connection dialog box will appear. Select the connection that you want to use for the script execution.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Executing Saved Script Files: Method 2



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Executing Saved Script Files: Method 2

To run a saved SQL script, perform the following:

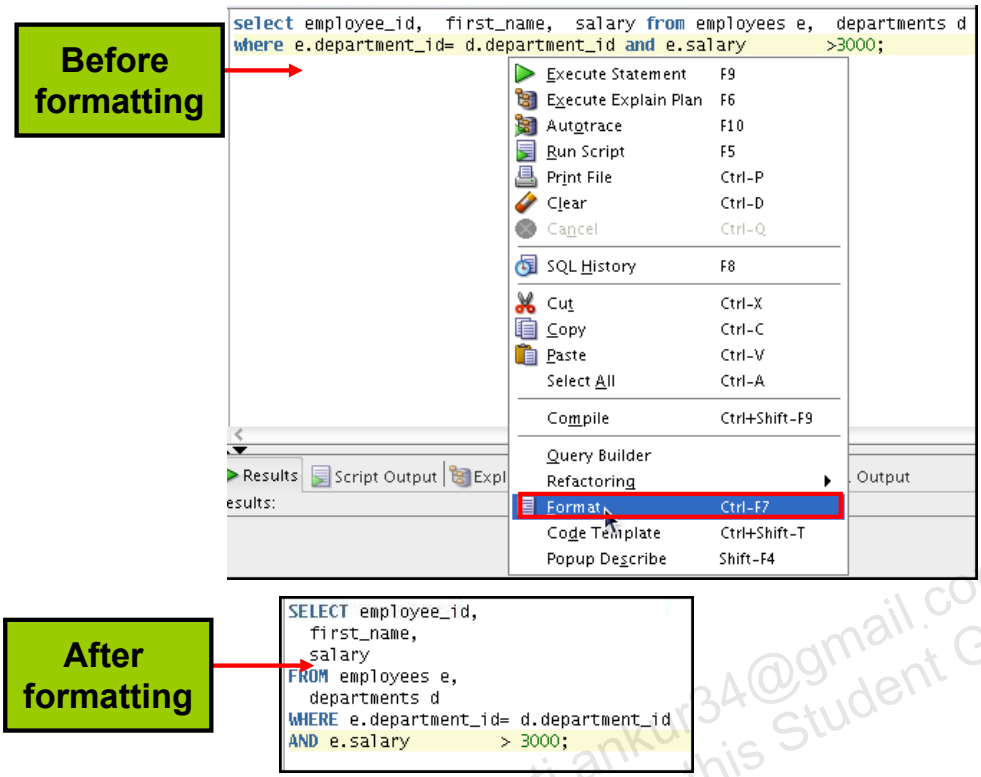
1. Use the **@** command followed by the location and name of the file that you want to run, in the Enter SQL Statement window.
2. Click the Run Script icon.

The results from running the file are displayed on the Script Output tabbed page. You can also save the script output by clicking the Save icon on the Script Output tabbed page. The Windows File Save dialog box appears and you can identify a name and location for your file.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Formatting the SQL Code



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Formatting the SQL Code

You may want to beautify the indentation, spacing, capitalization, and line separation of the SQL code. SQL Developer has a feature for formatting SQL code.

To format the SQL code, right-click in the statement area and select Format SQL.

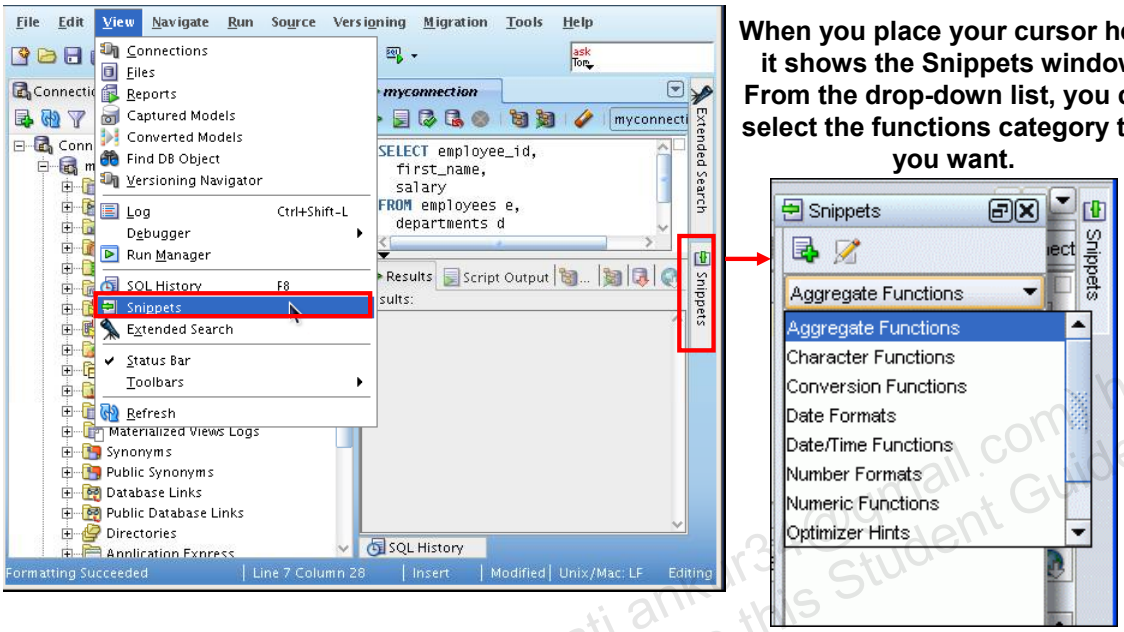
In the example in the slide, before formatting, the SQL code has the keywords not capitalized and the statement not properly indented. After formatting, the SQL code is beautified with the keywords capitalized and the statement properly indented.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Using Snippets

Snippets are code fragments, such as syntax or examples.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Using Snippets

You may want to use certain code fragments when you use the SQL Worksheet, or create or edit a PL/SQL function or procedure. SQL Developer has a feature called Snippets, which are code fragments such as SQL functions, Optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag Snippets into the Editor window.

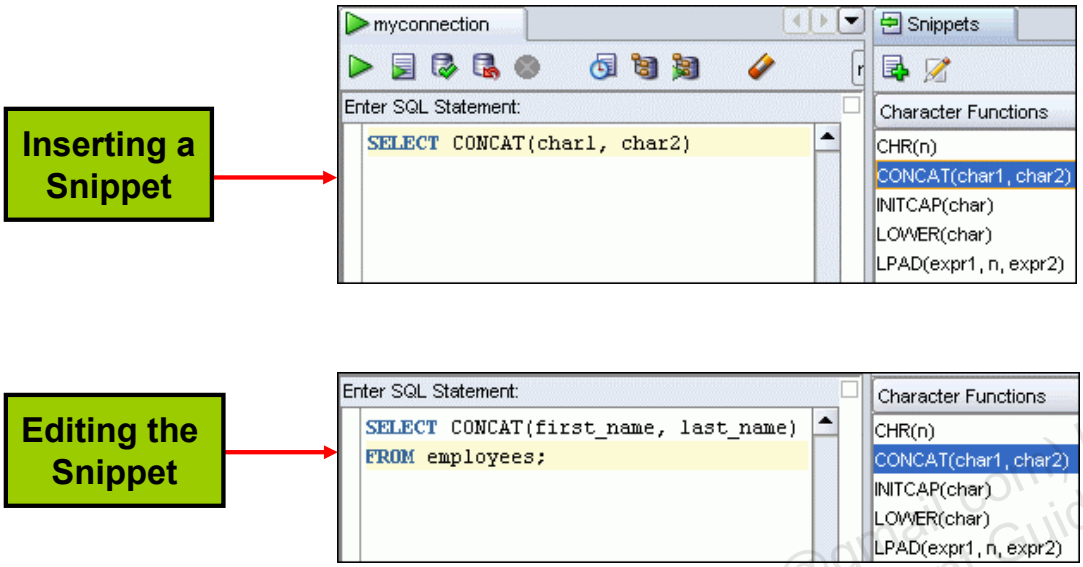
To display Snippets, select View > Snippets.

The Snippets window is displayed at the right. You can use the drop-down list to select a group. A Snippets button is placed in the right window margin, so that you can display the Snippets window if it becomes hidden.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur3@gmail.com) has a non-transferable license to use this Student Guide.

Using Snippets: Example



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Using Snippets: Example

To insert a Snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the Snippet from the Snippets window into the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function in a tool tip, place the cursor over the function name.

The example in the slide shows that `CONCAT(char1, char2)` is dragged from the Character Functions group in the Snippets window. Then the `CONCAT` function syntax is edited and the rest of the statement is added as follows:

