



Integrated Cloud Applications & Platform Services



# Oracle Database 12c R2: PL/SQL Fundamentals

Student Guide

D80182GC20

Edition 2.0 | April 2017 | D98673

Learn more from Oracle University at [education.oracle.com](http://education.oracle.com)



Oracle Internal & Oracle Academy Only

**Author**

Jayashree Sharma

**Technical Contributors  
and Reviewers**

Bryan Roberts

Miyuki Osato

Nancy Greenberg

Suresh Rajan

**Editor**

Vijayalakshmi Narasimhan

**Graphic Designers**

Prakash Dharmalingam

Seema Bopaiah

Maheshwari Krishnamurthy

**Publishers**

Syed Imtiaz Ali

Sujatha Nagendra

Veena Narasimhan

**Copyright © 2017, Oracle and/or its affiliates. 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 and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

## Contents

### 1 Introduction

- Lesson Objectives 1-2
- Lesson Agenda 1-3
- Course Objectives 1-4
- Course Road Map 1-5
- Lesson Agenda 1-8
- Human Resources (HR) Schema for This Course 1-9
- Course Agenda 1-10
- Class Account Information 1-11
- Appendices and Practices Used in This Course 1-12
- Lesson Agenda 1-13
- Oracle Database 12c: Focus Areas 1-14
- Oracle Database 12c 1-15
- Lesson Agenda 1-17
- PL/SQL Development Environments 1-18
- Oracle SQL Developer 1-19
- Specifications of SQL Developer 1-20
- SQL Developer 4.1.3 Interface 1-21
- Coding PL/SQL in SQL\*Plus 1-22
- Lesson Agenda 1-23
- Oracle SQL and PL/SQL Documentation 1-24
- Additional Resources 1-25
- Summary 1-26
- Practice 1 Overview: Getting Started 1-27

### 2 Introduction to PL/SQL

- Course Road Map 2-2
- Objectives 2-3
- Agenda 2-4
- Limitations of SQL 2-5
- Why PL/SQL? 2-6
- Why PL/SQL 2-7
- About PL/SQL 2-8
- Benefits of PL/SQL 2-9
- PL/SQL Runtime Architecture 2-11

PL/SQL Block Structure	2-12
Agenda	2-13
Block Types	2-14
Examining an Anonymous Block	2-16
Executing an Anonymous Block	2-17
Agenda	2-18
Enabling Output of a PL/SQL Block	2-19
Viewing the Output of a PL/SQL Block	2-20
Quiz	2-21
Summary	2-22
Practice 2: Overview	2-23

### **3 Declaring PL/SQL Variables**

Course Road Map	3-2
Objectives	3-3
Agenda	3-4
Variables	3-5
Variables in PL/SQL	3-6
Requirements for Variable Names	3-8
Using Variables in PL/SQL	3-9
Declaring and Initializing PL/SQL Variables	3-10
Agenda	3-11
Declaring and Initializing PL/SQL Variables	3-12
Initializing Variables Through a SELECT Statement	3-13
Types of Variables	3-15
Declaring Variables	3-16
Guidelines for Declaring and Initializing PL/SQL Variables	3-17
Guidelines for Declaring PL/SQL Variables	3-18
Naming Conventions of the PL/SQL Structures Used in This Course	3-19
Data Types for Strings	3-20
Delimiters in String Literals	3-21
Data Types for Numeric values	3-22
Data Types for Date and Time values	3-23
Data Type Conversion	3-25
Agenda	3-28
The %TYPE Attribute	3-29
Declaring Variables with the %TYPE Attribute	3-30
Declaring Boolean Variables	3-31
LOB Data Type Variables	3-32
Composite Data Types: Records and Collections	3-33
Agenda	3-34

Bind Variables 3-35  
Bind Variables: Examples 3-36  
Using AUTOPRINT with Bind Variables 3-37  
Quiz 3-38  
Summary 3-39  
Practice 3: Overview 3-40

#### **4 Writing Executable Statements**

Course Road Map 4-2  
Objectives 4-3  
Agenda 4-4  
Lexical Units in a PL/SQL Block 4-5  
PL/SQL Block Syntax and Guidelines 4-7  
Commenting Code 4-8  
SQL Functions in PL/SQL 4-9  
SQL Functions in PL/SQL: Examples 4-10  
Using Sequences in PL/SQL blocks 4-11  
Using Sequences in PL/SQL Blocks 4-14  
Agenda 4-15  
Nested blocks 4-16  
Nested Blocks: Example 4-17  
Variable Scope and Visibility 4-18  
Using a Qualifier with Nested Blocks 4-20  
Challenge: Determining the Variable Scope 4-21  
Agenda 4-23  
Operators in PL/SQL 4-24  
Operators in PL/SQL: Examples 4-25  
Programming Guidelines 4-26  
Indenting Code 4-27  
Quiz 4-28  
Summary 4-29  
Practice 4: Overview 4-30

#### **5 Using SQL Statements Within a PL/SQL Block**

Course Road Map 5-2  
Objectives 5-3  
Agenda 5-4  
SQL Statements in PL/SQL 5-5  
SELECT Statements in PL/SQL 5-7  
Retrieving Data in PL/SQL: Example 5-9  
Retrieving Data in PL/SQL 5-10

Naming Ambiguities 5-11  
Avoiding Naming Ambiguities 5-12  
Agenda 5-13  
Using PL/SQL to Manipulate Data 5-14  
Insert Data: Example 5-15  
Update Data: Example 5-16  
Delete Data: Example 5-17  
Merging Rows 5-18  
Agenda 5-20  
SQL Cursor 5-21  
SQL Cursor Attributes for Implicit Cursors 5-23  
Quiz 5-25  
Summary 5-26  
Practice 5: Overview 5-27

## 6 Writing Control Structures

Course Road Map 6-2  
Objectives 6-3  
PL/SQL Control Structures 6-4  
Agenda 6-5  
IF Statement 6-6  
IF-ELSIF Statements 6-7  
Simple IF Statement 6-8  
IF THEN ELSE Statement 6-9  
IF ELSIF ELSE Clause 6-10  
NULL Value an in IF Statement 6-11  
Agenda 6-12  
CASE Expressions 6-13  
Searched CASE Expressions 6-14  
CASE Statement 6-16  
Handling Nulls 6-17  
Logic Tables 6-18  
Boolean Expression or Logical Expression? 6-19  
Agenda 6-20  
Iterative Control: LOOP Statements 6-21  
Basic Loops 6-22  
Basic Loop: Example 6-23  
WHILE Loops 6-24  
WHILE Loops: Example 6-25  
FOR Loops 6-26  
FOR Loops: Example 6-28

FOR Loop Rules 6-29  
Suggested Use of Loops 6-30  
Nested Loops and Labels 6-31  
Nested Loops and Labels: Example 6-32  
PL/SQL CONTINUE Statement 6-33  
PL/SQL CONTINUE Statement: Example 1 6-34  
PL/SQL CONTINUE Statement: Example 2 6-35  
Quiz 6-36  
Summary 6-37  
Practice 6: Overview 6-38

## 7 Working with Composite Data Types

Course Road Map 7-2  
Objectives 7-3  
Agenda 7-4  
Composite Data Types 7-5  
PL/SQL Records Versus Collections 7-7  
Agenda 7-8  
PL/SQL Records 7-9  
Creating a PL/SQL Record 7-10  
Creating a PL/SQL Record: Example 7-11  
PL/SQL Record Structure 7-12  
%ROWTYPE Attribute 7-14  
Creating a PL/SQL Record: Example 7-16  
Advantages of Using the %ROWTYPE Attribute 7-17  
Another %ROWTYPE Attribute: Example 7-18  
Inserting a Record by Using %ROWTYPE 7-19  
Updating a Row in a Table by Using a Record 7-20  
Agenda 7-21  
Associative Arrays (INDEX BY Tables) 7-22  
Associative Array Structure 7-24  
Steps to Create an Associative Array 7-25  
Creating and Accessing Associative Arrays 7-26  
Associative Arrays with Record values 7-27  
Using Collection Methods 7-28  
Using Collection Methods with Associative Arrays 7-29  
Nested Tables 7-30  
Nested Tables: Syntax and Usage 7-31  
Variable-Sized Arrays (Varrays) 7-33  
VARARRAYs: Syntax and Usage 7-34  
Summarizing Collections 7-35

Quiz 7-36

Summary 7-37

Practice 7: Overview 7-38

## **8 Using Explicit Cursors**

Course Road Map 8-2

Objectives 8-3

Agenda 8-4

Cursors 8-5

Implicit Cursors 8-6

Explicit Cursor 8-7

Controlling Explicit Cursors 8-8

Agenda 8-9

Declaring the Cursor 8-10

Opening the Cursor 8-12

Fetching Data from the Cursor 8-13

Closing the Cursor 8-15

Cursors and Records 8-16

Cursor FOR Loops 8-17

Explicit Cursor Attributes 8-19

%ISOPEN Attribute 8-20

%ROWCOUNT and %NOTFOUND: Example 8-21

Cursor FOR Loops Using Subqueries 8-22

Agenda 8-23

Cursors with Parameters 8-24

Agenda 8-26

FOR UPDATE Clause 8-27

WHERE CURRENT OF Clause 8-29

WHERE CURRENT OF Clause: Example 8-30

Quiz 8-32

Summary 8-33

Practice 8: Overview 8-34

## **9 Handling Exceptions**

Course Road Map 9-2

Objectives 9-3

Agenda 9-4

What Is an Exception? 9-5

Handling an Exception: Example 9-6

Understanding Exceptions with PL/SQL 9-7

Handling Exceptions 9-8

Exception Types 9-9  
Agenda 9-10  
Syntax to Trap Exceptions 9-11  
Guidelines for Trapping Exceptions 9-13  
Trapping Internally Predefined Exceptions 9-14  
Internally Defined Exception Trapping: Example 9-15  
Trapping Predefined Exceptions 9-16  
Functions for Trapping Exceptions 9-19  
Trapping User-Defined Exceptions 9-21  
RAISE Statement 9-22  
Trapping User-Defined Exceptions 9-23  
Propagating Exceptions in a Sub-Block 9-24  
The RAISE\_APPLICATION\_ERROR Procedure 9-25  
Quiz 9-28  
Summary 9-29  
Practice 9: Overview 9-30

## **10 Introducing Stored Procedures and Functions**

Course Road Map 10-2  
Objectives 10-3  
Agenda 10-4  
What Are PL/SQL Subprograms? 10-5  
Differences Between Anonymous Blocks and Subprograms 10-7  
Agenda 10-8  
Procedure: Syntax 10-9  
Creating a Procedure 10-10  
Invoking a Procedure 10-12  
Agenda 10-13  
Function: Syntax 10-14  
Creating a Function 10-15  
Invoking a Function 10-16  
Passing a Parameter to the Function 10-17  
Invoking the Function with a Parameter 10-18  
Quiz 10-19  
Summary 10-20  
Practice 10: Overview 10-21

## **11 Oracle Cloud Overview**

Lesson Objectives 11-2  
Lesson Agenda 11-3  
Introduction to Oracle Cloud 11-4

Oracle Cloud Services 11-5  
Cloud Deployment Models 11-6  
Lesson Agenda 11-7  
Evolving from On-premises to Exadata Express 11-8  
What is in Exadata Express? 11-9  
Exadata Express for Users 11-10  
Exadata Express for Developers 11-11  
Getting Started with Exadata Express 11-12  
Oracle Exadata Express Cloud Service 11-13  
Getting Started with Exadata Express 11-14  
Managing Exadata 11-15  
Service Console 11-16  
Web Access through Service Console 11-17  
Client Access Configuration through Service Console 11-18  
Database Administration through Service Console 11-19  
SQL Workshop 11-20  
Connecting through Database Clients 11-22  
Enabling SQL\*Net Access for Client Applications 11-23  
Downloading Client Credentials 11-24  
Connecting Oracle SQL Developer 11-25  
Connecting Oracle SQLcl 11-26  
Summary 11-27

## A Using SQL Developer

Objectives A-2  
What Is Oracle SQL Developer? A-3  
Specifications of SQL Developer A-4  
SQL Developer 4.1.3 Interface A-5  
Creating a Database Connection A-7  
Browsing Database Objects A-10  
Displaying the Table Structure A-11  
Browsing Files A-12  
Creating a Schema Object A-13  
Creating a New Table: Example A-14  
Using the SQL Worksheet A-15  
Executing SQL Statements A-19  
Saving SQL Scripts A-20  
Executing Saved Script Files: Method 1 A-21  
Executing Saved Script Files: Method 2 A-22  
Formatting the SQL Code A-23  
Using Snippets A-24