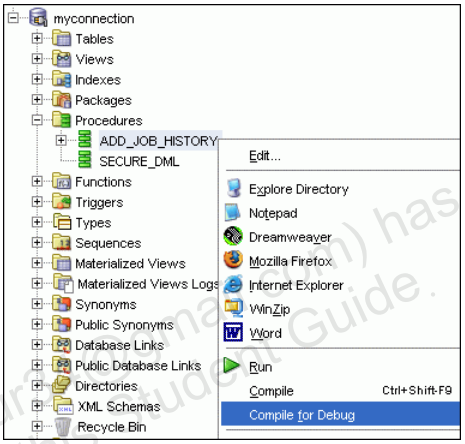
```
SELECT CONCAT(first_name, last_name)
FROM employees;
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Debugging Procedures and Functions

- Use SQL Developer to debug PL/SQL functions and procedures.
- Use the Compile for Debug option to perform a PL/SQL compilation so that the procedure can be debugged.
- Use Debug menu options to set breakpoints, and to perform step into and step over tasks.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur@humberlearning.com) has a non-transferable license to use this Student Guide.

Debugging Procedures and Functions

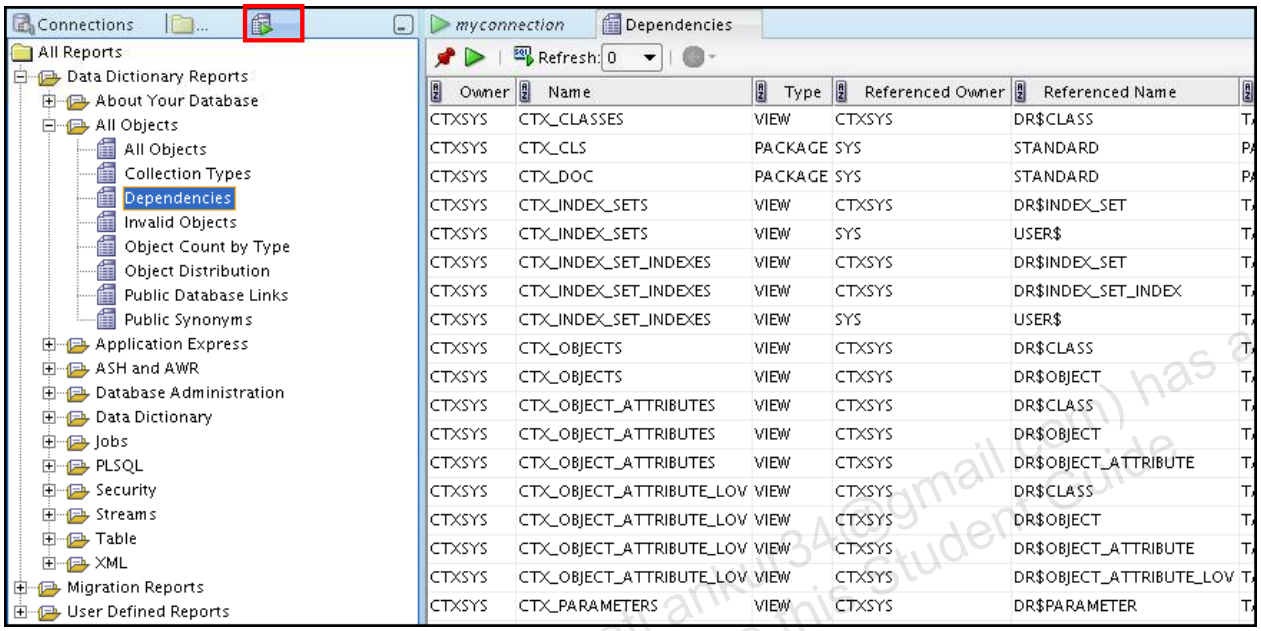
In SQL Developer, you can debug PL/SQL procedures and functions. Using the Debug menu options, you can perform the following debugging tasks:

- **Find Execution Point** goes to the next execution point.
- **Resume** continues execution.
- **Step Over** bypasses the next method and goes to the next statement after the method.
- **Step Into** goes to the first statement in the next method.
- **Step Out** leaves the current method and goes to the next statement.
- **Step to End of Method** goes to the last statement of the current method.
- **Pause** halts execution but does not exit, thus allowing you to resume execution.
- **Terminate** halts and exits the execution. You cannot resume execution from this point; instead, to start running or debugging from the beginning of the function or procedure, click the Run or Debug icon on the Source tab toolbar.
- **Garbage Collection** removes invalid objects from the cache in favor of more frequently accessed and more valid objects.

These options are also available as icons on the debugging toolbar.

Database Reporting

SQL Developer provides a number of predefined reports about the database and its objects.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this student guide

Database Reporting

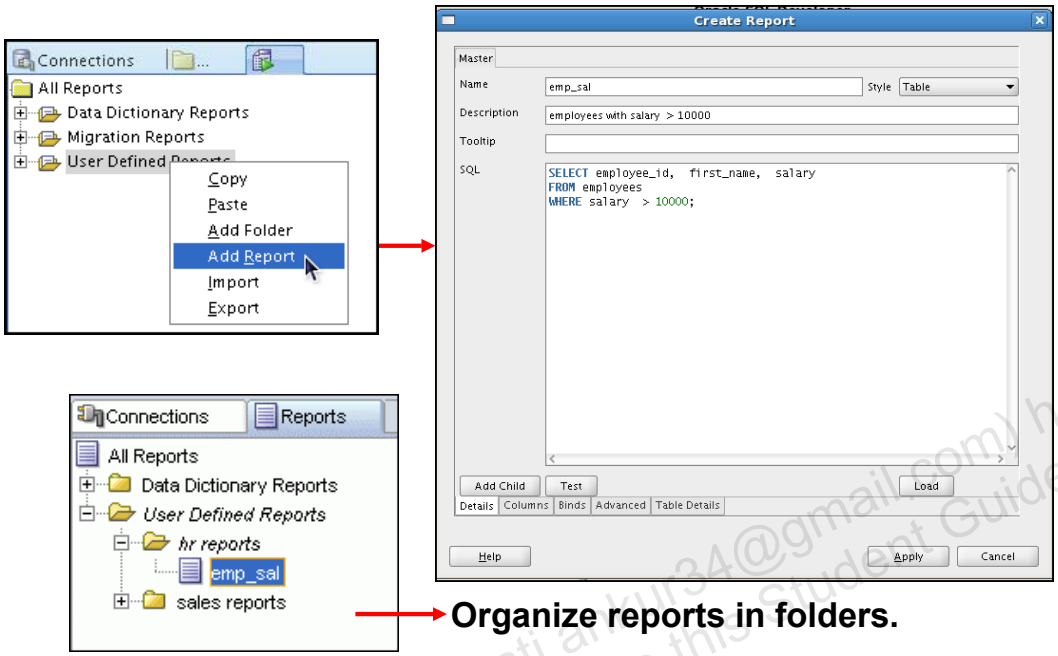
SQL Developer provides many reports about the database and its objects. These reports can be grouped into the following categories:

- About Your Database reports
- Database Administration reports
- Table reports
- PL/SQL reports
- Security reports
- XML reports
- Jobs reports
- Streams reports
- All Objects reports
- Data Dictionary reports
- User-defined reports

To display reports, click the Reports tab to the left of the window. Individual reports are displayed in tabbed panes at the right of the window. For each report, you can select (using a drop-down list) the database connection for which to display the report. For reports about objects, the objects shown are only those visible to the database user associated with the selected database connection, and the rows are usually ordered by Owner. You can also create your own user-defined reports.

Creating a User-Defined Report

Create and save user-defined reports for repeated use.



Organize reports in folders.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating a User-Defined Report

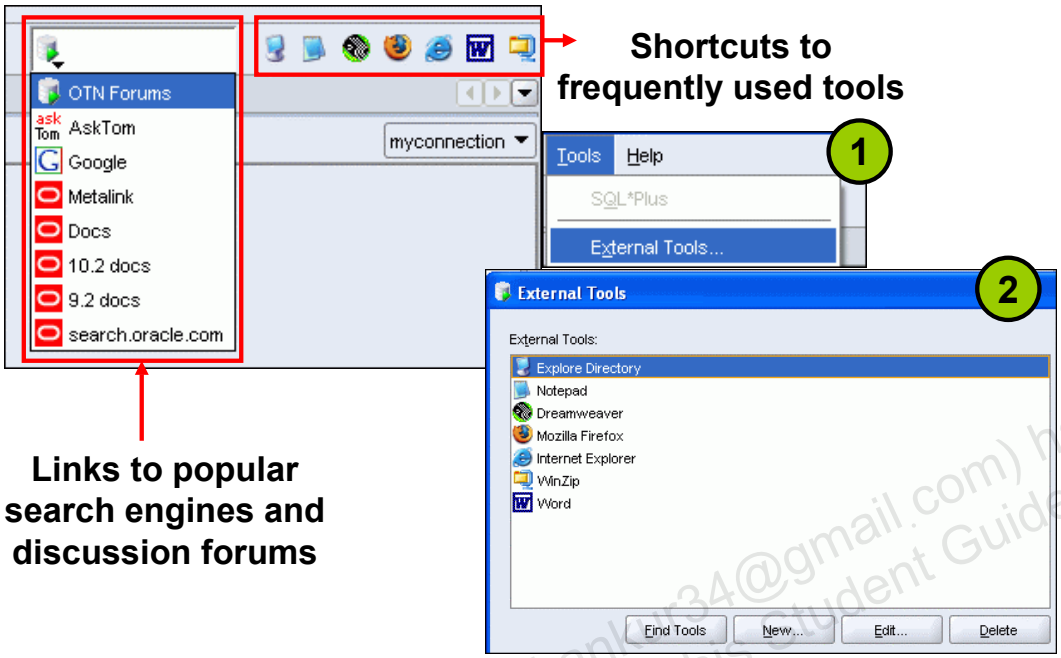
User-defined reports are reports created by SQL Developer users. To create a user-defined report, perform the following steps:

1. Right-click the User Defined Reports node under Reports and select Add Report.
2. In the Create Report Dialog box, specify the report name and the SQL query to retrieve information for the report. Then, click Apply.

In the example in the slide, the report name is specified as `emp_sal`. An optional description is provided indicating that the report contains details of employees with `salary >= 10000`. The complete SQL statement for retrieving the information to be displayed in the user-defined report is specified in the SQL box. You can also include an optional tool tip to be displayed when the cursor hovers over the report name in the Reports navigator display.

You can organize user-defined reports in folders, and you can create a hierarchy of folders and subfolders. To create a folder for user-defined reports, right-click the User Defined Reports node or any folder name under that node and select Add Folder. Information about user-defined reports, including any folders for these reports, is stored in a file named `UserReports.xml` under the directory for user-specific information.

Search Engines and External Tools



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Search Engines and External Tools

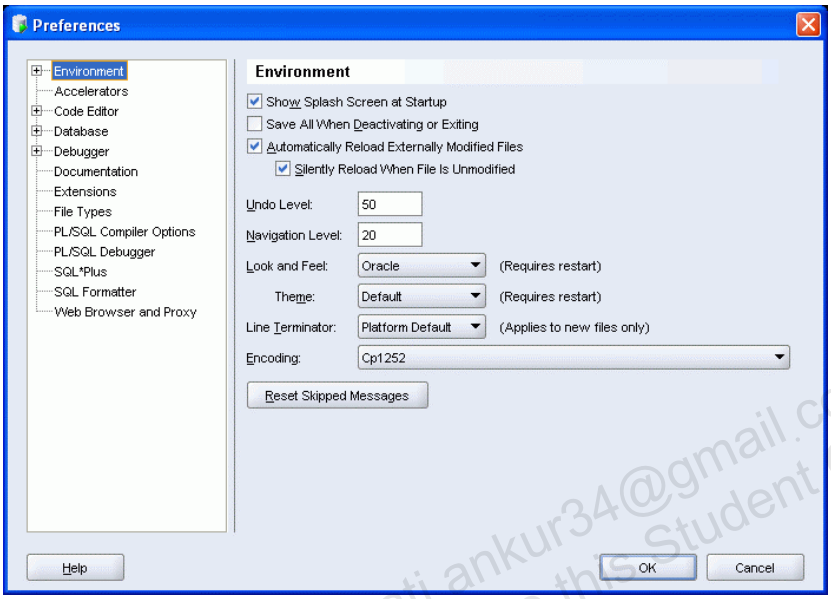
To enhance the productivity of SQL developers, SQL Developer has added quick links to popular search engines and discussion forums such as AskTom, Google, and so on. Also, you have shortcut icons to some of the frequently used tools, such as Notepad, Microsoft Word, and Dreamweaver, available to you.

You can add external tools to the existing list or even delete shortcuts to tools that you do not use frequently. To do so, perform the following:

1. From the Tools menu, select External Tools.
2. In the External Tools dialog box, select New to add new tools. Select Delete to remove any tool from the list.

Setting Preferences

- Customize the SQL Developer interface and environment.
- From the Tools menu, select Preferences.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Setting Preferences

You can customize many aspects of the SQL Developer interface and environment by modifying the SQL Developer preferences according to your requirements. To modify the SQL Developer preferences, select Tools, and then Preferences.

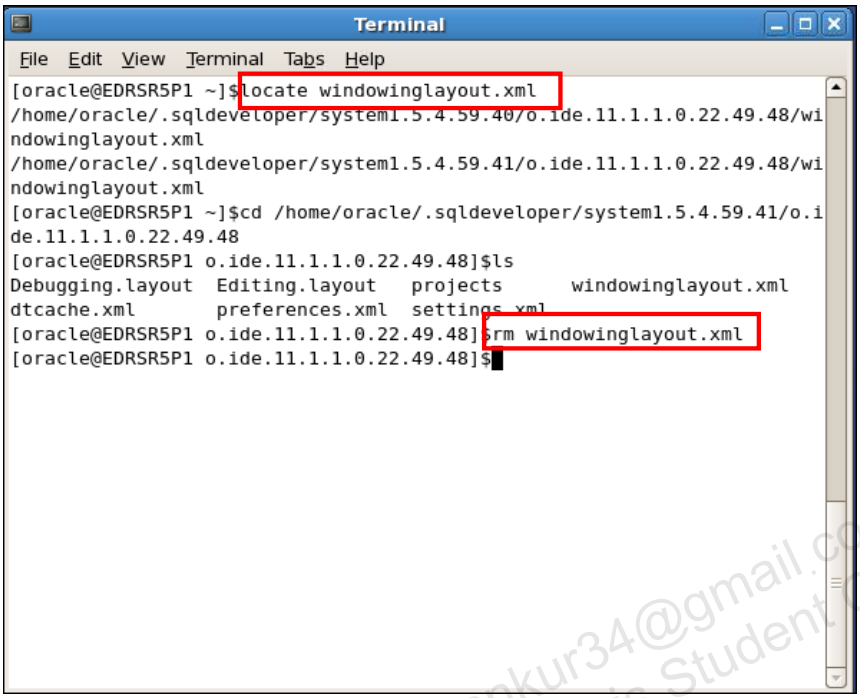
The preferences are grouped into the following categories:

- Environment
- Accelerators (Keyboard shortcuts)
- Code Editors
- Database
- Debugger
- Documentation
- Extensions
- File Types
- Migration
- PL/SQL Compilers
- PL/SQL Debugger, and so on

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Resetting the SQL Developer Layout



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Resetting the SQL Developer Layout

While working with SQL Developer, if the Connections Navigator disappears or if you cannot dock the Log window in its original place, perform the following steps to fix the problem:

1. Exit SQL Developer.
2. Open a terminal window and use the locate command to find the location of `windowinglayout.xml`.
3. Go to the directory that has `windowinglayout.xml` and delete it.
4. Restart SQL Developer.

Summary

In this appendix, you should have learned how to use SQL Developer to do the following:

- Browse, create, and edit database objects
- Execute SQL statements and scripts in SQL Worksheet
- Create and save custom reports

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Summary

SQL Developer is a free graphical tool to simplify database development tasks. Using SQL Developer, you can browse, create, and edit database objects. You can use SQL Worksheet to run SQL statements and scripts. SQL Developer enables you to create and save your own special set of reports for repeated use.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Humber College

D

Using SQL*Plus

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

After completing this appendix, you should be able to do the following:

- Log in to SQL*Plus
- Edit SQL commands
- Format the output using SQL*Plus commands
- Interact with script files

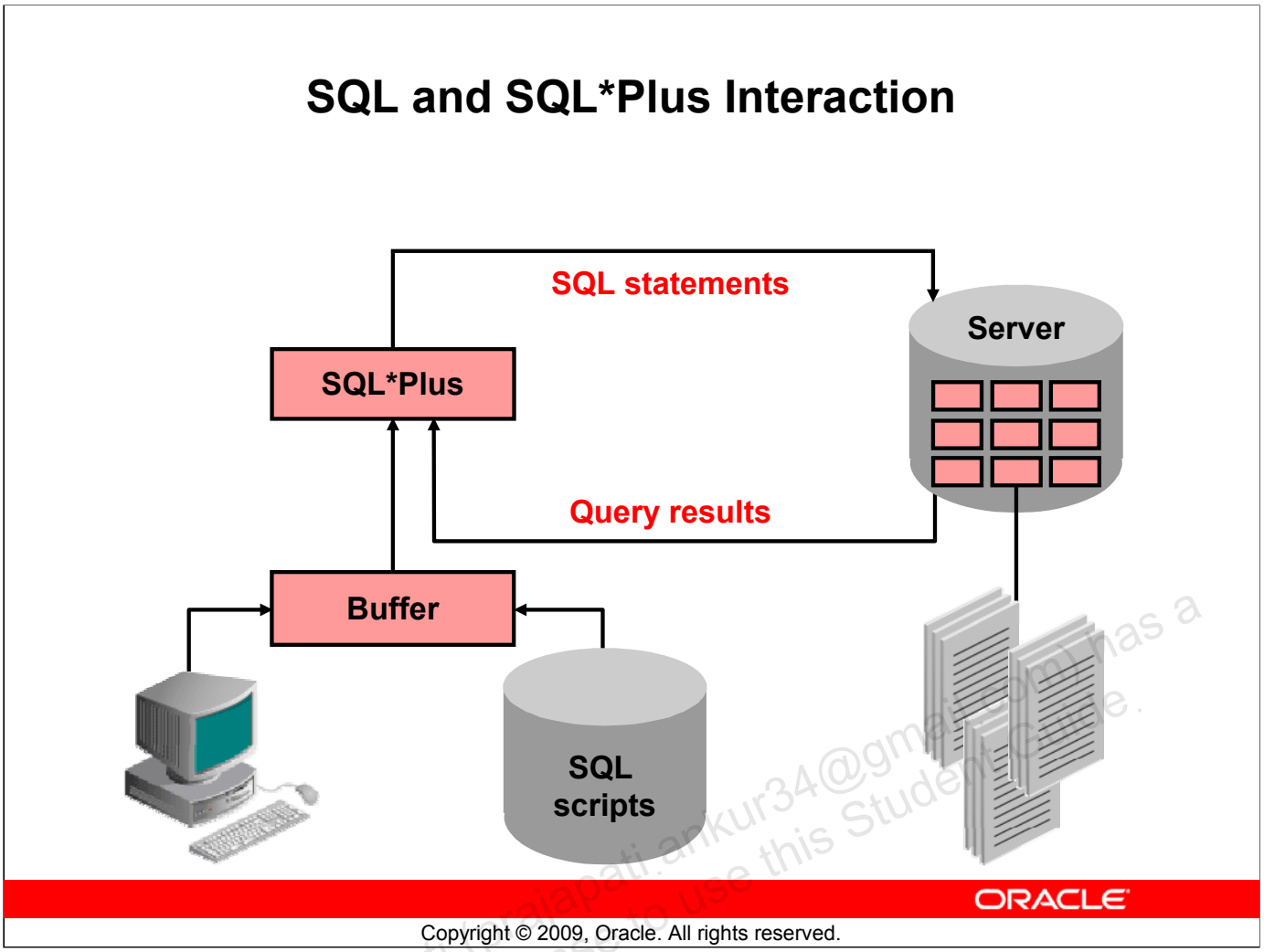
ORACLE
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Objectives

You might want to create `SELECT` statements that can be used again and again. This appendix also covers the use of SQL*Plus commands to execute SQL statements. You learn how to format output using SQL*Plus commands, edit SQL commands, and save scripts in SQL*Plus.



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL and SQL*Plus

SQL is a command language that is used for communication with the Oracle Server from any tool or application. Oracle SQL contains many extensions. When you enter a SQL statement, it is stored in a part of memory called the *SQL buffer* and remains there until you enter a new SQL statement. SQL*Plus is an Oracle tool that recognizes and submits SQL statements to the Oracle9*i* Server for execution. It contains its own command language.

Features of SQL

- Can be used by a range of users, including those with little or no programming experience
- Is a nonprocedural language
- Reduces the amount of time required for creating and maintaining systems
- Is an English-like language

Features of SQL*Plus

- Accepts ad hoc entry of statements
- Accepts SQL input from files
- Provides a line editor for modifying SQL statements
- Controls environmental settings
- Formats query results into basic reports
- Accesses local and remote databases

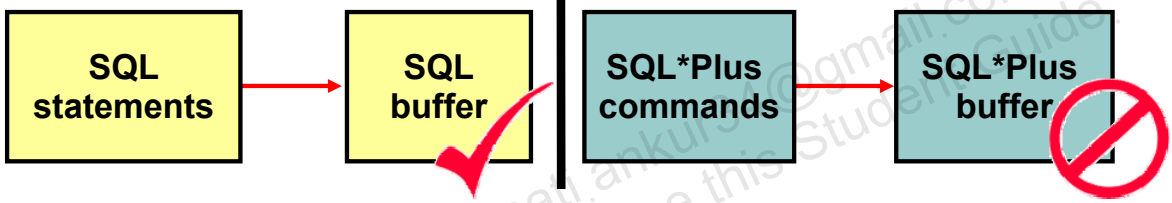
SQL Statements Versus SQL*Plus Commands

SQL

- A language
- ANSI-standard
- Keywords cannot be abbreviated.
- Statements manipulate data and table definitions in the database.

SQL*Plus

- An environment
- Oracle-proprietary
- Keywords can be abbreviated.
- Commands do not allow manipulation of values in the database.



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL and SQL*Plus (continued)

The following table compares SQL and SQL*Plus:

SQL	SQL*Plus
Is a language for communicating with the Oracle server to access data	Recognizes SQL statements and sends them to the server
Is based on American National Standards Institute (ANSI)–standard SQL	Is the Oracle-proprietary interface for executing SQL statements
Manipulates data and table definitions in the database	Does not allow manipulation of values in the database
Is entered into the SQL buffer on one or more lines	Is entered one line at a time, not stored in the SQL buffer
Does not have a continuation character	Uses a dash (–) as a continuation character if the command is longer than one line
Cannot be abbreviated	Can be abbreviated
Uses a termination character to execute commands immediately	Does not require termination characters; executes commands immediately
Uses functions to perform some formatting	Uses commands to format data

SQL*Plus: Overview

- Log in to SQL*Plus.
- Describe the table structure.
- Edit your SQL statement.
- Execute SQL from SQL*Plus.
- Save SQL statements to files and append SQL statements to files.
- Execute saved files.
- Load commands from the file to the buffer to edit.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL*Plus

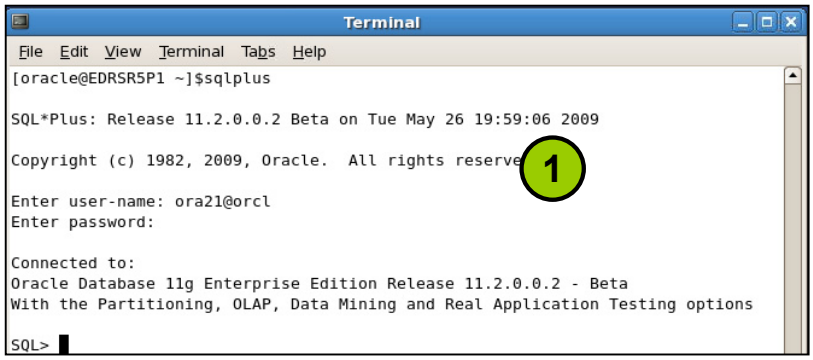
SQL*Plus is an environment in which you can:

- Execute SQL statements to retrieve, modify, add, and remove data from the database
- Format, perform calculations on, store, and print query results in the form of reports
- Create script files to store SQL statements for repeated use in the future

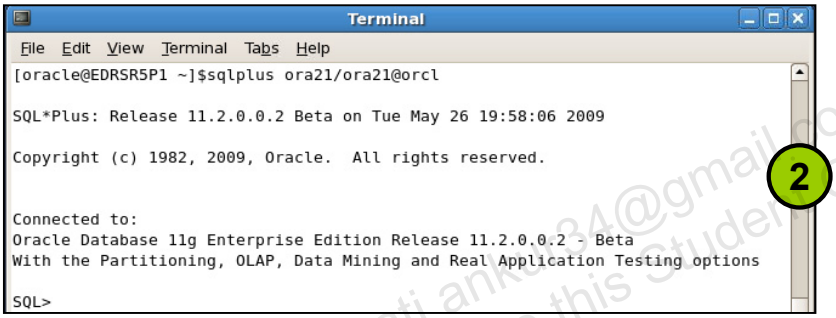
SQL*Plus commands can be divided into the following main categories:

Category	Purpose
Environment	Affect the general behavior of SQL statements for the session
Format	Format query results
File manipulation	Save, load, and run script files
Execution	Send SQL statements from the SQL buffer to the Oracle server
Edit	Modify SQL statements in the buffer
Interaction	Create and pass variables to SQL statements, print variable values, and print messages to the screen
Miscellaneous	Connect to the database, manipulate the SQL*Plus environment, and display column definitions

Logging In to SQL*Plus