Using Snippets: Example	A-25
Debugging Procedures and Functions	A-26
Database Reporting	A-27
Creating a User-Defined Report	A-28
Search Engines and External Tools	A-29
Setting Preferences	A-30
Resetting the SQL Developer Layout	A-32
Data Modeler in SQL Developer	A-33
Summary	A-34

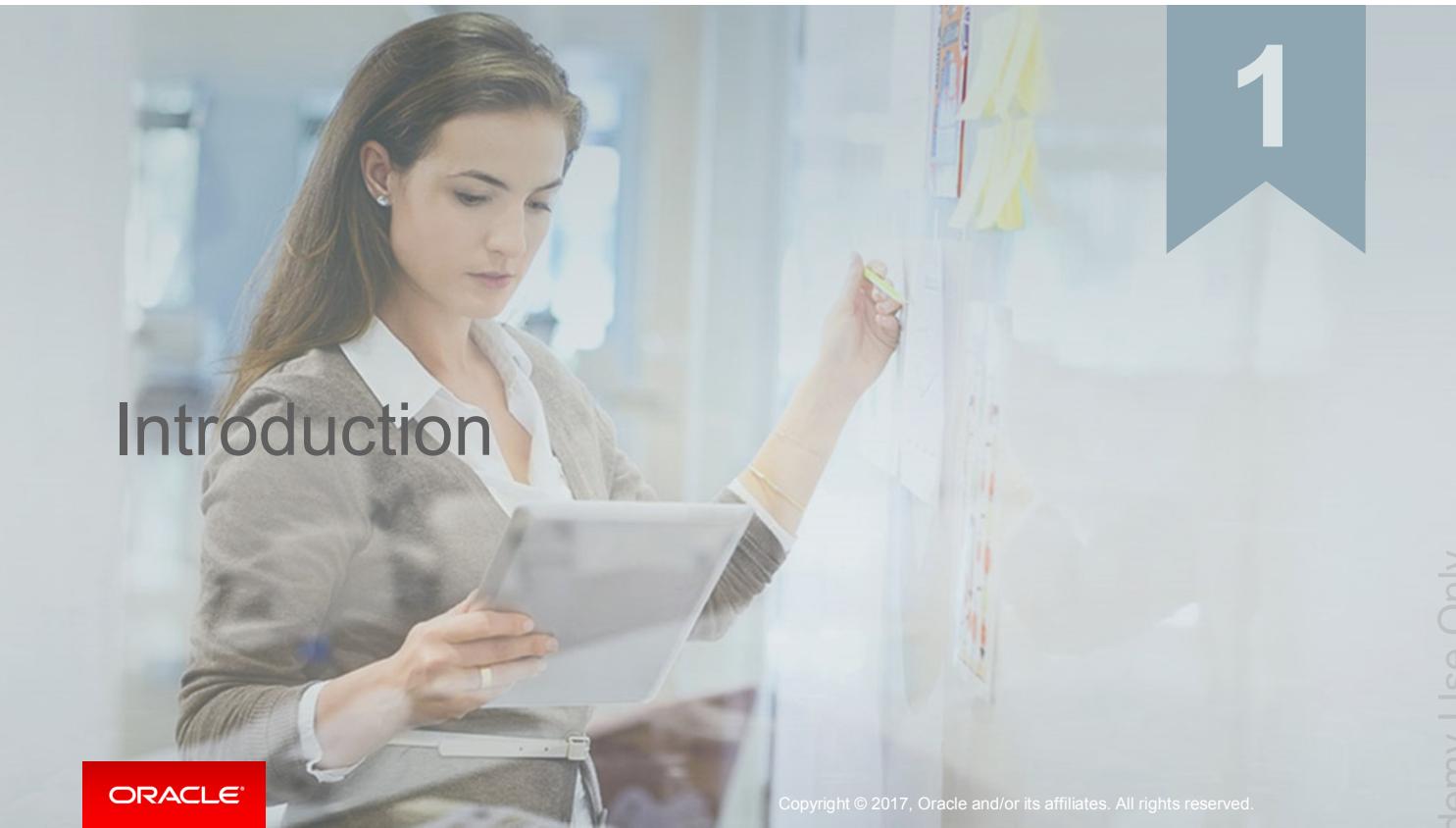
## B Using SQL\*Plus

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

## C Commonly Used SQL Commands

Objectives	C-2
Basic SELECT Statement	C-3
SELECT Statement	C-4
WHERE Clause	C-5
ORDER BY Clause	C-6
GROUP BY Clause	C-7
Data Definition Language	C-8
CREATE TABLE Statement	C-9
ALTER TABLE Statement	C-10
DROP TABLE Statement	C-11
GRANT Statement	C-12
Privilege Types	C-13
REVOKE Statement	C-14

TRUNCATE TABLE Statement	C-15
Data Manipulation Language	C-16
INSERT Statement	C-17
UPDATE Statement Syntax	C-18
DELETE Statement	C-19
Transaction Control Statements	C-20
COMMIT Statement	C-21
ROLLBACK Statement	C-22
SAVEPOINT Statement	C-23
Joins	C-24
Types of Joins	C-25
Qualifying Ambiguous Column Names	C-26
Natural Join	C-27
Equijoins	C-28
Retrieving Records with Equijoins	C-29
Additional Search Conditions Using the AND and WHERE Operators	C-30
Retrieving Records with Nonequijoins	C-31
Retrieving Records by Using the USING Clause	C-32
Retrieving Records by Using the ON Clause	C-33
Left Outer Join	C-34
Right Outer Join	C-35
Full Outer Join	C-36
Self-Join: Example	C-37
Cross Join	C-38
Summary	C-39
<b>D REF Cursors</b>	
Cursor Variables	D-2
Using Cursor Variables	D-3
Defining REF CURSOR Types	D-4
Using the OPEN-FOR, FETCH, and CLOSE Statements	D-7
Example of Fetching	D-10



1

# Introduction

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## 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 this course
- Identify the available user interface environments that can be used in this course
- Reference the available appendixes, documentation, and other resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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. The course discusses the fundamentals of PL/SQL. You learn how to write PL/SQL code blocks, and are introduced to the concepts of procedures and functions.

You are also introduced to a programming tool, SQL Developer 4.1.3.

# Lesson Agenda

- Course objectives and course agenda
- The schema and appendixes used in this course
- Overview of Oracle Database 12c and related products
- Available PL/SQL development environments
- Oracle documentation and additional resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## 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 runtime errors
- Describe stored procedures and functions



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

# Course Road Map

Lesson 1: Course Overview

## Unit 1: Introducing PL/SQL

Unit 2: Programming with PL/SQL

Unit 3: Working with PL/SQL Code

▶ Lesson 2: Introduction to PL/SQL

▶ Lesson 3: Declaring PL/SQL Variables

▶ Lesson 4: Writing Anonymous PL/SQL blocks

▶ Lesson 5: Using SQL Statements in PLSQL blocks



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide provides a graphical representation of the overall course structure.

The course is organized into three units: Introducing PL/SQL, Programming with PL/SQL, and Working with PL/SQL Code.

In the first unit, we have four lessons:

1. Introduction to PL/SQL
2. Declaring PL/SQL Variables
3. Writing Anonymous PL/SQL Blocks
4. Using SQL Statements in PL/SQL Blocks.

You learn to write code in PL/SQL and define some anonymous PL/SQL blocks in this unit.

# Course Road Map

Lesson 1: Course Overview

Unit 1: Introducing PL/SQL

**Unit 2: Programming with PL/SQL**

Unit 3: Working with PL/SQL Code

▶ Lesson 6: Writing Control Structures

▶ Lesson 7: Working with Composite Data Types

▶ Lesson 8: Using Explicit Cursors



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In Unit 2, you learn to program with PL/SQL by using control structures, composite data types, and explicit cursors.

# Course Road Map

Lesson 1: Course Overview

Unit 1: Introducing PL/SQL

Unit 2: Programming with PL/SQL

**Unit 3: Working with PL/SQL Code**

▶ Lesson 9: Handling Exceptions

▶ Lesson 10: Creating Procedures and Functions

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In Unit 3, you learn how to handle exceptions that might occur during the execution of PL/SQL blocks. You are also introduced to the concept of procedures and functions in PL/SQL.

## Lesson Agenda

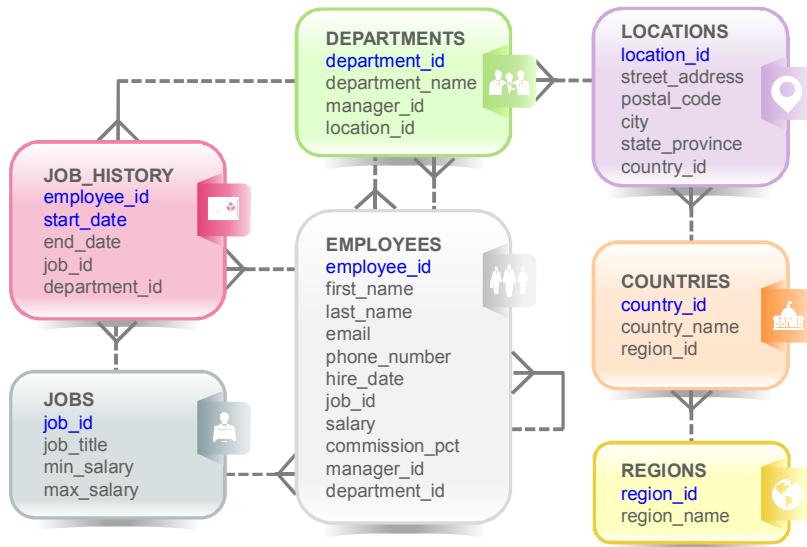
- Course objectives and course agenda
- The schema and appendixes used in this course
- Overview of Oracle Database 12c and related products
- Available PL/SQL development environments
- Oracle documentation and additional resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Human Resources (HR) Schema for This Course



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The Human Resources (HR) schema is part of the Oracle Sample Schemas that can be installed in an Oracle database. You will use the HR schema in all the practice lessons of the course. The fields marked in blue are the primary key attributes for the respective entities.

A description of each of the tables in the HR schema is as follows:

- The REGIONS table contains rows that represent a region such as the Americas or Asia.
- The COUNTRIES table contains rows for countries, each of which is associated with a region.
- The LOCATIONS table contains the specific address of a specific office, warehouse, or production site of a company in a particular country.
- The DEPARTMENTS table shows details of the departments in which employees work. Each department may have a relationship that represents the department manager in the EMPLOYEES table.
- The EMPLOYEES table contains details of each employee working in a department. Some employees may not be assigned to any department.
- The JOBS table contains the job types that can be held by each employee.
- The JOB\_HISTORY table contains the job history of 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:
  1. Introduction
  2. Introduction to PL/SQL
  3. Declaring PL/SQL Variables
  4. Writing Executable Statements
  5. Using SQL Statements within a PL/SQL Block
  6. Writing Control Structures
- Day 2:
  7. Working with Composite Data Types
  8. Using Explicit Cursors
  9. Handling Exceptions
  10. Introducing Stored Procedures and Functions

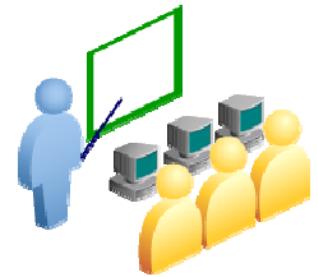


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## 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.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Appendices and Practices Used in This Course

- Appendix A: Table Descriptions and Data
- Appendix B: Using SQL Developer
- Appendix C: Using SQL\*Plus
- Appendix D: Commonly Used SQL Commands
- Activity Guide: Practices and Solutions