```
sqlplus [username[/password[@database]]]
```



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur.34@gmail.com) has a non-transferable license to use this Student Guide.

Logging In to SQL*Plus

How you invoke SQL*Plus depends on the type of operating system that you are running Oracle Database on.

To log in from a Linux environment, perform the following steps:

1. Right-click on your Linux desktop and select terminal.
2. Enter the `sqlplus` command shown in the slide.
3. Enter the username, password, and database name.

In the syntax:

`username`	Is your database username
`password`	Is your database password (Your password is visible if you enter it here.)
`@database`	Is the database connect string

Note: To ensure the integrity of your password, do not enter it at the operating system prompt. Instead, enter only your username. Enter your password at the password prompt.

Displaying the Table Structure

Use the SQL*Plus `DESCRIBE` command to display the structure of a table:

```
DESC[RIBE] tablename
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Displaying the Table Structure

In SQL*Plus, you can display the structure of a table by using the `DESCRIBE` command. The result of the command is a display of column names and data types, as well as an indication whether a column must contain data.

In the syntax:

`tablename` Is the name of any existing table, view, or synonym that is accessible to the user

To describe the `DEPARTMENTS` table, use this command:

```
SQL> DESCRIBE DEPARTMENTS
Name                      Null?    Type
------------------------- -------- ---------------

DEPARTMENT_ID             NOT NULL NUMBER(4)
DEPARTMENT_NAME           NOT NULL VARCHAR2(30)
MANAGER_ID                NUMBER(6)
LOCATION_ID               NUMBER(4)
```

Displaying the Table Structure

```
DESCRIBE departments
```

```
Name                      Null?    Type
------------------------- -------- ------------
DEPARTMENT_ID             NOT NULL NUMBER(4)
DEPARTMENT_NAME           NOT NULL VARCHAR2(30)
MANAGER_ID                NUMBER(6)
LOCATION_ID               NUMBER(4)
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Displaying the Table Structure (continued)

The example in the slide displays information about the structure of the `DEPARTMENTS` table. In the result:

- `Null?`: Specifies whether a column must contain data (`NOT NULL` indicates that a column must contain data.)
- `Type`: Displays the data type for a column

SQL*Plus Editing Commands

- A[PPEND] *text*
- C[HANGE] / *old* / *new*
- C[HANGE] / *text* /
- CL[EAR] BUFF[ER]
- DEL
- DEL *n*
- DEL *m n*

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL*Plus Editing Commands

SQL*Plus commands are entered one line at a time, and are not stored in the SQL buffer.

Command	Description
A[PPEND] *text*	Adds text to the end of the current line
C[HANGE] / *old* / *new*	Changes *old* text to *new* in the current line
C[HANGE] / *text* /	Deletes *text* from the current line
CL[EAR] BUFF[ER]	Deletes all lines from the SQL buffer
DEL	Deletes the current line
DEL *n*	Deletes line *n*
DEL *m n*	Deletes lines *m* to *n* inclusive

Guidelines

- If you press Enter before completing a command, SQL*Plus prompts you with a line number.
- You terminate the SQL buffer either by entering one of the terminator characters (semicolon or slash) or by pressing Enter twice. The SQL prompt then appears.

SQL*Plus Editing Commands

- `I[NPUT]`
- `I[NPUT] text`
- `L[IST]`
- `L[IST] n`
- `L[IST] m n`
- `R[UN]`
- `n`
- `n text`
- `0 text`

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Prof. M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL*Plus Editing Commands (continued)

Command	Description
`I[NPUT]`	Inserts an indefinite number of lines
`I[NPUT] text`	Inserts a line consisting of *text*
`L[IST]`	Lists all lines in the SQL buffer
`L[IST] n`	Lists one line (specified by *n*)
`L[IST] m n`	Lists a range of lines (*m* to *n* inclusive)
`R[UN]`	Displays and runs the current SQL statement in the buffer
n	Specifies the line to make the current line
n text	Replaces line *n* with *text*
`0 text`	Inserts a line before line 1

Note: You can enter only one SQL*Plus command for each SQL prompt. SQL*Plus commands are not stored in the buffer. To continue a SQL*Plus command on the next line, end the first line with a hyphen (-).

Using LIST, n, and APPEND

```
LIST
  1  SELECT last_name
  2* FROM   employees
```

```
1
  1* SELECT last_name
```

```
A , job_id
  1* SELECT last_name, job_id
```

```
LIST
  1  SELECT last_name, job_id
  2* FROM   employees
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Using LIST, n, and APPEND

- Use the `L[IST]` command to display the contents of the SQL buffer. The asterisk (`*`) beside line 2 in the buffer indicates that line 2 is the current line. Any edits that you made apply to the current line.
- Change the number of the current line by entering the number (`n`) of the line that you want to edit. The new current line is displayed.
- Use the `A[PPEND]` command to add text to the current line. The newly edited line is displayed. Verify the new contents of the buffer by using the `LIST` command.

Note: Many SQL*Plus commands, including `LIST` and `APPEND`, can be abbreviated to only their first letters. `LIST` can be abbreviated to `L`; `APPEND` can be abbreviated to `A`.

Using the CHANGE Command

```
LIST
  1* SELECT * from employees
```

```
c/employees/departments
  1* SELECT * from departments
```

```
LIST
  1* SELECT * from departments
```

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Using the CHANGE Command

- Use `L[IST]` to display the contents of the buffer.
- Use the `C[HANGE]` command to alter the contents of the current line in the SQL buffer. In this case, replace the `employees` table with the `departments` table. The new current line is displayed.
- Use the `L[IST]` command to verify the new contents of the buffer.

SQL*Plus File Commands

- SAVE *filename*
- GET *filename*
- START *filename*
- @ *filename*
- EDIT *filename*
- SPOOL *filename*
- EXIT

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

SQL*Plus File Commands

SQL statements communicate with the Oracle Server. SQL*Plus commands control the environment, format query results, and manage files. You can use the commands described in the following table:

Command	Description
SAV[E] *filename* [.ext] [REP[LACE]APP[END]]	Saves the current contents of the SQL buffer to a file. Use APPEND to add to an existing file; use REPLACE to overwrite an existing file. The default extension is .sql.
GET *filename* [.ext]	Writes the contents of a previously saved file to the SQL buffer. The default extension for the file name is .sql.
STA[RT] *filename* [.ext]	Runs a previously saved command file
@ *filename*	Runs a previously saved command file (same as START)
ED[IT]	Invokes the editor and saves the buffer contents to a file named afiedt.buf
ED[IT] [*filename*[.ext]]	Invokes the editor to edit the contents of a saved file
SPO[OL] [*filename*[.ext]| OFF|OUT]	Stores query results in a file. OFF closes the spool file. OUT closes the spool file and sends the file results to the printer.
EXIT	Quits SQL*Plus

Using the SAVE and START Commands

```
LIST
  1  SELECT last_name, manager_id, department_id
  2* FROM employees
```

```
SAVE my_query
  Created file my_query
```

```
START my_query

LAST_NAME                 MANAGER_ID DEPARTMENT_ID
------------------------- ---------- -------------
King                                           90
Kochhar                          100           90
...
107 rows selected.
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Using the SAVE and START Commands

SAVE

Use the SAVE command to store the current contents of the buffer in a file. Thus, you can store frequently used scripts for use in the future.

START

Use the START command to run a script in SQL*Plus. You can also, alternatively, use the symbol @ to run a script.

```
@my_query
```

SERVEROUTPUT Command

- Use the SET SERVEROUT[PUT] command to control whether to display the output of stored procedures or PL/SQL blocks in SQL*Plus.
- The DBMS_OUTPUT line length limit is increased from 255 bytes to 32,767 bytes.
- The default size is now unlimited.
- Resources are not preallocated when SERVEROUTPUT is set.
- Because there is no performance penalty, use UNLIMITED unless you want to conserve physical memory.

```
SET SERVEROUT[PUT] {ON | OFF} [SIZE {n | UNL[IMITED]}]
    [FOR[MAT] {WRA[PPED] | WOR[D_WRAPPED] | TRU[NCATED]}]
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur.s@gmail.com) has a non-transferable license to use this Student Guide.

SERVEROUTPUT Command

Most of the PL/SQL programs perform input and output through SQL statements, to store data in database tables or query those tables. All other PL/SQL input/output is performed through APIs that interact with other programs. For example, the DBMS_OUTPUT package has procedures such as PUT_LINE. To see the result outside of PL/SQL requires another program, such as SQL*Plus, to read and display the data passed to DBMS_OUTPUT.

SQL*Plus does not display DBMS_OUTPUT data unless you first issue the SQL*Plus command SET SERVEROUTPUT ON as follows:

```
SET SERVEROUTPUT ON
```

Note

- SIZE sets the number of bytes of the output that can be buffered within the Oracle Database server. The default is UNLIMITED. *n* cannot be less than 2,000 or greater than 1,000,000.
- For additional information about SERVEROUTPUT, see the *Oracle Database PL/SQL User's Guide and Reference 11g*.