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Lesson Agenda

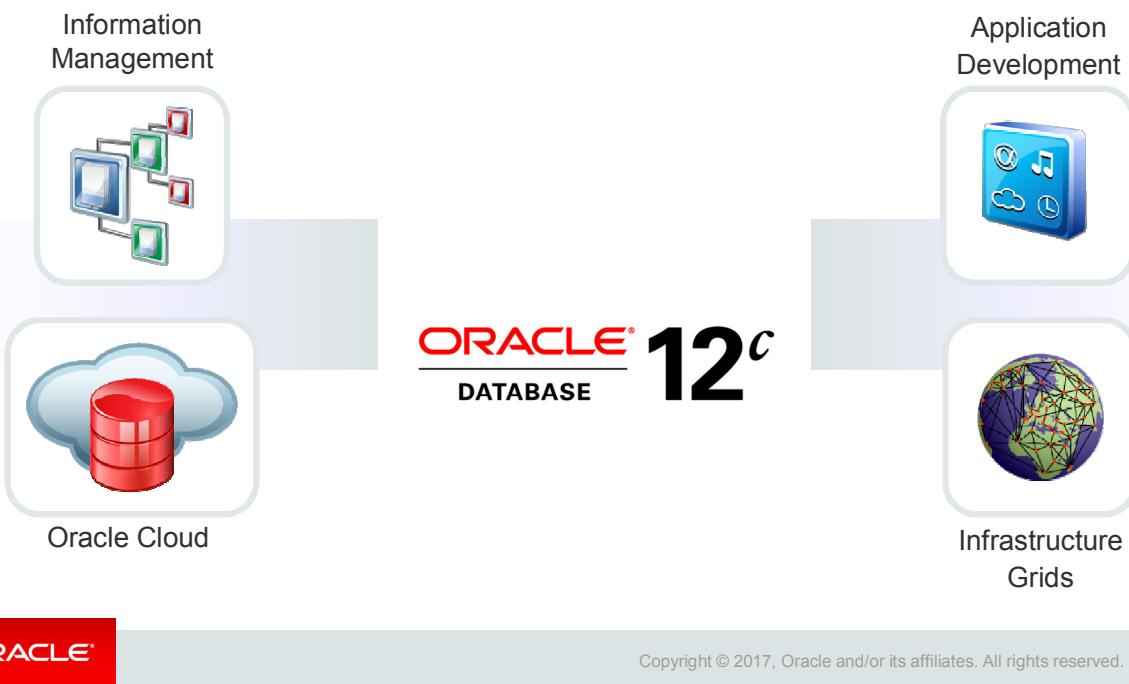
- Course objectives and course agenda
- The schema and appendixes used in this course
- Overview of Oracle Database 12c and related products
- Available PL/SQL development environments
- Oracle documentation and additional resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Oracle Database 12c: Focus Areas



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

By using Oracle Database 12c, you can utilize the following features across focus areas:

- With the **Infrastructure Grid** technology of Oracle, you can pool low-cost servers and storage to form systems that deliver the highest quality of service in terms of manageability, high availability, and performance. Oracle Database 12c also helps you to consolidate and extend the benefits of grid computing and manage changes in a controlled and cost-effective manner.
- Oracle Database 12c enables **Information Management** by providing capabilities in content management, information integration, and information lifecycle management areas. You can manage content of advanced data types such as Extensible Markup Language (XML), text, spatial, multimedia, medical imaging, and semantic technologies using the features provided by Oracle.
- With Oracle Database 12c, you can manage all the major **Application Development** environments such as PL/SQL, Java/JDBC, .NET, Windows, PHP, SQL Developer, and Application Express.
- You can now plug into **Oracle Cloud** with Oracle Database 12c. This will help you to standardize, consolidate, and automate database services on the cloud.

## Oracle Database 12c



High Availability

ORACLE<sup>®</sup> DATABASE 12<sup>c</sup>

Manageability



Performance



Security



Information Integration

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Imagine you have an organization that needs to support multiple terabytes of information for users who demand fast and secure access to business applications round the clock. The database systems must be reliable and must be able to recover quickly in the event of any kind of failure. Oracle Database 12c is designed to help organizations manage infrastructure grids easily and deliver high-quality service:

- **Manageability:** By using some of the change assurance, management automation, and fault diagnostics features, the database administrators (DBAs) can increase their productivity, reduce costs, minimize errors, and maximize quality of service. Some of the useful features that promote better management are the Database Replay facility, the SQL Performance Analyzer, the Automatic SQL Tuning facility, and Real-Time Database Operations Monitoring.

Enterprise Manager Database Express 12c is a web-based tool for managing Oracle databases. It greatly simplifies database performance diagnostics by consolidating the relevant database performance screens into a view called Database Performance Hub. DBAs get a single, consolidated view of the current real-time and historical view of the database performance across multiple dimensions such as database load, monitored SQL and PL/SQL, and Active Session History (ASH) on a single page for the selected time period.

- **High availability:** By using the high availability features, you can reduce the risk of down time and data loss. These features improve online operations and enable faster database upgrades.
- **Performance:** By using capabilities such as SecureFiles, Result Caches, and so on, you can greatly improve the performance of your database. Oracle Database 12c enables organizations to manage large, scalable, transactional, and data warehousing systems that deliver fast data access using low-cost modular storage.
- **Security:** Oracle Database 12c helps in protecting your information with unique secure configurations, data encryption and masking, and sophisticated auditing capabilities.
- **Information integration:** You can utilize Oracle Database 12c features to integrate data throughout the enterprise in a better way. You can also manage the changing data in your database by using Oracle Database 12c's advanced information lifecycle management capabilities.

## Lesson Agenda

- Course objectives and course agenda
- The schema and appendixes used in this course
- Overview of Oracle Database 12c and related products
- Available PL/SQL development environments
- Oracle documentation and additional resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# PL/SQL Development Environments

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

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use various tools provided by Oracle to write PL/SQL code. In this course, we primarily focus on SQL Developer. We also look at the usage of SQL\*Plus.

- **Oracle SQL Developer** is an integrated development environment, which allows database users and administrators to perform their database tasks in fewer clicks and keystrokes. For developers, it provides powerful editors for working with SQL, PL/SQL, stored Java procedures, and XML. You can run queries, generate execution plans, export data to different formats such as XML and HTML, and so on.
- **Oracle SQL\*Plus**: SQL\*Plus is an interactive and batch query tool that is installed with every installation of Oracle Database. It has a command-line user interface, a Windows Graphical User Interface (GUI), and the *i*SQL\*Plus web-based user interface. SQL\*Plus has its own commands and environment, and it provides access to Oracle Database.

You can use SQL\*Plus to generate reports interactively or as batch processes, and to output the results to a text file, a screen, or to an HTML file for browsing on the Internet. You can generate reports dynamically by using the HTML output facility of SQL\*Plus, or by using the dynamic reporting capability of *i*SQL\*Plus to run a script from a web page.

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

# 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 by using standard Oracle database authentication.
- You use SQL Developer in this course.



SQL Developer

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## What is Oracle SQL Developer?

Oracle SQL Developer is a free graphical tool that is designed to improve productivity and simplify the performance of everyday database tasks. The user interface of SQL Developer allows you to browse, create, and manage database objects.

Using SQL Developer, you can easily create and maintain stored procedures, test SQL statements, and view optimizer plans, with just a few clicks.

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 the objects in the database.

## Specifications of SQL Developer

- Is developed in Java
- Supports the Windows, Linux, and Mac OS X platforms
- Enables default connectivity by using the JDBC Thin driver
- Connects to Oracle Database version 9.2.0.1 and later
- Connects to Oracle Database on Cloud also



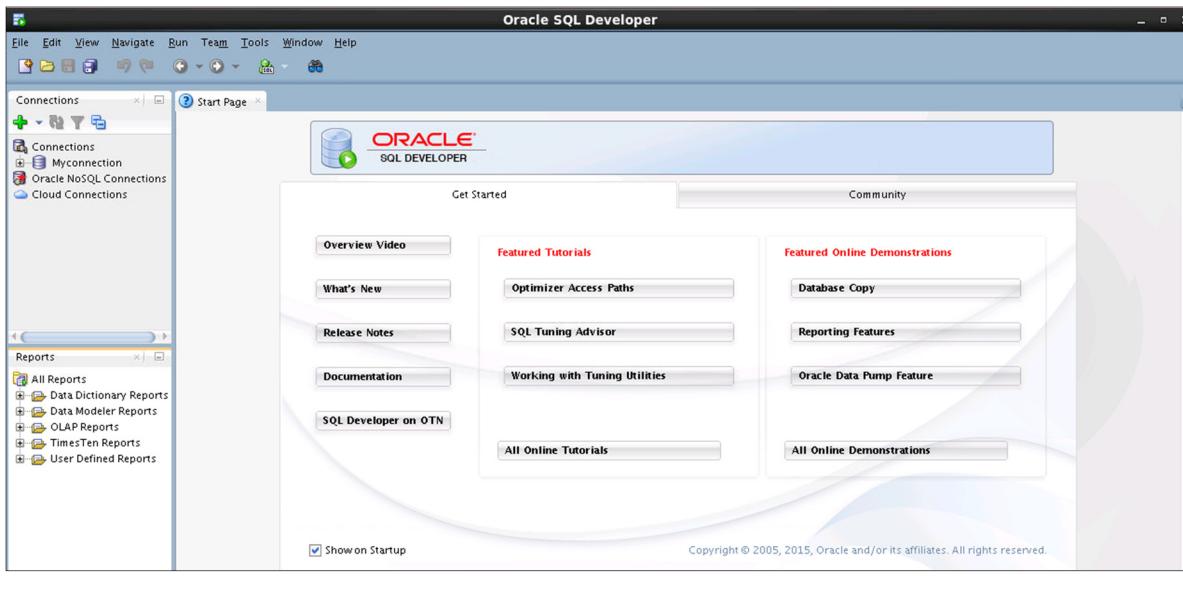
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Oracle SQL Developer is developed in Java, leveraging 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; therefore, no Oracle Home is required. SQL Developer does not require an installer. All you need to do is simply 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. SQL Developer allows you to connect to Oracle Database Service on cloud also.

# SQL Developer 4.1.3 Interface



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The SQL Developer 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.
- **Reports tab:** Identified by the Reports icon, this tab enables you to run predefined reports or create and add your own reports.
- **Start page:** The Start page gives you some links that are helpful when you use SQL Developer to create applications.

## General Navigation and Use

SQL Developer uses the left pane for navigation to find and select 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. You can start the connection creation wizard by clicking on “+” on the Connections tab.

## Coding PL/SQL in SQL\*Plus



A screenshot of the Oracle SQL\*Plus command-line interface. A red arrow points from a small icon of a smartphone displaying 'SQL> ...' towards the terminal window. The terminal window shows the following session:

```
oracle@EDRSR9P1:~/Desktop
File Edit View Search Terminal Help
Copyright (c) 1982, 2012, Oracle. All rights reserved.

Enter user-name: ora41
Enter password:
Last Successful login time: Mon Sep 2012 21:55:44 +00:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.2 - 64bit Beta
With the Partitioning, OLAP, Data Mining and Real Application Testing options

SQL> set serveroutput on
SQL> create or replace procedure hello is
 2 begin
 3   dbms_output.put_line('Hello Class!');
 4 end;
 5 /

Procedure created.

SQL> execute hello
Hello Class!

PL/SQL procedure successfully completed.

SQL>
```

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Oracle SQL\*Plus is a command-line interface that enables you to submit SQL statements and PL/SQL blocks for execution and receive 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 by using an icon or the command line

When you code PL/SQL subprograms by 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.

## Lesson Agenda

- Course objectives and course agenda
- The schema and appendixes used in this course
- Overview of Oracle Database 12c and related products
- Available PL/SQL development environments
- Oracle documentation and additional resources



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Oracle SQL and PL/SQL Documentation

- *Oracle Database PL/SQL Language Reference*
- *Oracle Database Reference*
- *Oracle Database SQL Language Reference*
- *Oracle Database Concepts*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database SQL Developer User's Guide*
- *Oracle Database 2 Day Developer's Guide*
- *Getting Started with Oracle Cloud*



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Additional Resources

For additional information about the new Oracle SQL and PL/SQL new features, refer to the following:

- Oracle Database: New Features self study
- What's New in PL/SQL in Oracle Database on the Oracle Technology Network (OTN):
  - <http://www.oracle.com/technetwork/database/features/plsql/index.html>
- The online SQL Developer Home page and the SQL Developer tutorial available at:
  - [http://www.oracle.com/technology/products/database/sql\\_developer/index.html](http://www.oracle.com/technology/products/database/sql_developer/index.html)
  - <http://download.oracle.com/oll/tutorials/SQLDeveloper/index.htm>



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## 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



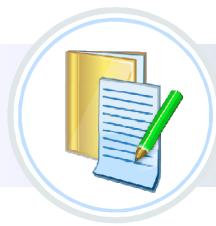
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Practice 1 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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 environment that is available in this course.



2

# Introduction to PL/SQL

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Course Road Map

Lesson 1: Course Overview

## Unit 1: Introducing PL/SQL

Unit 2: Programming with PL/SQL

Unit 3: Working with PL/SQL Code

▶ Lesson 2: PL/SQL Overview

▶ Lesson 3: Declaring PL/SQL Variables

▶ Lesson 4: Writing Executable Statements

▶ Lesson 5: Using SQL Statements in PLSQL Programs

You are here!

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You are in lesson 2, which is part of Unit 1: Introducing PL/SQL.

# 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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
- Understanding PL/SQL blocks
- Generating output messages in PL/SQL



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Limitations of SQL

- Performs one operation at a time on the database
- Lacks the capability of logically grouping multiple database operations



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

**Structured Query Language (SQL)** provides a standard interface to access relational databases. All SQL statements are instructions to relational databases.

A SQL statement can query and manipulate a database, one operation at a time. A single SQL statement can create a table in a database, query data from the database, modify data in the database, or delete data in the database; however each SQL statement can perform any one of these operation only at a time.

Let's look at a scenario in which you transfer funds by using a Bank application. A typical funds transfer will have the following steps to be performed on the database.

1. Deduct funds from the sender's account (an update operation on the corresponding row).
2. Add funds to the receiver's account (an update operation on the receiver's account).

This scenario may also have conditional checks such as:

1. You may want to check whether there are sufficient funds in the sender's account.
2. You may also want to check whether the current transaction will take the balance in the sender's account below minimum balance.

SQL provides for all these conditional checks and updates as independent individual operations. PL/SQL enables you to group such operations which together make a logical unit in the application context.

## Why PL/SQL?

- Enables modularization in the application
- Provides better security
- Enables maintainability
- Provides exception handling



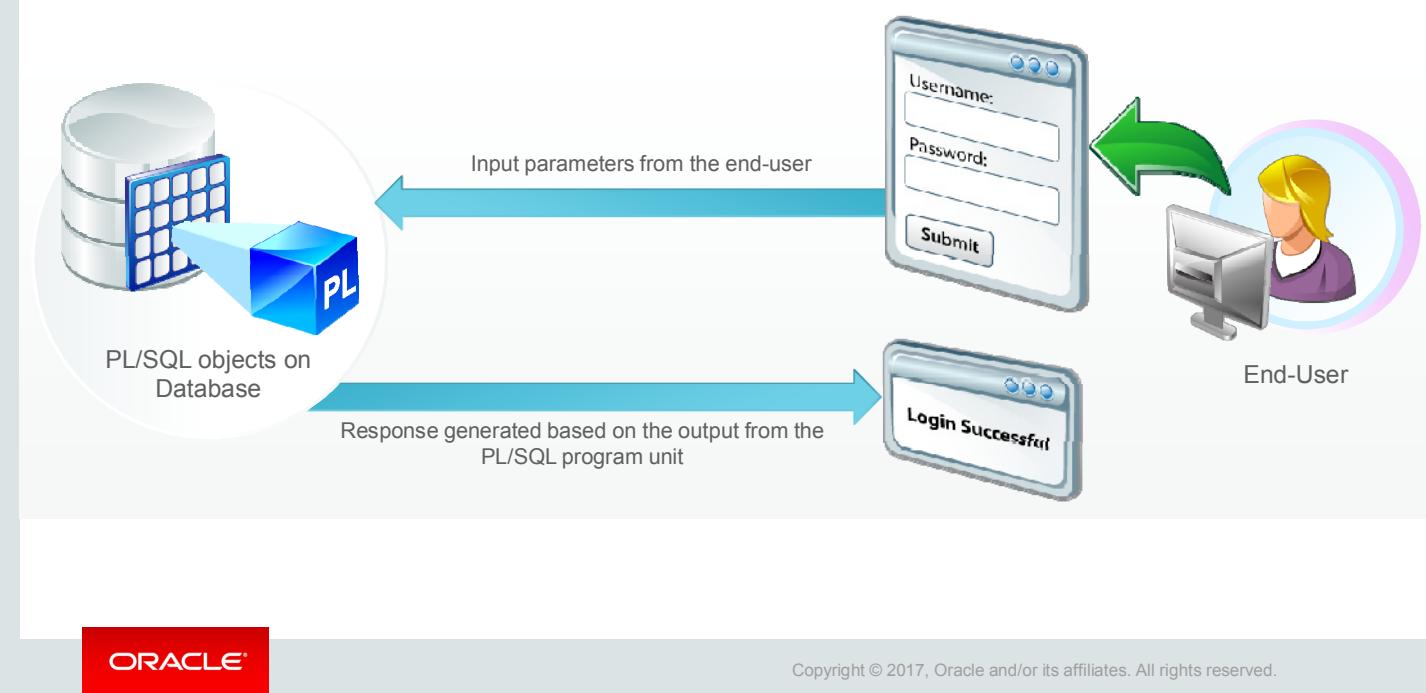
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

PL/SQL provides the following features, which enable efficient application development:

- **Modularization:** Large software systems are developed as modules first, which are later integrated together. The modules communicate through interfaces and work together as a single application.
- **Security:** The implementation of the module is invisible to the end user. That is, the end user knows only the input and output of the module, which is the interface of the module. Security is provided by hiding the implementation details.
- **Maintainability:** When you want to upgrade the performance of a module, you can do it independent of the other modules in the application.
- **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.”

## Why PL/SQL



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Consider a scenario where the HR manager of a company has to make some changes to the database. The HR manager cannot directly access the database. The system will first have to authenticate the HR manager. He/she must provide security credentials such as a username and password. These credentials are then verified against the credentials in the user database to authenticate the HR manager.

The HR application would, therefore, provide a user interface that would accept the required credentials from the end user.

These credentials are provided as parameters to a PL/SQL object on the database.

**Note:** Practical implementation of this scenario would include other intermediary infrastructure components. For simplicity, we have considered only the user interface and the database components.

The PL/SQL program units reside on the database and hide the implementation from end users. A PL/SQL unit that is defined for user authentication would accept credentials as parameters and perform the authentication process on the database.

In this case, the PL/SQL program unit would accept a username and password from the end user, and verify them against those that exist in the database. The authentication process would require execution of multiple SQL statements. A PL/SQL program unit groups these SQL statements into a single program unit, which can be explicitly invoked.

## About PL/SQL



- Stands for “Procedural Language Extension to SQL”
- Integrates procedural constructs with SQL
- Provides a block structure for executable units of code
- Provides procedural constructs such as:
  - Variables, constants, and data types
  - Control structures: Loops, conditional statements
- Enables writing reusable program units



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

**PL/SQL** is a Procedural Language extension to SQL. PL/SQL integrates SQL statements into procedural language structures such as conditional statements and loop constructs.

Consider a scenario where you want to run an operation such as “Giving 10% bonus to all the employees in the company” on the database. If you execute this operation through SQL, you may have to execute an UPDATE statement for each row in the Employees table.

With PL/SQL, you can execute the UPDATE statement (SQL statement) on each row of the Employees table by using a loop construct (procedural construct). Thus, PL/SQL makes things simpler.

Maintaining and debugging code is also made easier with such a structure because you can easily understand the flow and execution of the program unit.

## Benefits of PL/SQL

- Integration of procedural constructs with SQL
- Improved performance
- Modularized program development
- Integration with Oracle tools
- Portability
- Exception handling
- Support for Object Oriented Programming



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

**Integration of procedural constructs with SQL:** Being a declarative language, SQL does not have procedural constructs such as conditional statements and loop statements. With PL/SQL, you can combine the data manipulating capability of SQL with the processing power of procedural languages.

**Improved performance:** You can create a program unit in PL/SQL, which can execute multiple SQL statements. When a SQL statement is executed, there is data exchange between the client and the database server. In the case of a PL/SQL program unit, which contains multiple SQL statements, there is exchange of data only once as a single PL/SQL program unit. This, in turn, improves application performance because of reduced data exchange.

**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:

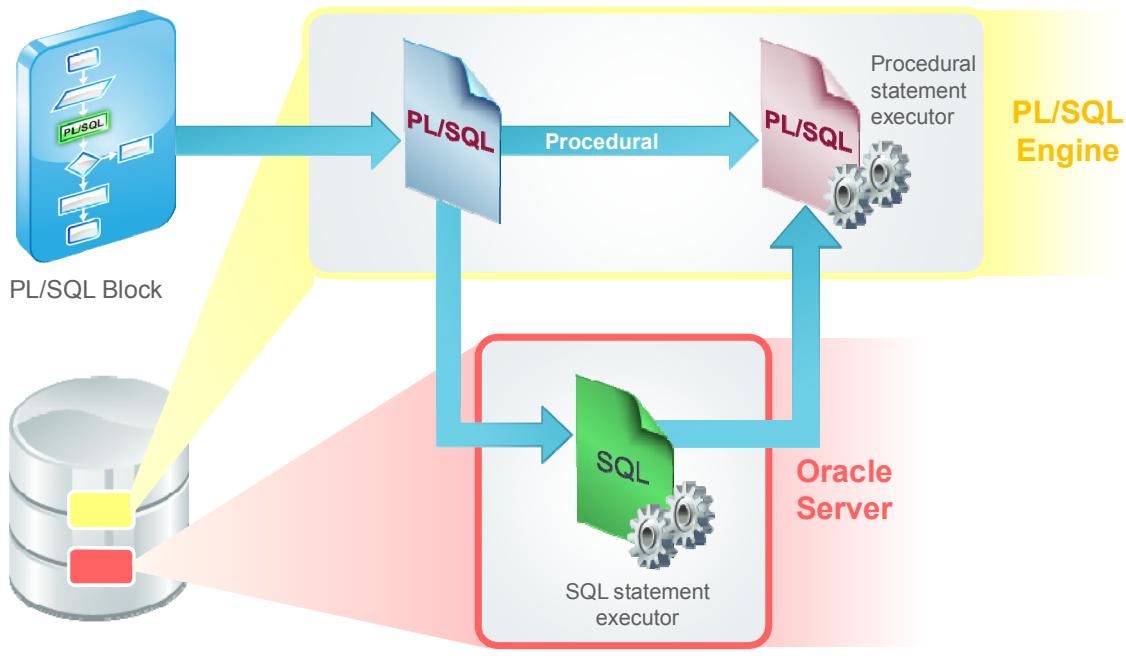
- You can group logically related statements within blocks.
- You can nest blocks inside larger blocks to build powerful programs.
- You can break down your application into smaller modules. If you are designing a complex application, PL/SQL allows you to break down the application into smaller, manageable,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

**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.

**Portability:** PL/SQL programs can run anywhere that an Oracle Server runs, irrespective of the operating system and platform. You do not need to customize them for 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 Runtime Architecture



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The execution of a PL/SQL block has two parts:

1. Execution of the procedural statements
2. Execution of the SQL statements

The procedural logic is executed on a PL/SQL engine and the SQL logic is executed on the Oracle Server.

The PL/SQL engine is a virtual machine that resides in memory and processes the PL/SQL m-code instructions. You can install the PL/SQL engine in the database or in an application development tool such as Oracle Forms. It processes all the procedural statements and passes the SQL statements to the SQL Engine on the Oracle Server.

If the PL/SQL unit does not have any SQL statements, the entire PL/SQL unit is processed by the PL/SQL engine. The SQL engine may invoke a PL/SQL statement executor to execute any function calls that are present in the SQL statements.

## 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)



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

- **DECLARE (optional):** The declarative section begins with the keyword `DECLARE` and ends when the executable section starts. You may or may not have a declaration section, because you may not always declare variables or constants for the PL/SQL block. You can declare variables, constants, cursors, and user-defined exceptions here.
- **BEGIN (required):** The executable section begins with the keyword `BEGIN`. This is the section where you write the procedural or SQL statements that must be executed. You should have at least one statement. The executable section of a PL/SQL block can include any number of PL/SQL blocks.

**Note:** Every `BEGIN` statement should have a corresponding `END` statement to demarcate the PL/SQL block.

- **EXCEPTION (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.