Using the SQL*Plus SPOOL Command

```
SPO[OL] [file_name[.ext] [CRE[ATE] | REP[LACE] |
    APP[END]] | OFF | OUT]
```

Option	Description
file_name[.ext]	Spools output to the specified file name
CRE[ATE]	Creates a new file with the name specified
REP[LACE]	Replaces the contents of an existing file. If the file does not exist, REPLACE creates the file.
APP[END]	Adds the contents of the buffer to the end of the file that you specify
OFF	Stops spooling
OUT	Stops spooling and sends the file to your computer's standard (default) printer

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur.34@gmail.com) has a non-transferable license to use this Student Guide.

Using the SQL*Plus SPOOL Command

The SPOOL command stores the query results in a file or optionally sends the file to a printer. The SPOOL command has been enhanced. You can now append to, or replace an existing file, where previously you could only use SPOOL to create (and replace) a file. REPLACE is the default.

To spool the output generated by commands in a script without displaying the output on the screen, use SET TERMOUT OFF. SET TERMOUT OFF does not affect the output from commands that run interactively.

You must use single quotation marks around file names containing white space. To create a valid HTML file using the SPOOL APPEND commands, you must use PROMPT or a similar command to create the HTML page header and footer. The SPOOL APPEND command does not parse HTML tags. SET SQLPLUSCOMPAT[IBILITY] to 9.2 or earlier to disable the CREATE, APPEND, and SAVE parameters.

Using the AUTOTRACE Command

- It displays a report after the successful execution of SQL DML statements such as SELECT, INSERT, UPDATE, or DELETE.
- The report can now include execution statistics and the query execution path.

```
SET AUTOT[RACE] {ON | OFF | TRACE[ONLY]} [EXP[LAIN]]
    [STAT[ISTICS]]
```

```
SET AUTOTRACE ON
-- The AUTOTRACE report includes both the optimizer
-- execution path and the SQL statement execution
-- statistics
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Using the AUTOTRACE Command

EXPLAIN shows the query execution path by performing an EXPLAIN PLAN. STATISTICS displays SQL statement statistics. The formatting of your AUTOTRACE report may vary depending on the version of the server to which you are connected and the configuration of the server. The DBMS_XPLAN package provides an easy way to display the output of the EXPLAIN PLAN command in several predefined formats.

Note

- For additional information about packages and subprograms, see the *Oracle Database PL/SQL Packages and Types Reference 11g* guide.
- For additional information about EXPLAIN PLAN, see the *Oracle Database SQL Reference 11g*.
- For additional information about Execution Plans and statistics, see the *Oracle Database Performance Tuning Guide 11g*.

Summary

In this appendix, you should have learned how to use SQL*Plus as an environment to do the following:

- Execute SQL statements
- Edit SQL statements
- Format the output
- Interact with script files

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Summary

SQL*Plus is an execution environment that you can use to send SQL commands to the database server, and to edit and save SQL commands. You can execute commands from the SQL prompt or from a script file.

Humber College

Using JDeveloper

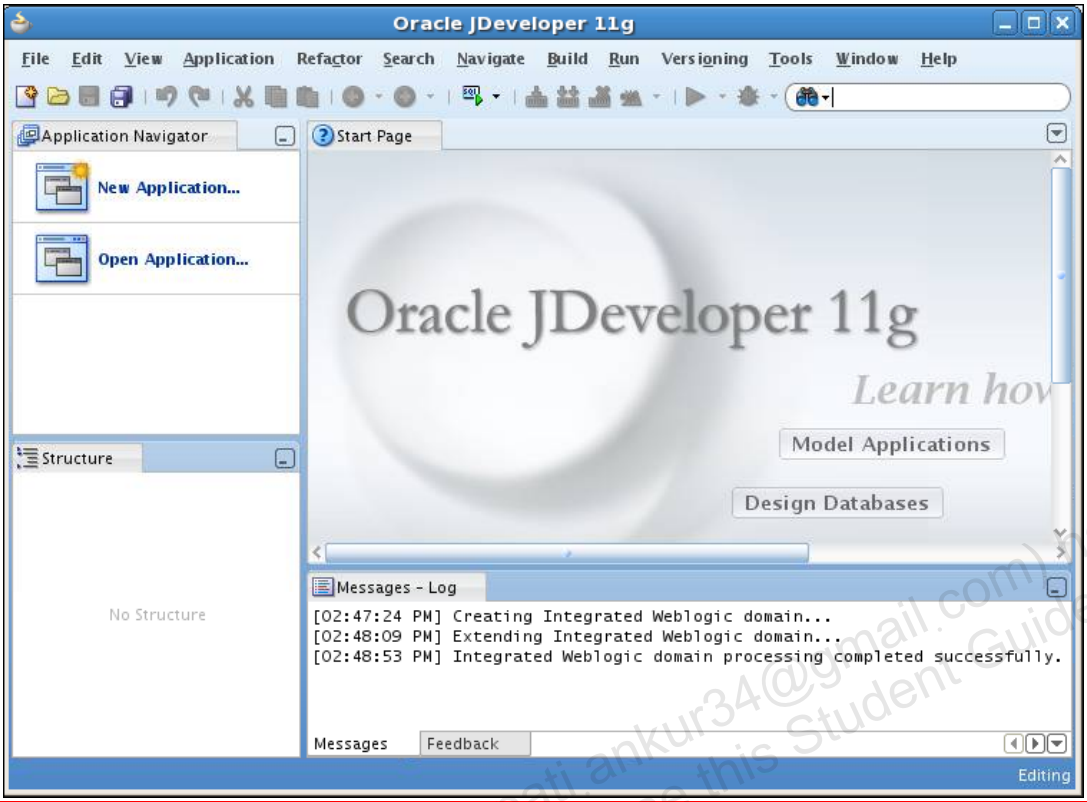
E

ORACLE
Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Oracle JDeveloper



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

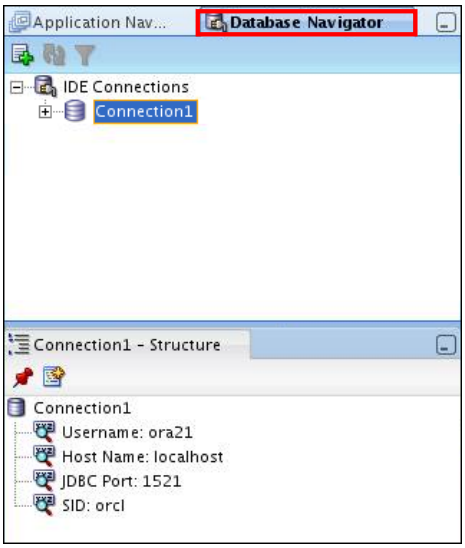
Ankur M. Prajapati (ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Oracle JDeveloper

Oracle JDeveloper is an integrated development environment (IDE) for developing and deploying Java applications and Web services. It supports every stage of the software development life cycle (SDLC) from modeling through deploying. It has the features to use the latest industry standards for Java, XML, and SQL while developing an application.

Oracle JDeveloper 11*g* initiates a new approach to J2EE development with features that enable visual and declarative development. This innovative approach makes J2EE development simple and efficient.

Database Navigator



ORACLE

Copyright © 2009, Oracle. All rights reserved.

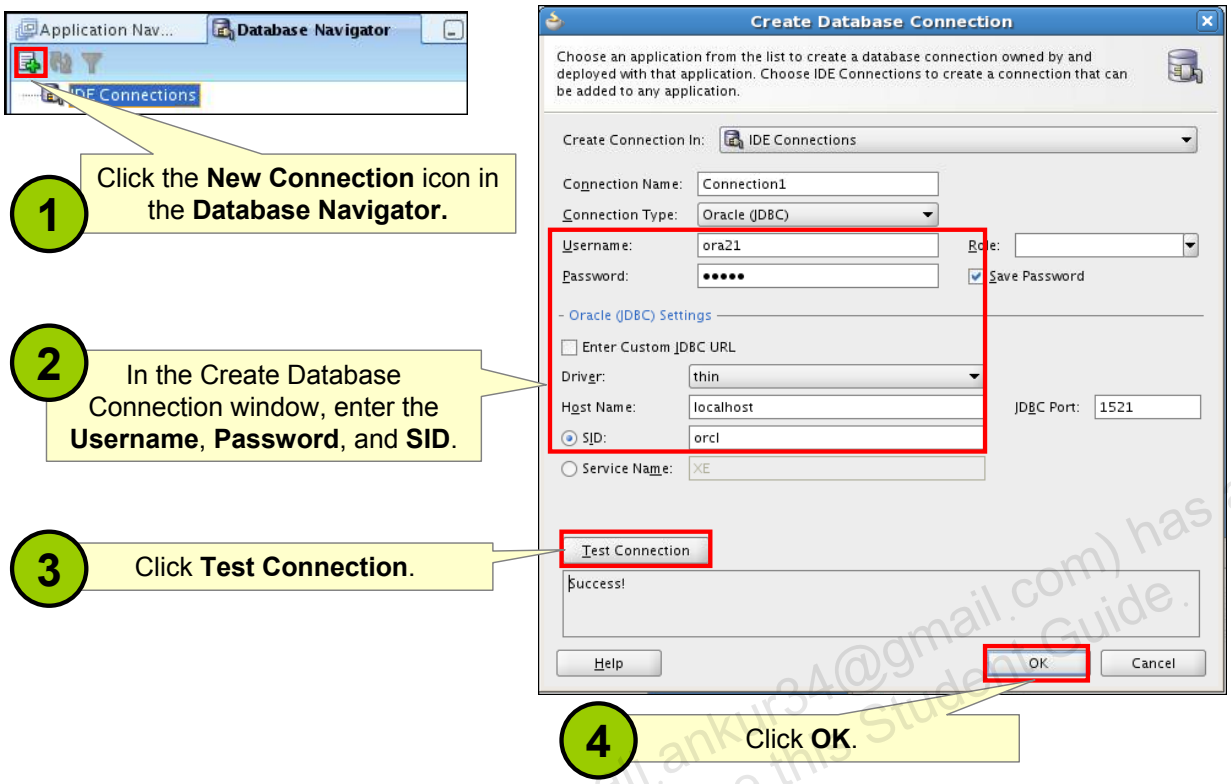
Database Navigator

Using Oracle JDeveloper, you can store information that is necessary to connect to a database in an object called "connection." A connection is stored as part of the IDE settings, and can be exported and imported for easy sharing among groups of users. A connection serves several purposes from browsing the database and building applications, all the way through to deployment.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Creating a Connection



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Creating a Connection

A connection is an object that specifies the necessary information for connecting to a specific database as a specific user of that database. You can create and test connections for multiple databases and for multiple schemas.

To create a database connection, perform the following steps:

1. Click the New Connection icon in the Database Navigator.
2. In the Create Database Connection window, enter the connection name. Enter the username and password of the schema that you want to connect to. Enter the SID of the database that you want to connect.
3. Click Test to ensure that the connection has been set correctly.
4. Click OK.

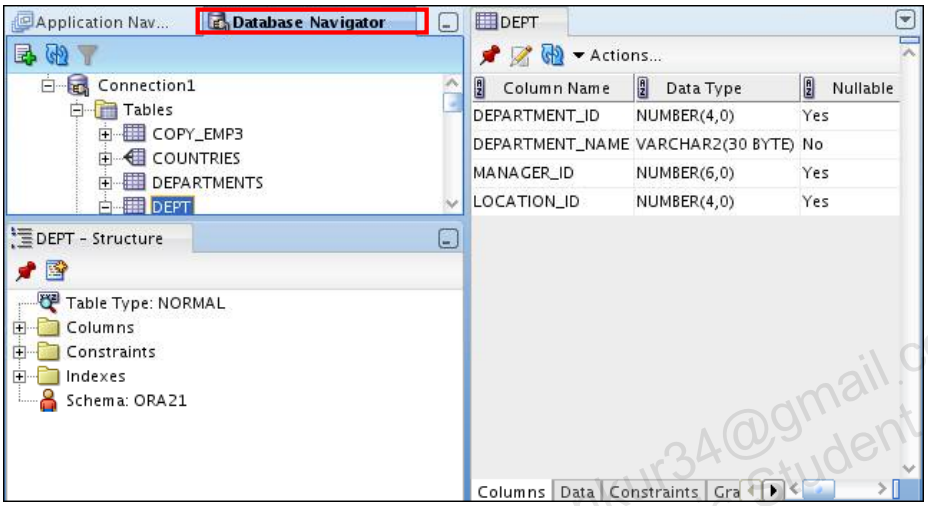
Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur84@gmail.com) has a non-transferable license to use this Student Guide.

Browsing Database Objects

Use the Database Navigator to:

- Browse through many objects in a database schema
- Review the definitions of objects at a glance



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

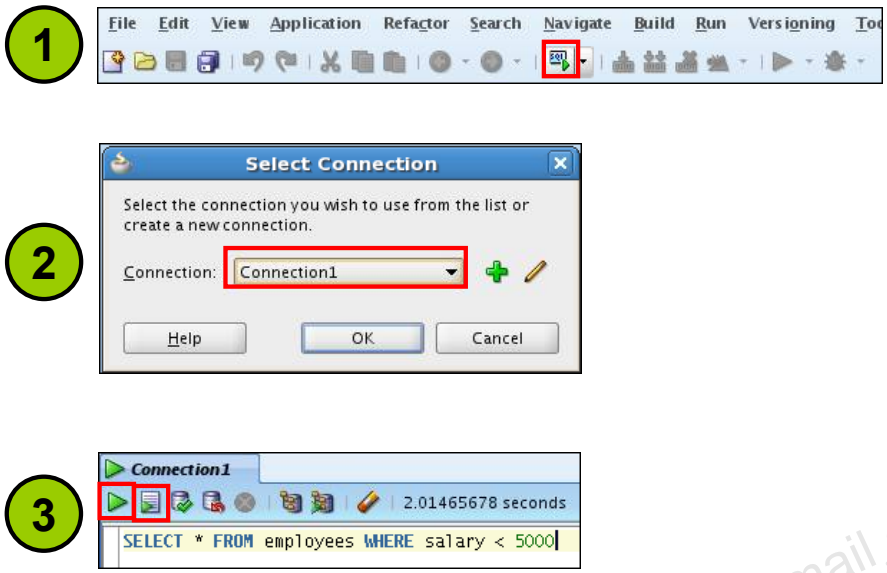
Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Browsing Database Objects

After you create a database connection, you can use the Database Navigator to browse through many objects in a database schema, including tables, views, indexes, packages, procedures, triggers, and types.

You can see the definition of objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

Executing SQL Statements



ORACLE

Copyright © 2009, Oracle. All rights reserved.

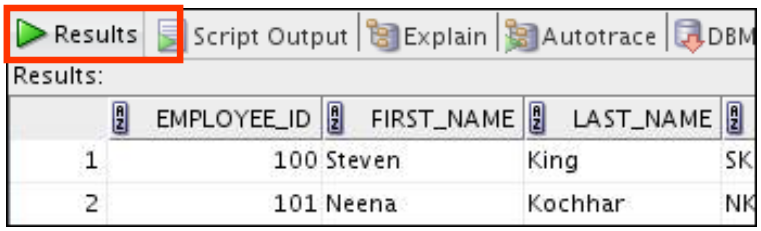
Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

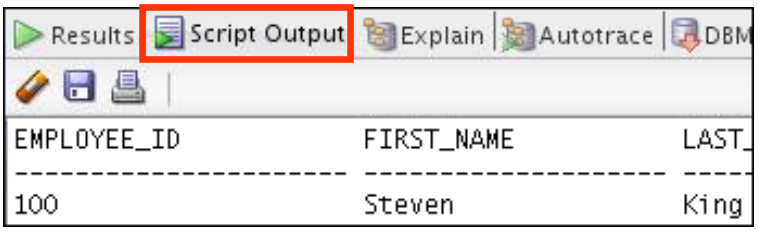
Executing SQL Statements

To execute a SQL statement, perform the following steps:

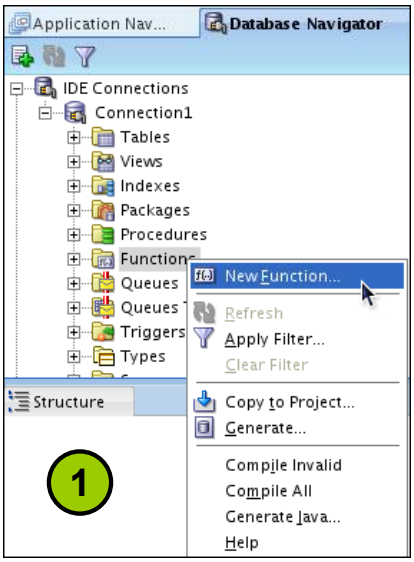
1. Click the Open SQL Worksheet icon.
2. Select the connection.
3. Execute the SQL command by clicking either of the following:
   - The **Execute statement** button or by pressing F9. The output is as follows:

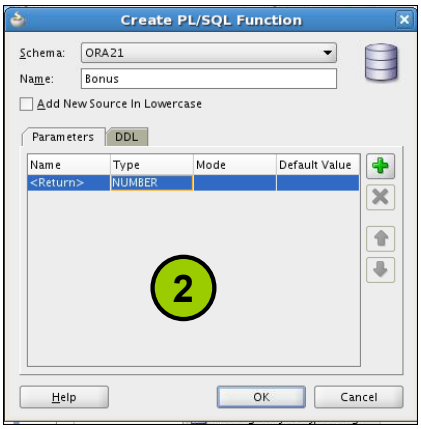


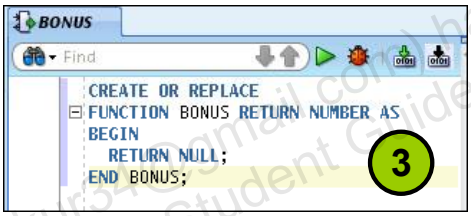
   - The **Run Script** button or by pressing F5. The output is as follows:



Creating Program Units







Skeleton of the function

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Creating Program Units

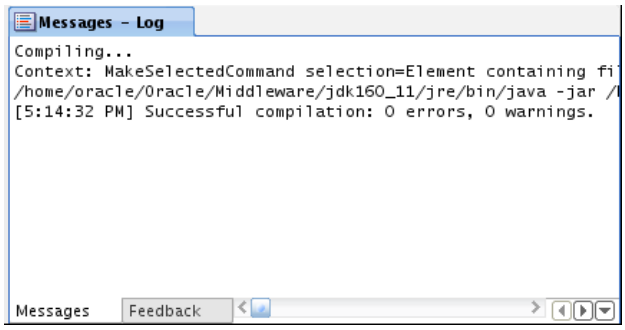
To create a PL/SQL program unit, perform the following steps:

1. Select View > Database Navigator. Select and expand a database connection. Right-click a folder corresponding to the object type (procedures, packages, or functions). Select "New [Procedures|Packages|Functions]".
2. Enter a valid name for the function, package, or procedure, and click OK.
3. A skeleton definition is created and opened in the Code Editor. You can then edit the subprogram to suit your need.

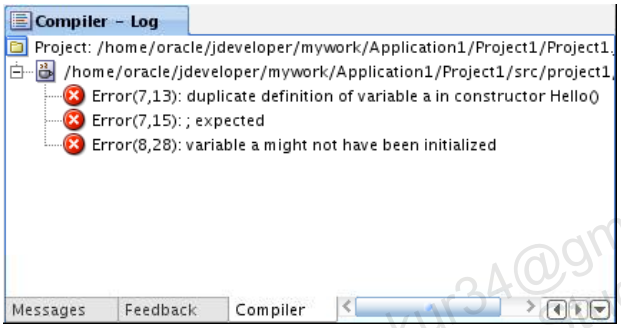
Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Compiling



Compilation with errors



Compilation without errors

ORACLE

Copyright © 2009, Oracle. All rights reserved.

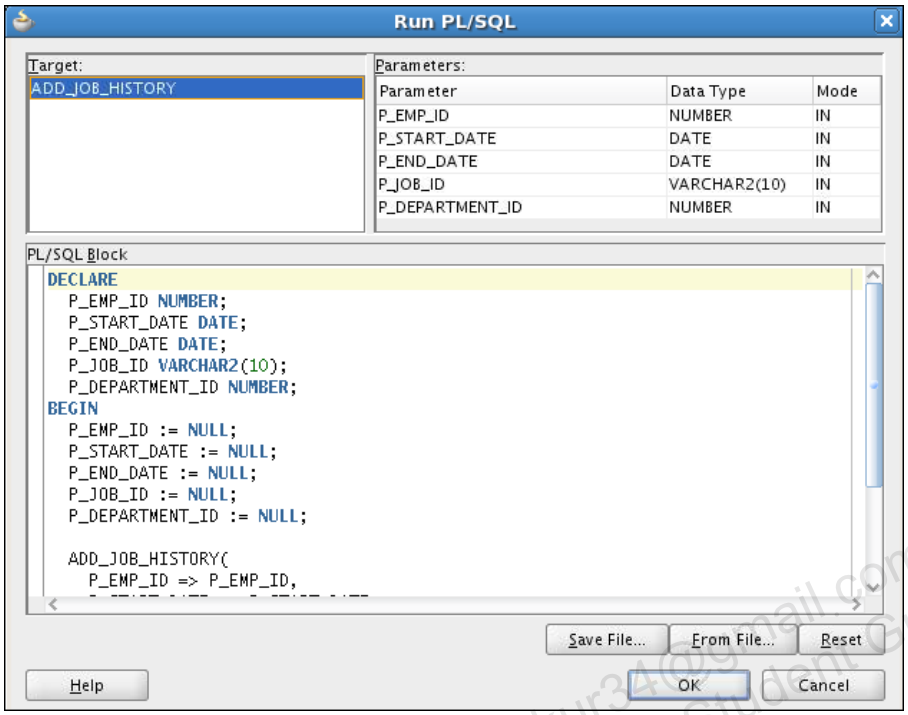
Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Compiling

After editing the skeleton definition, you need to compile the program unit. Right-click the PL/SQL object that you need to compile in the Connection Navigator, and then select Compile. Alternatively, you can press CTRL + SHIFT + F9 to compile.

Running a Program Unit



ORACLE

Copyright © 2009, Oracle. All rights reserved.

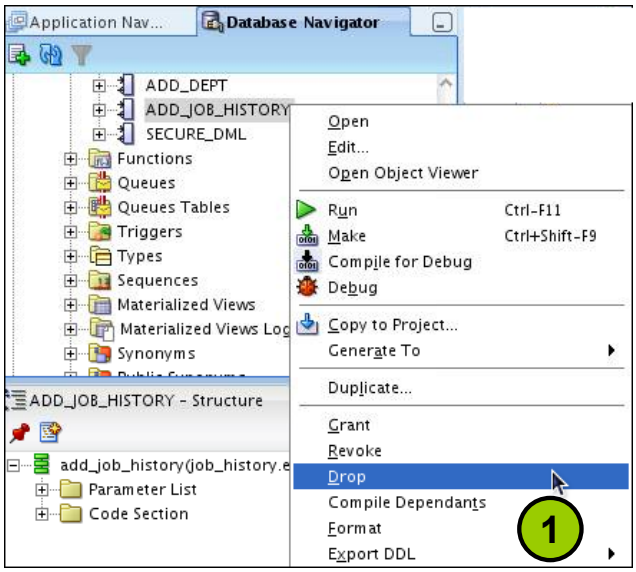
Running a Program Unit

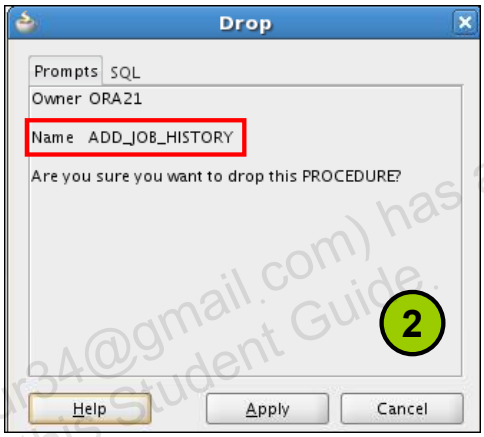
To execute the program unit, right-click the object and select Run. The Run PL/SQL dialog box appears. You may need to change the `NULL` values with reasonable values that are passed into the program unit. After you change the values, click OK. The output is displayed in the Message-Log window.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Dropping a Program Unit





ORACLE

Copyright © 2009, Oracle. All rights reserved.

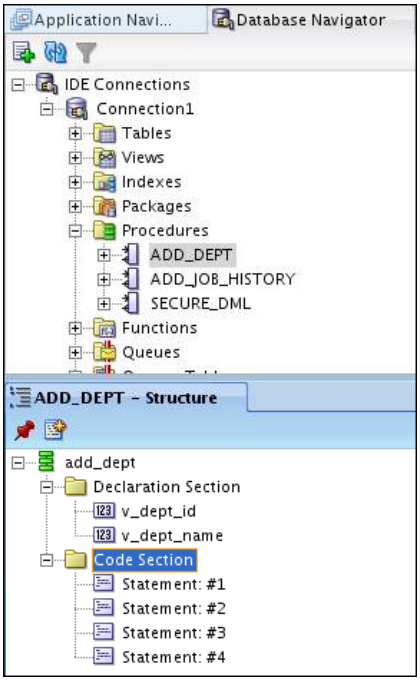
Dropping a Program Unit

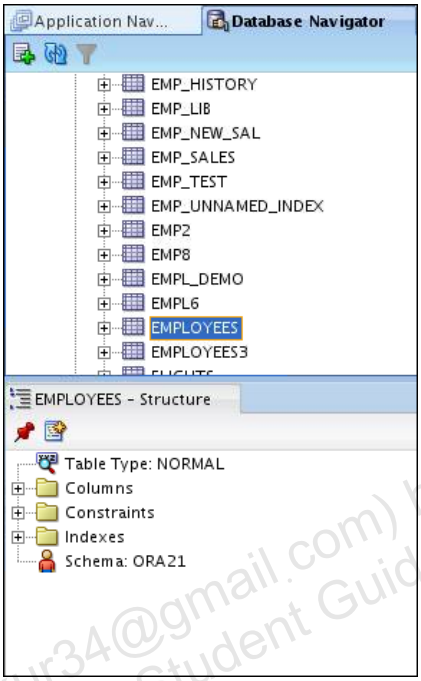
To drop a program unit, right-click the object and select Drop. The Drop Confirmation dialog box appears. Click **Apply**. The object is dropped from the database.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Structure Window





ORACLE

Copyright © 2009, Oracle. All rights reserved.

Structure Window

The Structure window offers a structural view of the data in the document that is currently selected in the active window of those windows that participate in providing structure: the navigators, the editors and viewers, and the Property Inspector.

Select View > Structure window to view the Structure window.

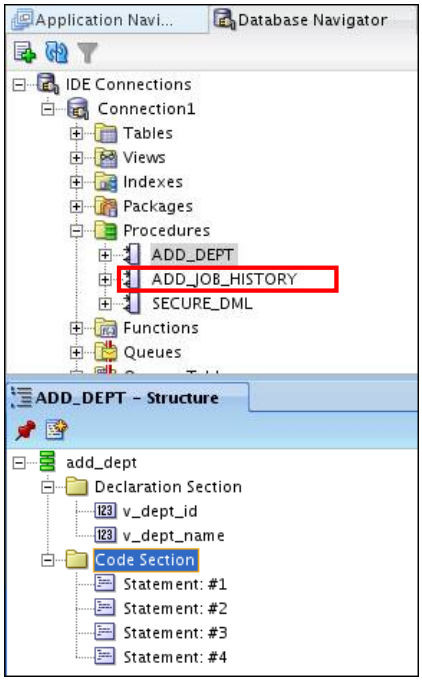
In the Structure window, you can view the document data in a variety of ways. The structures that are available for display are based on the document type. For a Java file, you can view code structure, UI structure, or UI model data. For an XML file, you can view XML structure, design structure, or UI model data.

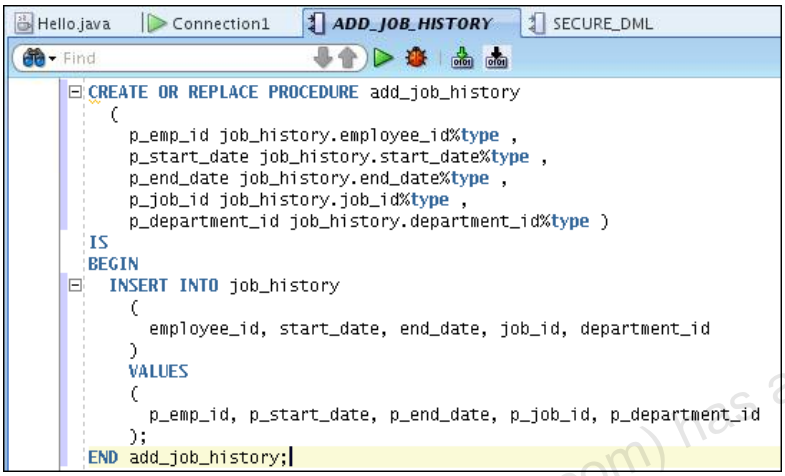
The Structure window is dynamic, tracking always the current selection of the active window (unless you freeze the window's contents on a particular view), as is pertinent to the currently active editor. When the current selection is a node in the navigator, the default editor is assumed. To change the view on the structure for the current selection, click a different structure tab.

Ankur M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Editor Window





ORACLE

Copyright © 2009, Oracle. All rights reserved.

Editor Window

When you double-click the name of a program unit, it opens in the Editor window. You can view all your project files in one single editor window, you can open multiple views of the same file, or you can open multiple views of different files.

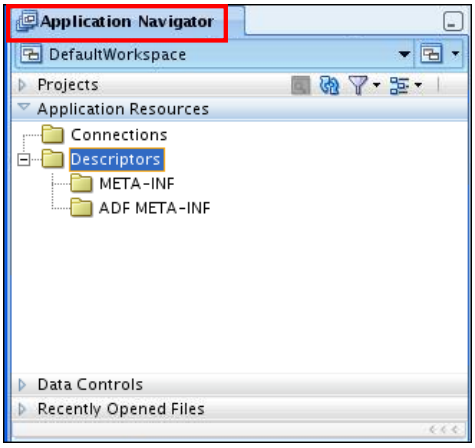
The tabs at the top of the editor window are the document tabs. Clicking a document tab gives that file focus, bringing it to the foreground of the window in the current editor.

The tabs at the bottom of the editor window for a given file are the editor tabs. Clicking an editor tab opens the file in that editor.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Application Navigator



ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Anup M Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

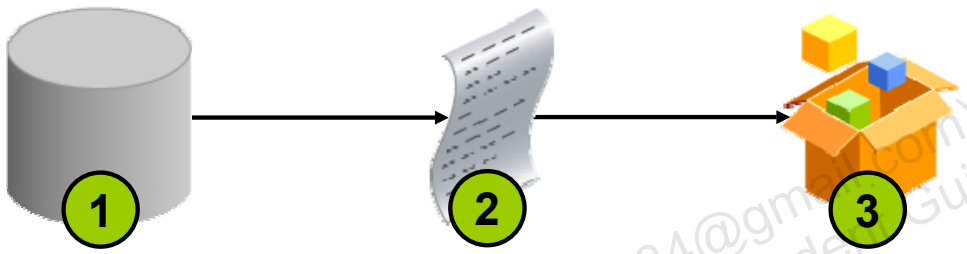
Application Navigator

The Application Navigator gives you a logical view of your application and the data that it contains. The Application Navigator provides an infrastructure that the different extensions can plug in to and use to organize their data and menus in a consistent, abstract manner. Although the Application Navigator can contain individual files (such as Java source files), it is designed to consolidate complex data. Complex data types such as entity objects, Unified Modeling Language (UML) diagrams, Enterprise JavaBeans (EJBs), or Web services appear in this navigator as single nodes. The raw files that make up these abstract nodes appear in the Structure window.

Deploying Java Stored Procedures

Before deploying Java stored procedures, perform the following steps:

1. Create a database connection.
2. Create a deployment profile.
3. Deploy the objects.



Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Deploying Java Stored Procedures

Create a deployment profile for Java stored procedures, and then deploy the classes and, optionally, any public static methods in JDeveloper using the settings in the profile.

When the database is deployed, the information provided in the Deployment Profile Wizard and two Oracle Database utilities are used:

- `loadjava` loads the Java class containing the stored procedures to an Oracle database.
- `publish` generates the PL/SQL call–specific wrappers for the loaded public static methods. Publishing enables the Java methods to be called as PL/SQL functions or procedures.

Publishing Java to PL/SQL

TrimLob.java | TRIMLOBPROC