## Agenda

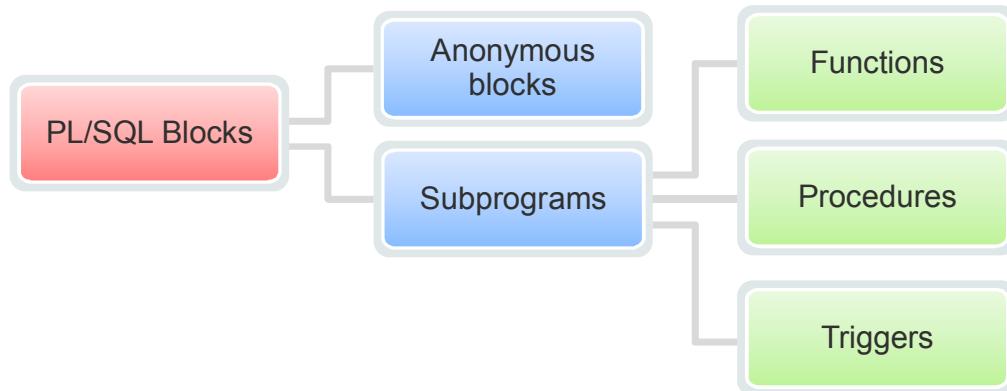
- Understanding the benefits and structure of PL/SQL
- Understanding PL/SQL blocks
- Generating output messages in PL/SQL



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Block Types



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A PL/SQL program comprises one or more blocks.

There are two types of PL/SQL blocks:

- Anonymous blocks
- Named PL/SQL blocks, which are also termed as subprograms

**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. If you want to execute the same block again, you may have to rewrite the block.

**Subprograms:** Subprograms are named PL/SQL blocks that are stored in the database. They are reusable blocks, and can be invoked by the application as per requirement. Subprograms are stored on the database server, and can also be invoked by other subprograms.

**Procedures:** Procedures are named PL/SQL blocks that can be explicitly executed. Procedures help in modularizing application logic.

**Functions:** Functions, like procedures, are named PL/SQL blocks that are explicitly invoked, which are capable of returning a value.

Both procedures and functions can accept some input parameters and process based on these parameters.

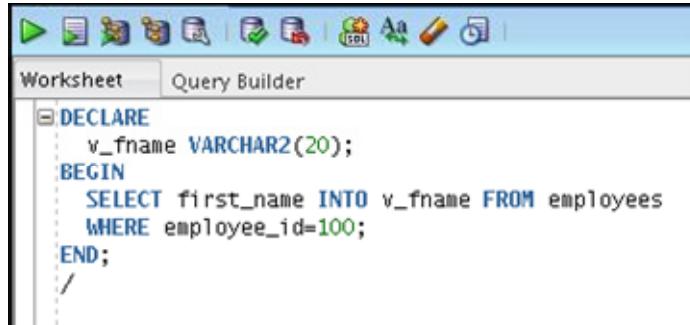
**Triggers:** Triggers are PL/SQL blocks that are conditionally invoked based on an event or operation that occurs in the database context.

For instance, if you want to write all the operations that occur in the database to a log file, you can define a trigger, which would do just that. You can define the trigger such that each time an insert, a delete, or an update is performed on the database, an entry is made to the log.

We discuss Procedures, Functions, and Triggers in the *Oracle Database 12c: Develop PL/SQL Program Units* course.

## Examining an Anonymous Block

An anonymous block in the SQL Developer workspace:



The screenshot shows the Oracle SQL Developer interface with the 'Worksheet' tab selected. The code area contains the following anonymous block:

```
DECLARE
    v_fname VARCHAR2(20);
BEGIN
    SELECT first_name INTO v_fname FROM employees
    WHERE employee_id=100;
END;
```

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

### Example

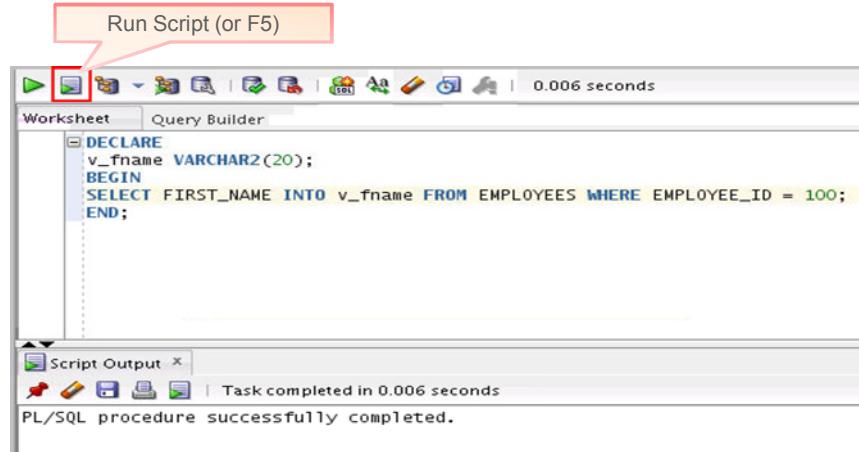
The example block has a declarative section and an executable section. You need not pay attention to the syntax of the statements in the block at this point in time.

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

**Note:** You can also write PL/SQL code through SQL\*Plus, which is a command-line tool.

# Executing an Anonymous Block

Click the Run Script icon to execute the anonymous block:



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

**Note:** The message “PL/SQL procedures successfully completed” is displayed in the Script Output window after the block is executed.

## Agenda

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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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, perform the following steps:

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 must be printed as an argument to this procedure (as shown in the slide). The procedure then outputs the argument.

**Note:** We discuss Packages in the *Oracle Database 12c: Program Units* course. A package is a logical grouping of different PL/SQL program blocks and other database objects. It may have procedures, functions, triggers, sequences, and other database objects grouped according to context.

## Viewing the Output of a PL/SQL Block

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

```
SET SERVEROUTPUT ON;
DECLARE
  v_fname VARCHAR2(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;
```

Script Output x  
Task completed in 0.002 seconds  
PL/SQL procedure successfully completed.  
The First Name of the Employee is Steven

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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
  - a. True
  - b. False



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Answer: b

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:

- 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 © 2017, Oracle and/or its affiliates. All rights reserved.

PL/SQL is a language that has programming features that serve as extensions to SQL. SQL, which is a non-procedural language, is made procedural with the 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 2: Overview

This practice covers the following topics:

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



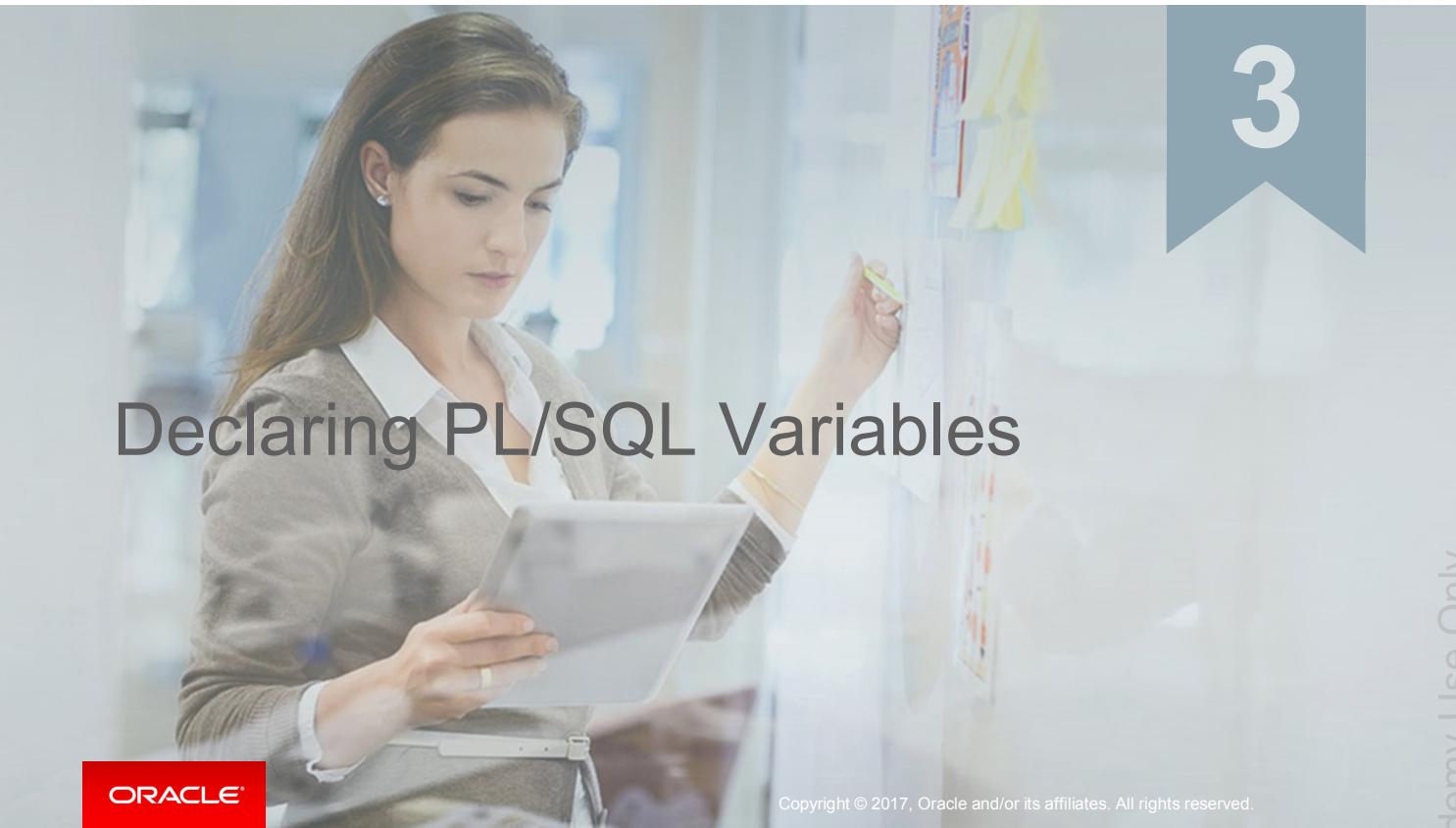
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.





3

# Declaring PL/SQL Variables

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Course Road Map

Lesson 1: Course Overview

**Unit 1: Introducing PL/SQL**

Unit 2: Programming with PL/SQL

Unit 3: Working with PL/SQL Code

ORACLE®

▶ Lesson 2: PL/SQL Overview

▶ **Lesson 3: Declaring PL/SQL Variables**

▶ Lesson 4: Writing Executable Statements

▶ Lesson 5: Using SQL Statements in PLSQL Programs

You are here!

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You are in lesson 3, which is part of Unit 1: Introducing PL/SQL.

## 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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
- Handling variables of different types
- Using the %TYPE attribute and composite data types
- Using bind variables

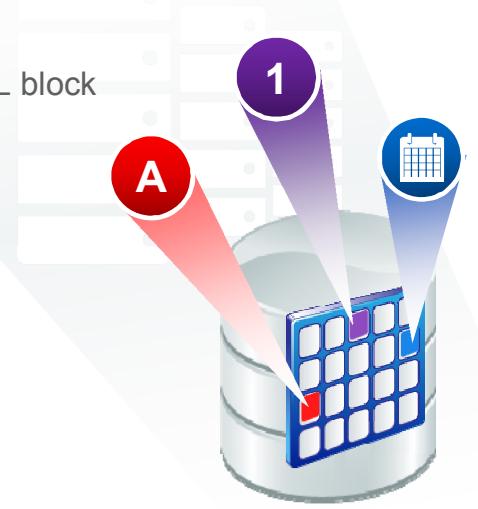


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Variables

- Are labeled storage locations
- Are used to store and process data in a PL/SQL block
- Can hold different types of data
- Should declare variables before using them in the PL/SQL block



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Why do we need variables?

We talk about variables in every programming language. PL/SQL is no exception. Variables are storage locations to store data that is processed in a given PL/SQL block.

These variables serve as placeholders in the machine's memory. You should declare these variables in the `DECLARE` section of the PL/SQL block before using them in the code.

Variables can hold different types of data such as numbers, characters, character strings, date, time, and so on.