```
public class TrimLob
{
  public static void main (String args []) throws SQLException {
  Connection conn=null;
   if (System.getProperty("oracle.jserver.version") != null)
  {
   conn = DriverManager.getConnection("jdbc:default:connection:");
  }
  else
  {
  DriverManager.registerDriver(new oracle.jdbc.OracleDriver());
   conn = DriverManager.getConnection("jdbc:oracle:thin:scott/tiger");
  }
```

1

```
[oracle@edrsr5p1-orcl ~]$ cd wkdir
[oracle@edrsr5p1-orcl wkdir]$ loadjava -u hr/hr TrimLob.java
[oracle@edrsr5p1-orcl wkdir]$
```

2

TrimLob.java | TRIMLOBPROC

```
CREATE OR REPLACE PROCEDURE TRIMLOBPROC
as language java
  name 'TrimLob.main(java.lang.String[])';
/
```

3

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Publishing Java to PL/SQL

The slide shows the Java code and illustrates how to publish the Java code in a PL/SQL procedure.

1. Create and compile a Java routine.
2. Execute the command shown in the slide to load the TrimLob Java routine into the database.
3. Publish the Java class method by creating the TrimLobProc PL/SQL call specification that references the Java class methods. Within the TrimLobProc PL/SQL specification, you identify the name of the Java class method and its parameters. This is called “Publishing” the Java class method.

For more details on how to run Java routines in the database, refer to the following OBE: http://www.oracle.com/technology/obe/hol08/11gR1_JDBC_Java/java_otn.htm

How Can I Learn More About JDeveloper 11*g*?

Topic	Website
Oracle JDeveloper Product Page	http://www.oracle.com/technology/products/jdev/index.html
Oracle JDeveloper 11*g* Tutorials	http://www.oracle.com/technology/obe/obe11jdev/11/index.html
Oracle JDeveloper 11*g* Product Documentation	http://www.oracle.com/technology/documentation/jdev.html
Oracle JDeveloper 11*g* Discussion Forum	http://forums.oracle.com/forums/forum.jspa?forumID=83

Ankur M. Prajapati (rajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

F

REF Cursors

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Cursor Variables

- Cursor variables are like C or Pascal pointers, which hold the memory location (address) of an item instead of the item itself.
- In PL/SQL, a pointer is declared as `REF X`, where `REF` is short for `REFERENCE` and `X` stands for a class of objects.
- A cursor variable has the data type `REF CURSOR`.
- A cursor is static, but a cursor variable is dynamic.
- Cursor variables give you more flexibility.

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Cursor Variables

Cursor variables are like C or Pascal pointers, which hold the memory location (address) of an item instead of the item itself. Thus, declaring a cursor variable creates a pointer, not an item. In PL/SQL, a pointer has the data type `REF X`, where `REF` is short for `REFERENCE` and `X` stands for a class of objects. A cursor variable has the `REF CURSOR` data type.

Like a cursor, a cursor variable points to the current row in the result set of a multirow query. However, cursors differ from cursor variables the way constants differ from variables. A cursor is static, but a cursor variable is dynamic because it is not tied to a specific query. You can open a cursor variable for any type-compatible query. This gives you more flexibility.

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, and then pass it as an input host variable (bind variable) to PL/SQL. Moreover, application development tools such as Oracle Forms and Oracle Reports, which have a PL/SQL engine, can use cursor variables entirely on the client side. The Oracle Server also has a PL/SQL engine. You can pass cursor variables back and forth between an application and server through remote procedure calls (RPCs).

Using Cursor Variables

- You can use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients.
- PL/SQL can share a pointer to the query work area in which the result set is stored.
- You can pass the value of a cursor variable freely from one scope to another.
- You can reduce network traffic by having a PL/SQL block open (or close) several host cursor variables in a single roundtrip.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Anjur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Using Cursor Variables

You use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area in which the result set is stored. For example, an OCI client, an Oracle Forms application, and the Oracle Server can all refer to the same work area.

A query work area remains accessible as long as any cursor variable points to it. Therefore, you can pass the value of a cursor variable freely from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block that is embedded in a Pro*C program, the work area to which the cursor variable points remains accessible after the block completes.

If you have a PL/SQL engine on the client side, calls from the client to the server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, and then continue to fetch from it back on the client side. Also, you can reduce network traffic by having a PL/SQL block open (or close) several host cursor variables in a single roundtrip.

A cursor variable holds a reference to the cursor work area in the Program Global Area (PGA) instead of addressing it with a static name. Because you address this area by a reference, you gain the flexibility of a variable.

Defining REF CURSOR Types

Define a REF CURSOR type:

```
Define a REF CURSOR type
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

Declare a cursor variable of that type:

```
ref_cv ref_type_name;
```

Example:

```
DECLARE
TYPE DeptCurTyp IS REF CURSOR RETURN
departments%ROWTYPE;
dept_cv DeptCurTyp;
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Defining REF CURSOR Types

To define a REF CURSOR, you perform two steps. First, you define a REF CURSOR type, and then you declare cursor variables of that type. You can define REF CURSOR types in any PL/SQL block, subprogram, or package using the following syntax:

```
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

where:

ref_type_name	Is a type specifier used in subsequent declarations of cursor variables
return_type	Represents a record or a row in a database table

In this example, you specify a return type that represents a row in the database table DEPARTMENT.

REF CURSOR types can be strong (restrictive) or weak (nonrestrictive). As the next example shows, a strong REF CURSOR type definition specifies a return type, but a weak definition does not:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;  --
strong
   TYPE GenericCurTyp IS REF CURSOR;  -- weak
```

Defining REF CURSOR Types (continued)

Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with type-compatible queries. However, weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query.

Declaring Cursor Variables

After you define a REF CURSOR type, you can declare cursor variables of that type in any PL/SQL block or subprogram. In the following example, you declare the cursor variable DEPT_CV:

```
DECLARE
   TYPE DeptCurTyp IS REF CURSOR RETURN departments%ROWTYPE;
   dept_cv DeptCurTyp;  -- declare cursor variable
```

Note: You cannot declare cursor variables in a package. Unlike packaged variables, cursor variables do not have persistent states. Remember, declaring a cursor variable creates a pointer, not an item. Cursor variables cannot be saved in the database; they follow the usual scoping and instantiation rules.

In the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to specify a record type that represents a row returned by a strongly (not weakly) typed cursor variable, as follows:

```
DECLARE
   TYPE TmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
   tmp_cv TmpCurTyp;  -- declare cursor variable
   TYPE EmpCurTyp IS REF CURSOR RETURN tmp_cv%ROWTYPE;
   emp_cv EmpCurTyp;  -- declare cursor variable
```

Similarly, you can use %TYPE to provide the data type of a record variable, as the following example shows:

```
DECLARE
   dept_rec departments%ROWTYPE;  -- declare record variable
   TYPE DeptCurTyp IS REF CURSOR RETURN dept_rec%TYPE;
   dept_cv DeptCurTyp;  -- declare cursor variable
```

In the final example, you specify a user-defined RECORD type in the RETURN clause:

```
DECLARE
   TYPE EmpRecTyp IS RECORD (
      empno NUMBER(4),
      ename VARCHAR2(10),
      sal   NUMBER(7,2));
   TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
   emp_cv EmpCurTyp;  -- declare cursor variable
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide

Cursor Variables as Parameters

You can declare cursor variables as the formal parameters of functions and procedures. In the following example, you define the `REF CURSOR` type `EmpCurTyp`, and then declare a cursor variable of that type as the formal parameter of a procedure:

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR RETURN emp%ROWTYPE;
   PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS ...
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Using the OPEN-FOR, FETCH, and CLOSE Statements

- The OPEN-FOR statement associates a cursor variable with a multirow query, executes the query, identifies the result set, and positions the cursor to point to the first row of the result set.
- The FETCH statement returns a row from the result set of a multirow query, assigns the values of the select-list items to the corresponding variables or fields in the INTO clause, increments the count kept by %ROWCOUNT, and advances the cursor to the next row.
- The CLOSE statement disables a cursor variable.

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Using the OPEN-FOR, FETCH, and CLOSE Statements

You use three statements to process a dynamic multirow query: OPEN-FOR, FETCH, and CLOSE. First, you "open" a cursor variable "for" a multirow query. Then you "fetch" rows from the result set one at a time. When all the rows are processed, you "close" the cursor variable.

Opening the Cursor Variable

The OPEN-FOR statement associates a cursor variable with a multirow query, executes the query, identifies the result set, positions the cursor to point to the first row of the results set, and then sets the rows-processed count kept by %ROWCOUNT to zero. Unlike the static form of OPEN-FOR, the dynamic form has an optional USING clause. At run time, bind arguments in the USING clause replace corresponding placeholders in the dynamic SELECT statement. The syntax is:

```
OPEN {cursor_variable | :host_cursor_variable} FOR
dynamic_string
          [USING bind_argument[, bind_argument]...];
```

where CURSOR_VARIABLE is a weakly typed cursor variable (one without a return type), HOST_CURSOR_VARIABLE is a cursor variable declared in a PL/SQL host environment such as an OCI program, and dynamic_string is a string expression that represents a multirow query.

Using the OPEN-FOR, FETCH, and CLOSE Statements (continued)

In the following example, the syntax declares a cursor variable, and then associates it with a dynamic SELECT statement that returns rows from the employees table:

```
DECLARE
 TYPE EmpCurTyp IS REF CURSOR;  -- define weak REF CURSOR
   type
 emp_cv   EmpCurTyp;  -- declare cursor variable
 my_ename VARCHAR2(15);
 my_sal   NUMBER := 1000;
BEGIN
 OPEN emp_cv FOR  -- open cursor variable
    'SELECT last_name, salary FROM employees WHERE salary >
        :s'
     USING my_sal;
 ...
END;
```

Any bind arguments in the query are evaluated only when the cursor variable is opened. Thus, to fetch rows from the cursor using different bind values, you must reopen the cursor variable with the bind arguments set to their new values each time.