## Variables in PL/SQL

The screenshot shows the Oracle SQL Developer interface. At the top, there's a toolbar with various icons. Below it is a menu bar with 'Columns', 'Data', 'Constraints', 'Grants', 'Statistics', 'Triggers', 'Flashback', 'Dependencies', 'Details', 'Partitions', 'Indexes', and 'SQL'. A table named 'employees' is displayed with columns: 'EMPLOYEE\_ID', 'FIRST\_NAME', 'LAST\_NAME', 'EMAIL', 'PHONE\_NUMBER', 'HIRE\_DATE', 'JOB\_ID', and 'SALARY'. Four rows of data are shown. Below the table is a large blue downward-pointing arrow. To the right of the arrow is a code editor window containing a PL/SQL block:

```

DECLARE
    v_sal NUMBER(8,2);
BEGIN
    SELECT salary INTO v_sal
    FROM employees
    WHERE employee_id = 100;
    UPDATE employees
    SET salary = v_sal+100
    WHERE employee_id = 100;
END;

```

Below the code editor is a 'Script Output' window showing the result of the execution:

```

Task completed in 0.002 second
PL/SQL procedure successfully completed.

```

**ORACLE**

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Let us now see the usage of variables in PL/SQL.

Consider a scenario where you are required to update the salary of an employee with employee ID 100. The update will be an increment by \$100.

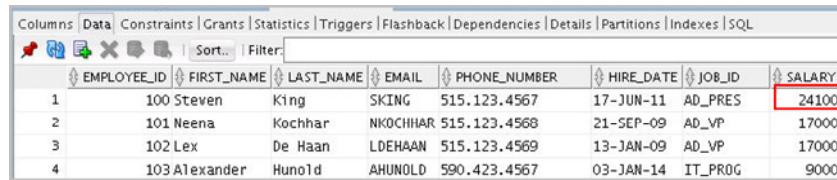
To perform this operation, you should first retrieve the salary of the employee into a variable `emp_sal`.

```
SELECT salary INTO v_sal FROM employees WHERE employee_id = 100;
```

Then you run an `UPDATE` command on it.

```
UPDATE employees SET salary = v_sal+100 WHERE employee_id = 100;
```

## Variables in PL/SQL



EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY
1	100 Steven	King	SKING	515.123.4567	17-JUN-11	AD_PRES	24100
2	101 Neena	Kochhar	NKOCHHAR	515.123.4568	21-SEP-09	AD_VP	17000
3	102 Lex	De Haan	LDEHAAN	515.123.4569	13-JAN-09	AD_VP	17000
4	103 Alexander	Hunold	AHUNOLD	590.423.4567	03-JAN-14	IT_PROG	9000



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide displays data in the employees table after executing the UPDATE operation. The salary of the employee with `employee_id` 100 is updated to \$24,100 from the initial value of \$24,000.

# Requirements for Variable Names

A variable name:

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Each variable is associated with a name, which is also termed an identifier. When you declare a variable, you specify the identifier name and the type of data that the variable will have during the execution of the PL/SQL block. (You learn about the data types that are supported in PL/SQL later in this lesson).

Follow the rules shown in the slide while defining variable names in a PL/SQL block.

Examples of some valid identifiers are as follows:

- X
- t2
- phone#
- credit\_limit
- LastName
- oracle\$number
- money\$\$\$\$
- try\_again\_

# Using Variables in PL/SQL

Variables are:

- Declared and (optionally) initialized in the declaration 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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use variables in the following ways:

- **Declare and initialize them in the declaration section:** You can declare variables in the DECLARE section 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. You must declare a variable before referencing it in other statements, including other declarative statements.
- **Use variables 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_location      VARCHAR2(13) := 'Atlanta';
    v_deptno        NUMBER(2) NOT NULL := 10;
    c_comm          CONSTANT NUMBER := 1400;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the syntax shown in the slide:

<i>identifier</i>	Is the name of the variable
<i>data type</i>	Is a scalar, composite, reference, or LOB data type (This course covers only scalar, composite, and LOB data types.)
CONSTANT	Constrains the variable so that its value cannot change (Constants must be initialized.)
NOT NULL	Constrains the variable so that it contains a value (NOT NULL variables must be initialized.)
<i>expr</i>	Is any PL/SQL expression that can be a literal expression, another variable, or an expression involving operators and functions
DEFAULT	Is the predefined default value that every data type has. You can initialize the variable of a data type to a default value by using DEFAULT during declaration.

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

# Agenda

- Introducing variables
- Handling variables of different types
- Using the %TYPE attribute and composite data types
- Using bind variables



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Declaring and Initializing PL/SQL Variables

```

DECLARE
    v_myName  VARCHAR2(20);

1 BEGIN
    DBMS_OUTPUT.PUT_LINE('My name is: '||v_myName );
    v_myName  := 'John';
    DBMS_OUTPUT.PUT_LINE('My name is: '||v_myName );
END;
/

```

```

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

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Initializing is assigning an initial value to the variable. In the slide, we demonstrate the difference between initializing and declaring.

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 while initializing.
  - Variables are assigned values with the assignment operator `:=`.
  - 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: '`.
  - The output of this anonymous block is:

```

PL/SQL procedure successfully completed.

My Name is:
My Name is: John

```

2. In the second block, the `v_myName` variable is declared and initialized in the `DECLARE` 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:

```

PL/SQL procedure successfully completed.

My Name is: Steven

```

# Initializing Variables Through a SELECT Statement

Retrieve data from the database with a SELECT statement.

Syntax:

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can initialize the variables in a PL/SQL block through a SELECT statement by using the INTO clause. The variables following the INTO clause should be declared in the DECLARE section.

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

**Guidelines for Retrieving Data Through a SELECT Statement:**

- Terminate each SQL statement with a semicolon ( ; ).
- Every value retrieved must be stored in a variable by using the INTO clause.
- When you use the INTO clause, you should fetch only one row. You can use the WHERE clause such that the number of values retrieved must match the number of variables following the INTO clause.
- 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.
- You can use group functions, such as SUM, in a SQL statement, and fetch that value into a variable. Group functions apply to groups of rows in a table and result in a single value.

# Types of Variables

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Each 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. The variable type is defined by the data type it is declared for. There are four categories of data types:

- **Scalar data types:** Scalar data types hold a single value. The value depends on the data type of the variable. For example, VARCHAR2 is a scalar data type. The `v_myName` variable in the example in the section titled “Declaring and Initializing PL/SQL Variables” (in this lesson) is of type VARCHAR2. Therefore, `v_myName` can hold a string value.
- **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.

In this lesson, we look at the Scalar data types.

Non-PL/SQL variables include host language variables or data from the screen fields in an application user interface. You learn more about bind variables later in this lesson.

For more information about LOBs, see the *PL/SQL User’s Guide and Reference*.

# Declaring 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;
  ...

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example variable declarations shown in the slide are defined as follows:

- `v_emp_job`: Variable to store an employee job title; the string is a variable length string with an upper limit of 9 characters
- `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. The numeric value can accommodate decimal values with a precision of two digits.
- `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

# 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.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Some guidelines to follow when you declare PL/SQL variables are as follows:

- 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`, where we understand the semantics of the column name according to the application context.
- 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 data type name. The following declaration names a constant of the 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.

```
c_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.

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

- 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 a 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.

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

## Naming Conventions of the 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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

## Data Types for Strings

- A string is a sequence of symbols that are part of a character set. To declare a string variable, PL/SQL offers:
  - CHAR
  - NCHAR
  - VARCHAR
  - NVARCHAR
  - CLOB
  - NCLOB



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The data types that are prefixed with “N” are “national character set” data types, which means that they are used to store Unicode character data.

- **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 byte. NCHAR is a national character set variant of CHAR.
- **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. However, the database will raise a compiler error if the size of the string is not specified.
- **CLOB**  
Used to store a variable length string that can store data up to 128 terabytes. You can use a CLOB data type if you expect the character string length to be greater than 32,767 bytes.

## 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

PL/SQL procedure successfully completed.  
3rd day in June is :Father's day  
2nd Sunday in May is :Mother's day



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 a string. The slide shows how to use the q' notation to specify delimiters. You can specify any character that is not present in the string as a 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]';
```

## Data Types for Numeric values

- PL/SQL offers different numeric data types to suit different purposes:
  - NUMBER
  - PLS\_INTEGER
  - SIMPLE\_INTEGER
  - BINARY\_INTEGER
  - BINARY\_FLOAT
  - BINARY\_DOUBLE
  - BOOLEAN (TRUE, FALSE or NULL)



BOOLEAN is a non-numeric data type, which can assume one of the three values – TRUE, FALSE or NULL

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

- 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.

- PLS\_INTEGER**

Integer data type, which conforms to the integer representation of your underlying hardware. It is also referred as BINARY\_INTEGER, and can accommodate integers in the range – 2,147,483,647 through 2,147,483,647.

- SIMPLE\_INTEGER**

Numeric data type, which results in shorter execution time for natively compiled code. SIMPLE\_INTEGER and PLS\_INTEGER differ in overflow semantics. PLS\_INTEGER provides better performance according to the underlying hardware architecture. Declare a variable as a SIMPLE\_INTEGER if you know that the value is never going to be NULL and the variable does not need overflow checking.

- BINARY\_INTEGER** is used to store signed integers.

- BINARY\_FLOAT**

Floating-point number in IEEE 754 format. It requires 5 bytes to store a value.

- BINARY\_DOUBLE**

Floating-point number in IEEE 754 format. It requires 9 bytes to store a value.

**BINARY** data types are useful to improve the performance of computation-intensive operations.

- BOOLEAN**

Base type that stores one of the three possible values used for logical calculations: TRUE, FALSE, or NULL

## Data Types for Date and Time values

- The Oracle database provides three data types to work with dates and times:
  - DATE
  - TIMESTAMP WITH TIME ZONE
  - TIMESTAMP WITH LOCAL TIME ZONE
  - INTERVAL YEAR TO MONTH
  - INTERVAL DAY TO SECOND



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

- DATE:** This is the 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. 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.

- **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 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 does not store time zone information internally, but you can see the local time zone information in the SQL output. In the case of `TIMESTAMP WITH TIME ZONE`, the time zone information is stored locally.
- **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.

### Choosing the right data type

Following are some guidelines that will enable you to choose the right data type for a date variable:

- Use `TIMESTAMP` only when you have to keep track of values down to fractions of seconds.
- Use `TIMESTAMP WITH TIMEZONE` when you have to keep track of the session time zone when the data was entered.
- Use `TIMESTAMP WITH LOCAL TIMEZONE` when you have to convert the value between the database time and the session time very often.
- Use the `DATE` type when your current application has to be compatible with an older application when the `TIMESTAMP` data type was not available in databases.

# Data Type Conversion

- Converts data to comparable data types
- Is of two types:
  - Implicit conversion
  - Explicit conversion
- Includes functions:
  - TO\_CHAR
  - TO\_DATE
  - TO\_NUMBER
  - TO\_TIMESTAMP



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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;
    DBMS_OUTPUT.PUT_LINE(v_total_salary);
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

**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.

# 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');
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

1. Because the string literal that is being assigned to `date_of_joining` is in certain format, this example performs implicit conversion, and assigns the specified date to `date_of_joining`.
2. 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

- Introducing variables
- Handling variables of different types
- Using the %TYPE attribute and composite data types
- Using bind variables



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## The %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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

PL/SQL variables are usually declared to hold and manipulate the 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 names. If you refer to a previously declared variable, prefix the variable name of the previously declared variable to the variable that is being declared.

### 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.

## 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;
...
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 `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 by using `%TYPE`. Therefore, if you declare a variable by using the `%TYPE` attribute that uses a database column that is defined as NOT NULL, you can assign a NULL value to the variable.

## Declaring Boolean Variables

- Only 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.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

With PL/SQL, you can compare variables in both SQL and procedural statements. These comparisons, called Boolean expressions, consist of simple or complex expressions that are 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, an inapplicable, or an unknown value.

### Examples

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

The following expression yields TRUE:

```
emp_sal1 < emp_sal2
```

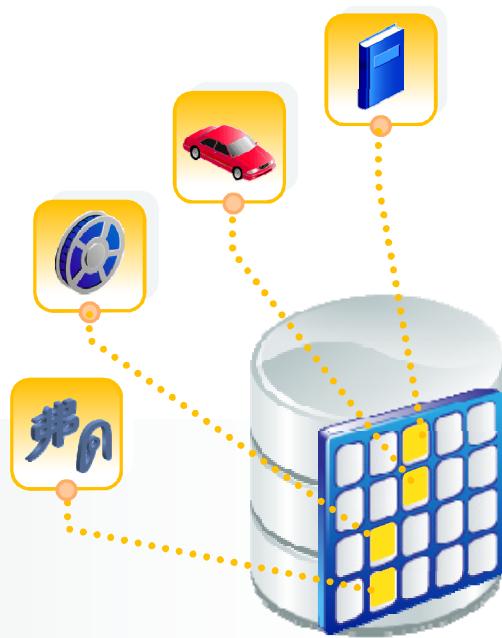
Declare and initialize a Boolean variable:

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

You can initialize a Boolean variable to one value among TRUE, FALSE, and NULL.

## LOB Data Type Variables

- Large objects (LOBs) are meant to store a large amount of data.
- LOB data types:
  - Character large object (CLOB)
  - Binary large object (BLOB)
  - Binary file (BFILE)
  - National language character large object (NCLOB)
- LOB data types enable you to store blocks of unstructured data up to 4 gigabytes in size.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

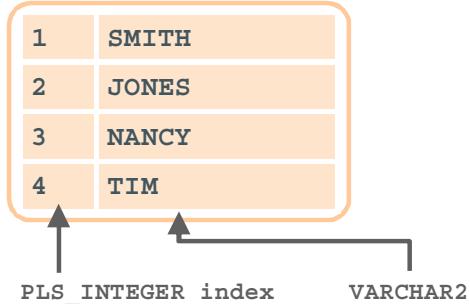
- The character large object (CLOB) data type is used to store large blocks of character data in the database.
- The binary large object (BLOB) data type is used to store large unstructured or structured binary objects in the database. When you insert such data into or retrieve such data from the database, the database does not interpret the data. The 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:



PL/SQL Collections:



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.



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 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 *Advanced PL/SQL* course.

# Agenda

- Introducing variables
- Handling variables of different types
- Using variable data types and the %TYPE attribute
- Using bind variables



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Bind Variables

- Bind variables are:
  - Created in the host environment
  - Also called *host* variables
  - Created with the VARIABLE keyword in PL/SQL
  - 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 by using the PRINT command.
- Bind variables are required when using SQL\*Plus and SQL Developer.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Bind variables are variables that you create in a host environment; therefore, they are also known as host variables.

## Uses of Bind Variables

Bind variables are created in the application environment and not in the DECLARE section of a PL/SQL block. Therefore, bind variables are accessible even after the block is executed and, can be accessed by other PL/SQL blocks. These variables can be passed as runtime 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 by using SQL Developer and view its value by using the PRINT command.

## Bind Variables: Examples

```
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
```

B_RESULT
-----
30000

Script Output X		
PL/SQL procedure successfully completed.		
B_EMP_SALARY	FIRST_NAME	LAST_NAME
7000	Oliver	Tuvault
	Sarah	Sewall
	Kimberely	Grant

```
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;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

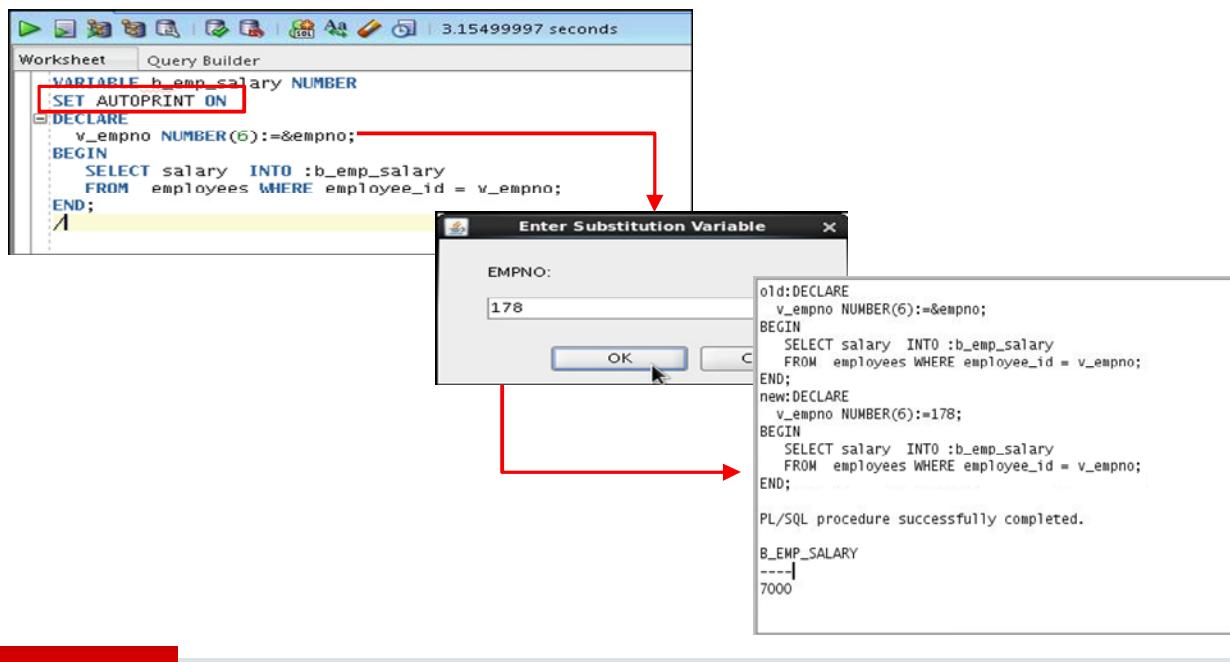
The first PL/SQL block in the slide creates and uses the bind variable `b_result`. The output resulting from the `PRINT` command is shown below the code in the slide.

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 second example in the slide, `b_emp_salary` is created as a bind variable. It is then used in the `SELECT` statement.

In an application, you can use bind variables to hold the values received from the application user interface. These variables can later be processed by the PL/SQL blocks on the database according to the application's requirement.

## Using AUTOPRINT with Bind Variables



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

### Example

In the code example in the slide:

- 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.

# Quiz



The %TYPE attribute:

- a. Is used to declare a variable according to a database column definition
- b. Is used to declare a variable according to a collection of columns in a database table or view
- c. Is used to declare a variable according to the definition of another declared variable
- d. Is prefixed with the database table and column names or the name of the declared variable



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

**Answer: a, c, d**

## The %TYPE Attribute

PL/SQL variables are usually declared to hold and manipulate the 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 names. If you refer to a previously declared variable, prefix the variable name of the previously-declared variable to the variable that is 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.

## 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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, which 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 also 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 3: 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 © 2017, Oracle and/or its affiliates. All rights reserved.

Exercises 1, 2, and 3 are paper based.

4

# Writing Executable Statements

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Course Road Map

Lesson 1: Course Overview

**Unit 1: Introducing PL/SQL**

Unit 2: Programming with PL/SQL

Unit 3: Working with PL/SQL Code

ORACLE®

▶ Lesson 2: PL/SQL Overview

▶ Lesson 3: Declaring PL/SQL Variables

▶ **Lesson 4: Writing Executable Statements**

▶ Lesson 5: Using SQL Statements in PLSQL Programs

You are here!

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Objectives

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

- Identify the 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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 © 2017, Oracle and/or its affiliates. All rights reserved.

# 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: --, /\* \*/



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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*.

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

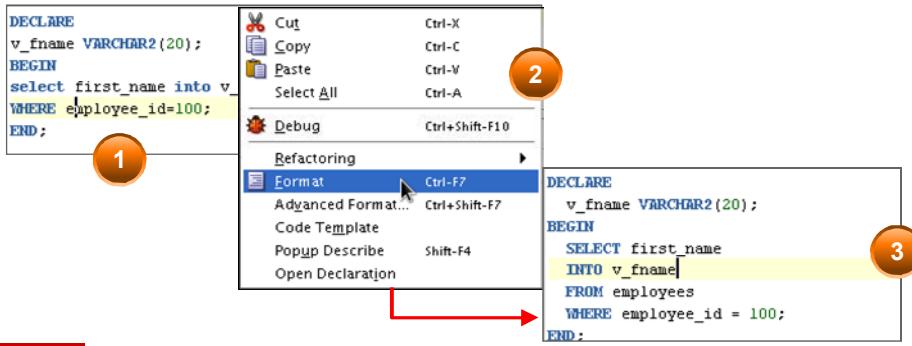
**Note:** This is only a subset and not a complete list of delimiters.

- **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 a 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 a 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.

# 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.
- Formatting Code: Statements can span several lines.

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



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## 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;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

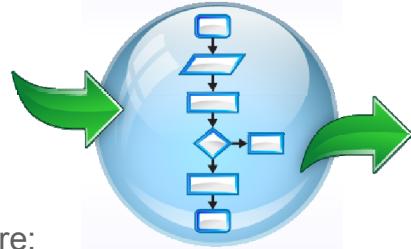
- A single-line comment is commonly prefixed with two hyphens (--)
- You can also enclose a comment between the symbols /\* and \*/

**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

- Predefined functions that are used in SQL can also be used in PL/SQL.
- Functions that are available in procedural statements are:
  - Single-row functions
  - Built-in functions with Strings
  - Built-in functions with Numbers
  - Built-in functions with Dates
- Functions that are not available in procedural statements are:
  - DECODE
  - Group functions



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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 form 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);
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 blocks



NEW HIRE



MANAGER



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Consider a scenario where a manager needs to assign an `employee_id` to each new employee joining his or her department. When you want to define an employee ID, you may have the following requirements:

1. You do not want the `employee_id` to be 0.
2. No two employees in the organization should have the same employee ID.
3. You cannot remember the employee ID assigned to the employee who last joined the organization so that you can increment it by 1 and assign it to the new employee.
4. Sometimes, you may not want the employee ID to start with a single digit, depending on the context requirement.

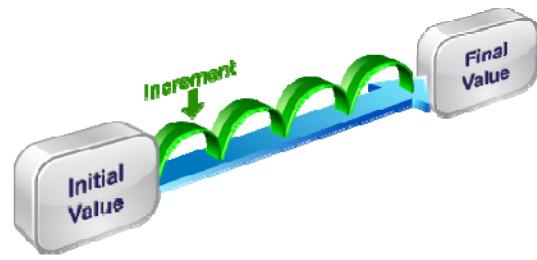
In such cases, using a sequence object that can be shared by the managers of multiple departments is the solution.

A sequence is an incrementing numeric value. Developers can define its initial value, the increment with which the next value can be generated, and the maximum value or the upper limit of the sequence. This object on the database can be shared by multiple users who can eventually access the next value in the sequence.

## Using Sequences in PL/SQL blocks

- Sequences are database objects that can be used by multiple users to generate sequential numbers.
- Sequences can be created through the `CREATE SEQUENCE` statement.

```
CREATE SEQUENCE emp_sequence
INCREMENT BY 1
START WITH 1
NOMAXVALUE;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

### Sequences

Sequences are database objects like views, constraints, and so on. You can use sequences to generate a sequence of numbers. You can define the increment size, starting value, and maximum value while creating a sequence. The numeric value can have 28 or fewer digits.

A single sequence object can be used by multiple users who use the database.

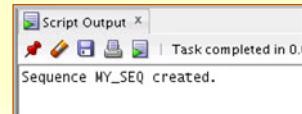
The following table shows the properties associated with a sequence that can be used to define its behavior:

INCREMENT BY	You can specify the value of each increment in the sequence. In the given example, you see that the sequence is incremented by 1. You can change it to any other value.
START WITH	You define the starting value of the sequence.
MAXVALUE	You define the upper limit of the sequence through MAXVALUE.

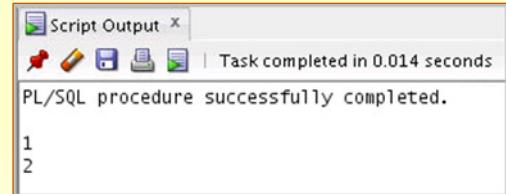
NOMAXVALUE	When you do not want to define an upper limit, you use NOMAXVALUE during sequence creation.
MINVALUE	You can specify the minimum value of a sequence.
NOMINVALUE	NOMINVALUE in CREATE SEQUENCE sets the minimum value of a sequence to 1.
CYCLE	This indicates that the sequence will restart from the minimum value after it reaches the maximum value.
NOCYCLE	This indicates that the sequence will not restart from the minimum after it reaches the maximum value.

## Using Sequences in PL/SQL Blocks

```
CREATE SEQUENCE my_seq  
INCREMENT BY 1  
START WITH 1  
NOMAXVALUE;
```



```
DECLARE  
    v_new_id NUMBER;  
BEGIN  
    v_new_id := my_seq.NEXTVAL;  
    DBMS_OUTPUT.PUT_LINE(v_new_id);  
    DBMS_OUTPUT.PUT_LINE(my_seq.NEXTVAL);  
END;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The usage of sequences in PL/SQL expressions is similar to that in SQL statements.