Fetching from the Cursor Variable

The FETCH statement returns a row from the result set of a multirow query, assigns the values of the select-list items to the corresponding variables or fields in the INTO clause, increments the count kept by %ROWCOUNT, and advances the cursor to the next row. Use the following syntax:

```
FETCH {cursor_variable | :host_cursor_variable}
 INTO {define_variable[, define_variable]... | record};
```

Continuing the example, fetch rows from the cursor variable emp_cv into the define variables MY_ENAME and MY_SAL:

```
LOOP
 FETCH emp_cv INTO my_ename, my_sal;  -- fetch next row
 EXIT WHEN emp_cv%NOTFOUND;  -- exit loop when last row is
   fetched
 -- process row
END LOOP;
```

For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-compatible variable or field in the INTO clause. You can use a different INTO clause on separate fetches with the same cursor variable. Each fetch retrieves another row from the same result set. If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Anwar M. Prajapati (prajapatiankur34@gmail.com) has a non-transferable license to use this Student Guide

Using the OPEN-FOR, FETCH, and CLOSE Statements (continued)

Closing the Cursor Variable

The CLOSE statement disables a cursor variable. After that, the associated result set is undefined. Use the following syntax:

```
CLOSE {cursor_variable | :host_cursor_variable};
```

In this example, when the last row is processed, close the emp_cv cursor variable:

```
LOOP
  FETCH emp_cv INTO my_ename, my_sal;
  EXIT WHEN emp_cv%NOTFOUND;
  -- process row
END LOOP;
CLOSE emp_cv;  -- close cursor variable
```

If you try to close an already-closed or never-opened cursor variable, PL/SQL raises INVALID_CURSOR.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Example of Fetching

```
DECLARE
   TYPE EmpCurTyp IS REF CURSOR;
   emp_cv   EmpCurTyp;
   emp_rec  employees%ROWTYPE;
   sql_stmt VARCHAR2(200);
   my_job   VARCHAR2(10) := 'ST_CLERK';
BEGIN
   sql_stmt := 'SELECT * FROM employees
                WHERE job_id = :j';
   OPEN emp_cv FOR sql_stmt USING my_job;
   LOOP
      FETCH emp_cv INTO emp_rec;
      EXIT WHEN emp_cv%NOTFOUND;
      -- process record
   END LOOP;
   CLOSE emp_cv;
END;
/
```

ORACLE

Copyright © 2009, Oracle. All rights reserved.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Example of Fetching

The example in the slide shows that you can fetch rows from the result set of a dynamic multirow query into a record. You must first define a `REF CURSOR` type, `EmpCurTyp`. You then define a cursor variable `emp_cv`, of the type `EmpcurTyp`. In the executable section of the PL/SQL block, the `OPEN-FOR` statement associates the cursor variable `emp_cv` with the multirow query, `sql_stmt`. The `FETCH` statement returns a row from the result set of a multirow query and assigns the values of the select-list items to `EMP_REC` in the `INTO` clause. When the last row is processed, close the `emp_cv` cursor variable.

Appendix AP
Practices and Solutions

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Table of Contents

Overview ... 3
Additional Practices and Solutions: Lessons 1 and 2 ... 4
Practice 1: Evaluating Declarations ... 4
Practice 2: Evaluating Expressions ... 4
Solution 1: Evaluating Declarations ... 5
Solution 2: Evaluating Expressions ... 5
Additional Practice and Solution: Lesson 3 ... 6
Practice 3: Evaluating Executable Statements ... 6
Solution 3: Evaluating Executable Statements ... 7
Additional Practices and Solutions for Lesson 4 ... 8
Practice 4-1: Interacting with the Oracle Server ... 8
Practice 4-2: Interacting with the Oracle Server ... 8
Solution 4-1: Interacting with the Oracle Server ... 9
Solution 4-2: Interacting with the Oracle Server ... 10
Additional Practices and Solutions for Lesson 5 ... 12
Practice 5-1: Writing Control Structures ... 12
Practice 5-2: Writing Control Structures ... 12
Solution 5-1: Writing Control Structures ... 13
Solution 5-2: Writing Control Structures ... 14
Additional Practices and Solutions for Lessons 6 and 7 ... 15
Practice 6/7-1: Fetching Data with an Explicit Cursor ... 15
Practice 6/7-2: Using Associative Arrays and Explicit Cursors ... 15
Solution 6/7-1: Fetching Data with an Explicit Cursor ... 16
Solution 6/7-2: Using Associative Arrays and Explicit Cursors ... 17
Additional Practice and Solution for Lesson 8 ... 19
Practice 8-1: Handling Exceptions ... 19
Solution 8-1: Handling Exceptions ... 20

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur Nirmal Pan (ankurnirmalpan1994@gmail.com) has a non-transferable license to use this Student Guide.

Overview

These additional practices are provided as a supplement to the *Oracle Database 11g: PL/SQL Fundamentals* course. In these practices, you apply the concepts that you learned in the course.

These additional practices provide supplemental practice in declaring variables, writing executable statements, interacting with the Oracle Server, writing control structures, and working with composite data types, cursors, and handle exceptions. The tables used in this portion of the additional practices include `employees`, `jobs`, `job_history`, and `departments`.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Additional Practices and Solutions: Lessons 1 and 2

These paper-based exercises are used for extra practice in declaring variables and writing executable statements.

Practice 1: Evaluating Declarations

Evaluate each of the following declarations. Determine which of them are not legal and explain why.

1.
```
DECLARE
name,dept     VARCHAR2(14);
```

2.
```
DECLARE
test          NUMBER(5);
```

3.
```
DECLARE
MAXSALARY     NUMBER(7,2) = 5000;
```

4.
```
DECLARE
JOINDATE      BOOLEAN := SYSDATE;
```

Practice 2: Evaluating Expressions

In each of the following assignments, determine the data type of the resulting expression.

1. `email := firstname || to_char(empno);`

2. `confirm := to_date('20-JAN-1999', 'DD-MON-YYYY');`

3. `sal := (1000*12) +  500`

4. `test := FALSE;`

5. `temp := temp1 < (temp2/ 3);`

6. `var := sysdate;`

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution 1: Evaluating Declarations

Evaluate each of the following declarations. Determine which of them are not legal and explain why.

1. ```
   DECLARE
   name,dept     VARCHAR2(14);
   ```

   This is illegal because only one identifier per declaration is allowed.

2. ```
   DECLARE
   test          NUMBER(5);
   ```

   This is legal.

3. ```
   DECLARE
   MAXSALARY     NUMBER(7,2) = 5000;
   ```

   This is illegal because the assignment operator is wrong. It should be `:=`.

4. ```
   DECLARE
   JOINDATE      BOOLEAN := SYSDATE;
   ```

   This is illegal because there is a mismatch in the data types. A Boolean data type cannot be assigned a date value. The data type should be date.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 2: Evaluating Expressions

In each of the following assignments, determine the data type of the resulting expression.

1. `email := firstname || to_char(empno);`

   Character string

2. `confirm := to_date('20-JAN-1999', 'DD-MON-YYYY');`

   Date

3. `sal := (1000*12) +  500`

   Number

4. `test := FALSE;`

   Boolean

5. temp := temp1 < (temp2/ 3);

   Boolean

6. `var := sysdate;`

   Date

Additional Practice and Solution: Lesson 3

Practice 3: Evaluating Executable Statements

In this paper-based exercise, you evaluate the PL/SQL block, and then answer the questions that follow by determining the data type and value of each variable, according to the rules of scoping.

```
DECLARE
      v_custid    NUMBER(4) := 1600;
      v_custname VARCHAR2(300) := 'Women Sports Club';
      v_ new_custid   NUMBER(3) := 500;
BEGIN
DECLARE
      v_custid     NUMBER(4) := 0;
      v_custname  VARCHAR2(300) := 'Shape up Sports Club';
      v_new_custid  NUMBER(3) := 300;
      v_new_custname  VARCHAR2(300) := 'Jansports Club';
BEGIN
      v_custid := v_new_custid;
      v_custname := v_custname || ' ' ||  v_new_custname;
1 ---------->
END;
      v_custid := (v_custid *12) / 10;
2 ---------->
END;
```

Evaluate the preceding PL/SQL block and determine the *value* and *data type* of each of the following variables, according to the rules of scoping:

1. `v_custid` at position 1:

2. `v_custname` at position 1:

3. `v_new_custid` at position 1:

4. `v_new_custname` at position 1:

5. `v_custid` at position 2:

6. `v_custname` at position 2:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 3: Evaluating Executable Statements

Evaluate the following PL/SQL block. Then, answer the questions that follow by determining the data type and value of each of the following variables, according to the rules of scoping.

```
DECLARE
      v_custid    NUMBER(4) := 1600;
      v_custname VARCHAR2(300) := 'Women Sports Club';
      v_ new_custid   NUMBER(3) := 500;
BEGIN
DECLARE
      v_custid     NUMBER(4) := 0;
      v_custname  VARCHAR2(300) := 'Shape up Sports Club';
      v_new_custid  NUMBER(3) := 300;
      v_new_custname  VARCHAR2(300) := 'Jansports Club';
BEGIN
      v_custid := v_new_custid;
      v_custname := v_custname || ' ' ||  v_new_custname;
1 ---------->
END;
      v_custid := (v_custid *12) / 10;
2 ---------->
END;
```

Evaluate the preceding PL/SQL block and determine the *value* and *data type* of each of the following variables, according to the rules of scoping:

1. `v_custid` at position 1:

   300, and the data type is `NUMBER`.

2. `v_custname` at position 1:

   Shape up Sports Club Jansports Club, and the data type is `VARCHAR2`.

3. `v_new_custid` at position 1:

   500, and the data type is `NUMBER` (or `INTEGER`).

4. `v_new_custname` at position 1:

   Jansports Club, and the data type is `VARCHAR2`.

5. `v_custid` at position 2:

   1920, and the data type is `NUMBER`.

6. `v_custname` at position 2:

   Women Sports Club, and the data type is `VARCHAR2`.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Additional Practices and Solutions for Lesson 4

Practice 4-1: Interacting with the Oracle Server

For this exercise, a temporary table is required to store the results.

1. Run the `lab_ap_04.sql` script that creates the table described here:

Column Name	NUM_STORE	CHAR_STORE	DATE_STORE
Key Type			
Nulls/Unique			
FK Table			
FK Column			
Data Type	Number	VARCHAR2	Date
Length	7,2	35	

2. Write a PL/SQL block that performs the following:

   a) Declares two variables and assigns the following values to these variables:

Variable	Data type	Contents
V_MESSAGE	VARCHAR2(35)	This is my first PL/SQL program
V_ DATE_WRITTEN	DATE	Current date

   b) Stores the values from these variables in the appropriate `TEMP` table columns

3. Verify your results by querying the `TEMP` table. The output results should appear as follows:

```
anonymous block completed
NUM_STORE            CHAR_STORE                          DATE_STORE
-------------------- ----------------------------------- --------------
                     This is my first PLSQL Program      15-JUL-09

1 rows selected
```

Practice 4-2: Interacting with the Oracle Server

In this exercise, you use data from the `employees` table.

1. Write a PL/SQL block to determine how many employees work for a specified department. The PL/SQL block should:
   - Use a substitution variable to store a department number
   - Print the number of people working in the specified department
2. When the block is run, a substitution variable window appears. Enter a valid department number and click OK. The output results should look similar to the following:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur K. Prajapati (prajapati.ankur@gmail.com) has a non-transferable license to use this Student Guide

```
anonymous block completed
6 employee(s) work for department number 30
```

Solution 4-1: Interacting with the Oracle Server

For this exercise, a temporary table is required to store the results.

1. Run the `lab_ap_04.sql` script that creates the table described here:

Column Name	NUM_STORE	CHAR_STORE	DATE_STORE
Key Type			
Nulls/Unique			
FK Table			
FK Column			
Data Type	Number	VARCHAR2	Date
Length	7,2	35	

2. Write a PL/SQL block that performs the following:

   a) Declares two variables and assigns the following values to these variables:

Variable	Data type	Contents
V_MESSAGE	VARCHAR2(35)	This is my first PL/SQL program
V_ DATE_WRITTEN	DATE	Current date

   b) Stores the values from these variables in the appropriate `TEMP` table columns

```
DECLARE
  V_MESSAGE VARCHAR2(35);
  V_DATE_WRITTEN DATE;
BEGIN
  V_MESSAGE := 'This is my first PLSQL Program';
  V_DATE_WRITTEN := SYSDATE;
  INSERT INTO temp(CHAR_STORE,DATE_STORE)
    VALUES (V_MESSAGE,V_DATE_WRITTEN);
END;
/
```

3. Verify your results by querying the `TEMP` table. The output results should look similar to the following:

```
SELECT * FROM TEMP;
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur.24@gmail.com) has a non-transferable license to use this Student Guide.

```
Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

anonymous block completed
NUM_STORE            CHAR_STORE                          DATE_STORE
--------------------- ----------------------------------- --------------
                      This is my first PLSQL Program      15-JUL-09

1 rows selected
```

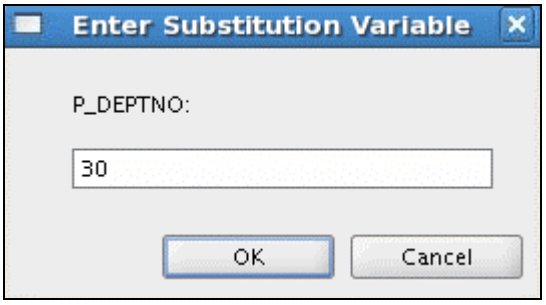
Solution 4-2: Interacting with the Oracle Server

In this exercise, you use data from the `employees` table.

1. Write a PL/SQL block to determine how many employees work for a specified department. The PL/SQL block should:
   - Use a substitution variable to store a department number
   - Print the number of people working in the specified department

```
SET SERVEROUTPUT ON;
DECLARE
   V_HOWMANY NUMBER(3);
   V_DEPTNO DEPARTMENTS.department_id%TYPE := &P_DEPTNO;
BEGIN
   SELECT COUNT(*) INTO V_HOWMANY FROM employees
   WHERE department_id = V_DEPTNO;
   DBMS_OUTPUT.PUT_LINE (V_HOWMANY  || ' employee(s)
    work for department number ' ||V_DEPTNO);
END;
/
```

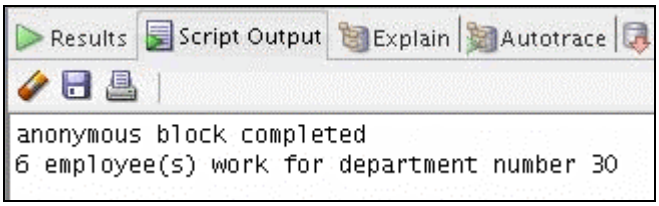
2. When the block is run, a substitution variable window appears. Enter a valid department number and click OK.



The output results should look similar to the following:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.



Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Additional Practices and Solutions for Lesson 5

In these practices, you use control structures to direct the logic of program flow.

Practice 5-1: Writing Control Structures

1. Write a PL/SQL block to accept a year input and check whether it is a leap year. **Hint**: The year should be exactly divisible by 4 but not divisible by 100, or it should be divisible by 400.
2. Test your solution by using the following table. For example, if the year entered is 1990, the output should be "1990 is not a leap year."

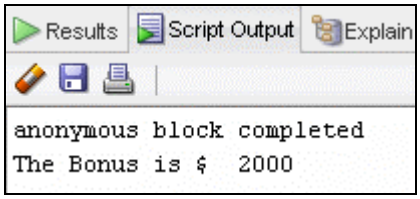
1990	Not a leap year
2000	Leap year
1996	Leap year
1886	Not a leap year
1992	Leap year
1824	Leap year

Practice 5-2: Writing Control Structures

1. Write a PL/SQL block to store the monthly salary of an employee in a substitution variable. The PL/SQL block should:
   - Calculate the annual salary as salary * 12
   - Calculate the bonus as indicated in the following table:

Annual Salary	**Bonus**
>= 20,000	2,000
19,999–10,000	1,000
<= 9,999	500

   - Display the amount of the bonus in the Script Output window in the following format:



2. Test the PL/SQL for the following test cases:

Monthly Salary	**Bonus**
3000	2000
1200	1000
800	500

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Solution 5-1: Writing Control Structures

1. Write a PL/SQL block to accept a year input and check whether it is a leap year.
   Hint: The year should be exactly divisible by 4 but not divisible by 100, or it should be divisible by 400.

```
SET SERVEROUTPUT ON;
DECLARE
   v_YEAR NUMBER(4) := &P_YEAR;
   v_REMAINDER1 NUMBER(5,2);
   v_REMAINDER2 NUMBER(5,2);
   v_REMAINDER3 NUMBER(5,2);
BEGIN
   v_REMAINDER1 := MOD(v_YEAR,4);
   v_REMAINDER2 := MOD(v_YEAR,100);
   v_REMAINDER3 := MOD(v_YEAR,400);
   IF ((v_REMAINDER1 = 0 AND v_REMAINDER2 <> 0 ) OR
       v_REMAINDER3 = 0) THEN
      DBMS_OUTPUT.PUT_LINE(v_YEAR || ' is a leap year');
   ELSE
      DBMS_OUTPUT.PUT_LINE(v_YEAR || ' is not a leap
      year');
   END IF;
END;
/
```

2. Test your solution by using the following table. For example, if the year entered is 1990, the output should be "1990 is not a leap year."

1990	Not a leap year
2000	Leap year
1996	Leap year
1886	Not a leap year
1992	Leap year
1824	Leap year

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

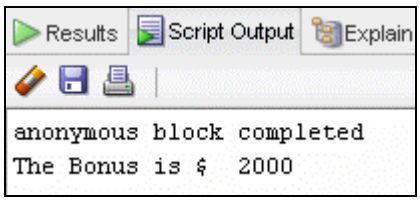
Solution 5-2: Writing Control Structures

1. Write a PL/SQL block to store the monthly salary of an employee in a substitution variable. The PL/SQL block should:

   - Calculate the annual salary as salary * 12
   - Calculate the bonus as indicated in the following table:

Annual Salary	Bonus
>= 20,000	2,000
19,999–10,000	1,000
<= 9,999	500

   - Display the amount of the bonus in the Script Output window in the following format:




```
SET SERVEROUTPUT ON;
DECLARE
   V_SAL        NUMBER(7,2) := &M_SALARY;
   V_BONUS      NUMBER(7,2);
   V_ANN_SALARY NUMBER(15,2);
BEGIN
   V_ANN_SALARY := V_SAL * 12;
   IF V_ANN_SALARY >= 20000 THEN
      V_BONUS := 2000;
   ELSIF V_ANN_SALARY <= 19999 AND V_ANN_SALARY >=10000 THEN
      V_BONUS := 1000;
   ELSE
      V_BONUS := 500;
   END IF;
   DBMS_OUTPUT.PUT_LINE ('The Bonus is $  ' ||
     TO_CHAR(V_BONUS));
END;
/
```

2. Test the PL/SQL for the following test cases:

Monthly Salary	Bonus
3000	2000
1200	1000
800	500

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

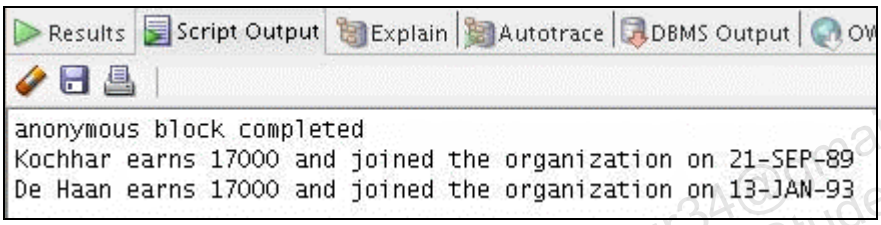
Additional Practices and Solutions for Lessons 6 and 7

In the following exercises, you practice using associative arrays (this topic is covered in lesson 6) and explicit cursors (this topic is covered in lesson 7). In the first exercise, you define and use an explicit cursor to fetch data. In the second exercise, you combine the use of associative arrays with an explicit cursor to output data that meets a certain criteria.

Practice 6/7-1: Fetching Data with an Explicit Cursor

In this practice, you create a PL/SQL block to perform the following:

1. Declare a cursor named `EMP_CUR` to select the employee's last name, salary, and hire date from the `EMPLOYEES` table
2. Process each row from the cursor, and if the salary is greater than 15,000 and the hire date is later than 01-FEB-1988, display the employee name, salary, and hire date in the format shown in the following sample output:



Practice 6/7-2: Using Associative Arrays and Explicit Cursors

In this practice, you create a PL/SQL block to retrieve and output the last name and department ID of each employee from the `EMPLOYEES` table for those employees whose `EMPLOYEE_ID` is less than 115.

In the PL/SQL block, use a cursor `FOR` loop strategy instead of the `OPEN` / `FETCH` / `CLOSE` cursor methods used in the previous practice.

1. In the declarative section:
   - Create two associative arrays. The unique key column for both arrays should be of the `BINARY INTEGER` data type. One array holds the employee's last name and the other holds the department ID.
   - Declare a cursor that selects the last name and department ID for employees whose ID is less than 115
   - Declare the appropriate counter variable to be used in the executable section
2. In the executable section, use a cursor `FOR` loop (covered in lesson 7) to access the cursor values, assign them to the appropriate associative arrays, and output those values from the arrays. The correct output should return 15 rows, in the following format:

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (ajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

```
anonymous block completed
Employee: King is in department number: 90
Employee: Kochhar is in department number: 90
Employee: De Haan is in department number: 90
```

Solution 6/7-1: Fetching Data with an Explicit Cursor

In this practice, you create a PL/SQL block to perform the following:

1. Declare a cursor named `EMP_CUR` to select the employee's last name, salary, and hire date from the `EMPLOYEES` table

```
SET SERVEROUTPUT ON;
DECLARE
  CURSOR C_EMP_CUR IS
    SELECT  last_name,salary,hire_date FROM EMPLOYEES;
  V_ENAME VARCHAR2(25);
  v_SAL   NUMBER(7,2);
  V_HIREDATE DATE;
```

2. Process each row from the cursor, and if the salary is greater than 15,000 and the hire date is later than 01-FEB-1988, display the employee name, salary, and hire date in the format shown in the following sample output:

```
BEGIN
  OPEN C_EMP_CUR;
  FETCH C_EMP_CUR INTO V_ENAME,V_SAL,V_HIREDATE;
  WHILE C_EMP_CUR%FOUND
  LOOP
    IF V_SAL > 15000 AND V_HIREDATE >=
       TO_DATE('01-FEB-1988','DD-MON-YYYY') THEN
        DBMS_OUTPUT.PUT_LINE (V_ENAME || ' earns '
        || TO_CHAR(V_SAL)|| ' and joined the organization on '
        || TO_DATE(V_HIREDATE,'DD-Mon-YYYY'));
    END IF;
    FETCH C_EMP_CUR INTO V_ENAME,V_SAL,V_HIREDATE;
  END LOOP;
  CLOSE C_EMP_CUR;
END;
/
```

```
anonymous block completed
Kochhar earns 17000 and joined the organization on 21-SEP-89
De Haan earns 17000 and joined the organization on 13-JAN-93
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution 6/7-2: Using Associative Arrays and Explicit Cursors

In this practice, you create a PL/SQL block to retrieve and output the last name and department ID of each employee from the `EMPLOYEES` table for those employees whose `EMPLOYEE_ID` is less than 115.

In the PL/SQL block, use a cursor `FOR` loop strategy instead of the `OPEN` / `FETCH` / `CLOSE` cursor methods used in the previous practice.

1. In the declarative section:
   - Create two associative arrays. The unique key column for both arrays should be of the `BINARY INTEGER` data type. One array holds the employee's last name and the other holds the department ID.
   - Declare a counter variable to be used in the executable section
   - Declare a cursor that selects the last name and department ID for employees whose ID is less than 115

```
SET SERVEROUTPUT ON;
DECLARE
   TYPE Table_Ename IS table of employees.last_name%TYPE
         INDEX BY BINARY_INTEGER;
   TYPE Table_dept IS table of employees.department_id%TYPE
         INDEX BY BINARY_INTEGER;
   Tename Table_Ename;
   Tdept  Table_dept;
   i BINARY_INTEGER :=0;
   CURSOR Namedept IS SELECT last_name,department_id
     FROM employees WHERE employee_id < 115;
```

2. In the executable section, use a cursor `FOR` loop (covered in lesson 7) to access the cursor values, assign them to the appropriate associative arrays, and output those values from the arrays.

```
BEGIN
   FOR emprec in Namedept
   LOOP
     i          := i +1;
     Tename(i)  := emprec.last_name;
     Tdept(i)   := emprec.department_id;
     DBMS_OUTPUT.PUT_LINE ('Employee: ' || Tename(i) ||
         ' is in department number: ' || Tdept(i));
   END LOOP;
END;
/
```

The correct output should return 15 rows, similar to the following:

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Results | Script Output | Explain | Autotrace | DBMS

```
anonymous block completed
Employee: King is in department number: 90
Employee: Kochhar is in department number: 90
Employee: De Haan is in department number: 90
Employee: Hunold is in department number: 60
Employee: Ernst is in department number: 60
Employee: Austin is in department number: 60
Employee: Pataballa is in department number: 60
Employee: Lorentz is in department number: 60
Employee: Greenberg is in department number: 100
Employee: Faviet is in department number: 100
Employee: Chen is in department number: 100
Employee: Sciarra is in department number: 100
Employee: Urman is in department number: 100
Employee: Popp is in department number: 100
Employee: Raphaely is in department number: 30
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Additional Practice and Solution for Lesson 8

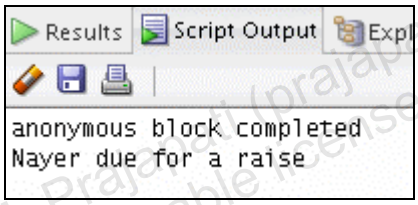
Practice 8-1: Handling Exceptions

For this exercise, you must first create a table to store some results. Run the `lab_ap_08.sql` script that creates the table for you. The script looks like the following:

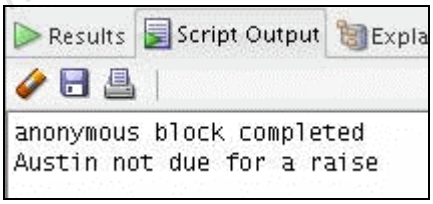
```
CREATE TABLE analysis
       (ename Varchar2(20), years Number(2), sal Number(8,2)
        );
```

In this practice, you write a PL/SQL block that handles an exception, as follows:

1. Declare variables for the employee last name, salary, and hire date. Use a substitution variable for the employee last name. Then, query the `employees` table for the `last_name`, `salary`, and `hire_date` of the specified employee.
2. If the employee has been with the organization for more than five years, and if that employee's salary is less than 3,500, raise an exception. In the exception handler, perform the following:
   - Output the following information: employee last name and the message "due for a raise," similar to the following:

   - Insert the last name, years of service, and salary into the `analysis` table.
3. If there is no exception, output the employee last name and the message "not due for a raise," similar to the following:

4. Verify the results by querying the `analysis` table. Use the following test cases to test the PL/SQL block.

LAST_NAME	MESSAGE
Austin	Not due for a raise
Nayer	Due for a raise
Fripp	Not due for a raise
Khoo	Due for a raise

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur1@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Solution 8-1: Handling Exceptions

For this exercise, you must first create a table to store some results. Run the `lab_ap_08.sql` script that creates the table for you. The script looks similar to the following:

```
CREATE TABLE analysis
       (ename Varchar2(20), years Number(2), sal Number(8,2)
        );
```

In this practice, you write a PL/SQL block that handles an exception, as follows:

1. Declare variables for the employee last name, salary, and hire date. Use a substitution variable for the employee last name. Then, query the `employees` table for the `last_name`, `salary`, and `hire_date` of the specified employee.
2. If the employee has been with the organization for more than five years, and if that employee's salary is less than 3,500, raise an exception. In the exception handler, perform the following:
   - Output the following information: employee last name and the message "due for a raise."
   - Insert the employee name, years of service, and salary into the `analysis` table.
3. If there is no exception, output the employee last name and the message "not due for a raise."

```
SET SERVEROUTPUT ON;
DECLARE
  E_DUE_FOR_RAISE EXCEPTION;
  V_HIREDATE EMPLOYEES.HIRE_DATE%TYPE;
  V_ENAME EMPLOYEES.LAST_NAME%TYPE :=  INITCAP( '& B_ENAME');
  V_SAL EMPLOYEES.SALARY%TYPE;
  V_YEARS NUMBER(2);
BEGIN
  SELECT LAST_NAME,SALARY,HIRE_DATE
  INTO V_ENAME,V_SAL,V_HIREDATE
  FROM employees WHERE last_name = V_ENAME;
  V_YEARS := MONTHS_BETWEEN(SYSDATE,V_HIREDATE)/12;
  IF V_SAL < 3500 AND V_YEARS > 5  THEN
      RAISE E_DUE_FOR_RAISE;
  ELSE
      DBMS_OUTPUT.PUT_LINE (' not due for a raise');
  END IF;
EXCEPTION
  WHEN E_DUE_FOR_RAISE THEN
    BEGIN
      DBMS_OUTPUT.PUT_LINE (V_NAME || ' due for a raise');
     INSERT INTO ANALYSIS(ENAME,YEARS,SAL)
       VALUES (V_ENAME,V_YEARS,V_SAL);
   END;
END;
/
```

Author: Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

4. Verify the results by querying the `analysis` table. Use the following test cases to test the PL/SQL block.

LAST_NAME	MESSAGE
Austin	Not due for a raise
Nayer	Due for a raise
Fripp	Not due for a raise
Khoo	Due for a raise

```
SELECT * FROM analysis;
```

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.

Unauthorized reproduction or distribution prohibited. Copyright© 2014, Oracle and/or its affiliates.

Ankur M. Prajapati (prajapati.ankur34@gmail.com) has a non-transferable license to use this Student Guide.