You can create a sequence once and use it multiple times in an application. You create a sequence by executing the **CREATE SEQUENCE** statement as shown in the slide.

After creating the sequence object, you can access the value of the sequence through the **NEXTVAL** and **CURRVAL** pseudocolumns.

In the PL/SQL block shown in the slide, we use **NEXTVAL** with the **my\_seq** sequence twice. First, we assign it to the variable **v\_new\_id** and output the value. Next, we use it within the **DBMS\_OUTPUT.PUT\_LINE** function call. Observe that you see a new incremented value the second time.

## Agenda

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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Nested blocks

PL/SQL blocks can be nested.

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



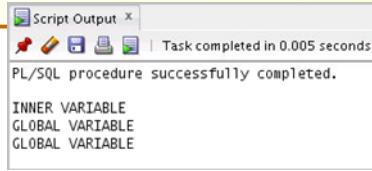
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 functionality 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) := 'INNER 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;
```



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

The `v_outer_variable` 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. When there is no variable with the same name in the inner block, the variable in the outer block is accessed. Therefore, the `v_outer_variable` variable is considered to be the global variable for all the enclosed blocks.

The `v_inner_variable` 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 up in the `DECLARE` 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;
/

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

```

PL/SQL procedure successfully 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.

## Scope

- 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 that is 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 the identifiers are distinct; changes in one do not affect the other.

## 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;

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 in the slide:

- 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:

```

PL/SQL procedure successfully 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 the 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;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 the 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**

## Agenda

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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Operators in PL/SQL

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

Same as in SQL

- Exponential operator (\*\*)



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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);
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

# Programming Guidelines

Make code maintenance easier by:

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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

- a. True
- b. False



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Answer: a

### 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 form only a subset of the complete list.

## Summary

In this lesson, you should have learned how to:

- Identify the 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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 sub blocks.
- 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 4: Overview

This practice covers the following topics:

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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Exercises 1 and 2 are paper based.



5

# Using SQL Statements Within a PL/SQL Block

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Course Road Map

Lesson 1: Course Overview

## Unit 1: Introducing PL/SQL

Unit 2: Programming with PL/SQL

Unit 3: Working with PL/SQL Code

▶ Lesson 2: PL/SQL Overview

▶ Lesson 3: Declaring PL/SQL Variables

▶ Lesson 4: Writing Executable Statements

▶ **Lesson 5: Using SQL Statements in PLSQL Programs**

You are here!

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You are in lesson 5, which is part of Unit 1: Introducing PL/SQL.

## 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
- Make use of the `INTO` clause to hold the values returned by a SQL statement
- Manipulate data with DML statements in PL/SQL
- Use transaction control statements in PL/SQL
- Differentiate between implicit cursors and explicit cursors
- Use SQL cursor attributes



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 manipulation language (DML) and transaction control statements in PL/SQL. You understand the need for cursors and differentiate between the two types of cursors. This lesson also presents the various SQL cursor attributes that can be used with implicit cursors.

# Agenda

- SQL statements in PL/SQL blocks
- Manipulating data with PL/SQL
- Introducing SQL cursors

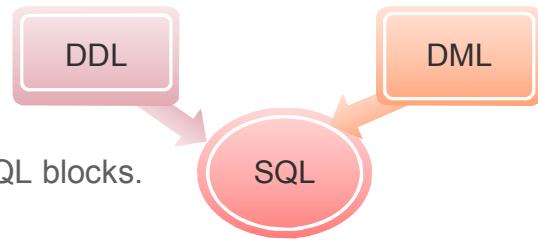


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## SQL Statements in PL/SQL

- SQL consists of three sub-languages:
  - DDL
  - DML
- DDL statements are generally not included in PL/SQL blocks.
- DML statements are included in PL/SQL blocks.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

SQL has three sub-languages:

- Data Definition Language (DDL)
- Data Manipulation Language (DML)
- Data Control Language (DCL)

DDL is used to define the database schema, define the structure of tables, and modify the structure of the tables.

DML is used to add data to, retrieve data from, and modify data in tables based on the structure of the database defined by the DDL statements.

DDL statements are executed once for every database to define the structure. After the structure is defined, operations are performed through DML statements.

DCL statements COMMIT and ROLLBACK are used to control transactions on a database and might be included in PL/SQL blocks. GRANT and REVOKE statements are used by the administrators and are generally not included in the PL/SQL block

When you define PL/SQL blocks, you generally write reusable code in those blocks. PL/SQL does not support DDL statements directly. You do not need reusable code for creating the database structure because it is created only once during the database life cycle.

DDL and DCL statements are supported through dynamic SQL.

Dynamic SQL statements are built as character strings at run time, and can contain placeholders for parameters. The details of working with dynamic SQL are covered in the *Oracle Database 12c: Develop PL/SQL Program Units* course.

## SQL Statements in PL/SQL

- SELECT command to retrieve data into the PL/SQL block.
- DML commands to make changes to the database.
- COMMIT, ROLLBACK, or SAVEPOINT commands for transaction control.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In a PL/SQL block, you use SQL statements to retrieve data from and modify data in a database table. PL/SQL supports data manipulation language (DML) and transaction control commands. 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 are executed through dynamic SQL statements.
- PL/SQL does not directly support data control language (DCL) statements such as GRANT or REVOKE. You can use dynamic SQL to execute them. The DCL statements that can be executed in PL/SQL are COMMIT, ROLLBACK, and SAVEPOINT.

## 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;
/
```

PL/SQL procedure successfully completed.  
First name is :Jennifer



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

### The `INTO` Clause

You use the `INTO` clause to retrieve data from a database through a `SELECT` statement into the variables in the PL/SQL block. The `INTO` clause is mandatory, and occurs between the `SELECT` and `FROM` clauses. You must specify one variable for each item retrieved from the database. The order and type of the variables must correspond with the items selected.

You generally use the `INTO` clause to populate either PL/SQL variables or host variables.

### Queries Must Return Only One Row

The `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 should precede the block. It is sufficient to execute `SET SERVEROUTPUT ON` once for every session.

## 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 from the result set into the variables in the PL/SQL block. If you have to retrieve multiple rows and operate on the data, you have to make use of explicit cursors. Cursors hold the result set of a `SELECT` statement and allow the developer to operate on each row of the result set by iterating through it in a loop.

You are introduced to cursors later in this lesson and learn about explicit cursors in the lesson titled “Using Explicit Cursors.”

## 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;
/
```

```
PL/SQL procedure successfully completed.
Hire date is :17-JUN-11
Salary is :24000
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the code in slide, the `v_emp_hiredate` and `v_emp_salary` variables are declared in the `DECLARE` section of the PL/SQL block. In the executable section, the values of the `hire_date` and `salary` columns for the employee with `employee_id` 100 are retrieved from the `employees` table and stored in the `v_emp_hiredate` and `v_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.

The PL/SQL engine will retrieve the data in the order specified in the `SELECT` statement and place them in the variables as they are retrieved. The `hire_date` value will be placed in `v_emp_hiredate` and the `salary` value is placed in `v_emp_salary`. If you reverse the variable names, the engine will try to place `hire_date` in `v_emp_salary`, realize that they are of incompatible types, and give 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;
```

```
PL/SQL procedure successfully completed.
The sum of salary is :28800
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 by 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 by 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

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 runtime 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;
    .

```

## Avoiding Naming Ambiguities

- Use a naming convention to avoid ambiguity in the WHERE clause.
- Avoid using database column names as identifiers.
- 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.
- The names of variables take precedence over the function names.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 when you have identifier names that are the same as database column names because PL/SQL checks the database first for a column in the table.
- Keep a watch on the following precedence rules while defining local variables and formal parameters in the PL/SQL blocks.
  - Local variables and formal parameters have higher precedence than database table names.
  - Database table column names have higher precedence than the local variables in a PL/SQL block.
  - Variable names have higher precedence than function names; therefore, keep a watch if you are giving the same identifier names to local variables and function blocks.

# Agenda

- SQL statements in PL/SQL blocks
- Manipulating data with PL/SQL
- Introducing SQL cursors



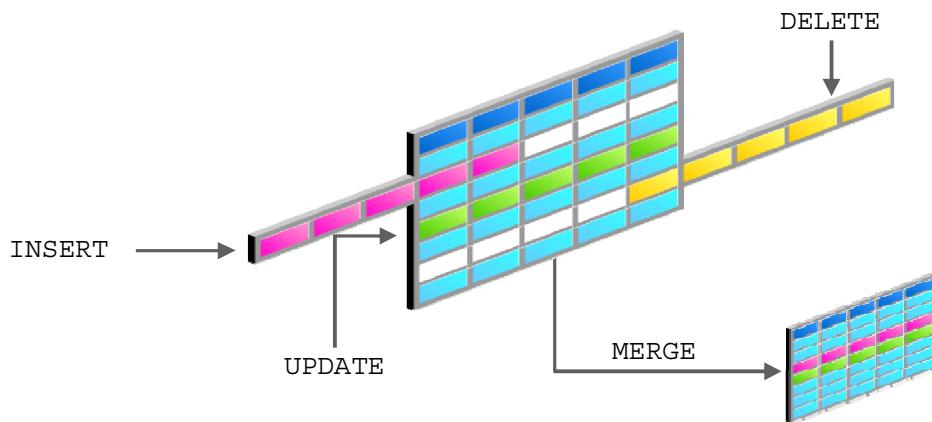
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Using PL/SQL to Manipulate Data

Make changes to database tables by using DML commands:

- INSERT
- UPDATE
- DELETE
- MERGE



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You manipulate data in a 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.

## Insert 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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. Therefore, the command `rollback` is used as shown in the code for slide 15\_sa in `code_ex_05.sql`.

## Update Data: Example

Increase the salary of all employees who are stock clerks.

```
SELECT first_name, salary
FROM employees
WHERE job_id = 'ST_CLERK';
```

Script Output		Query Result	
		SQL   All Rows Fetched: 20 in 0.001 seconds	
		FIRST_NAME	SALARY
1	Julia	3200	
2	Irene	2700	
3	James	2400	
4	Steven	2200	
5	Laura	3300	

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

PL/SQL procedure successfully completed.

```
SELECT first_name, salary
FROM employees
WHERE job_id = 'ST_CLERK';
```

Script Output		Query Result	
		SQL   All Rows Fetched: 20 in 0.003 seconds	
		FIRST_NAME	SALARY
1	Juli	4000	
2	Irene	3500	
3	James	3200	
4	Steven	3000	
5	Laura	4100	



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Let us first see the current state of the table before we run the update in the PL/SQL block.

For simplicity, we will see only 5 rows of the result set. You can see the actual result when you execute the code in your work station.

The SELECT statement retrieves the `first_name` and `salary` of employees whose `job_id` is `ST_CLERK`.

Execute the PL/SQL block with the UPDATE statement. If the block executes without errors, you see the response `PL/SQL Procedure successfully completed` on the Script Output tab.

Execute the same SELECT statement again to see the update in the Employees table. You see the new values in the Query result as a result of the UPDATE operation.

**Note:** Remember to run a rollback after executing the operation to maintain data consistency for further demonstrations.

## Delete Data: Example

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

```
DECLARE
    v_emp      employees.employee_id%TYPE := 176;
BEGIN
    DELETE FROM employees
    WHERE employee_id = v_emp;
END;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

The code shown in the slide deletes the row from employees table whose employee\_id = 176

**Note :** A DELETE operation will not complete if it is violating a constraint.

## 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;
/

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 in the slide matches the `emp_no` 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.

Before executing the `MERGE` statement, you should have the `copy_emp` table created. You can create the `copy_emp` table with the following `CREATE` statement:

```

CREATE TABLE copy_emp (emp_no NUMBER(6,0) PRIMARY KEY,
first_name VARCHAR2(20), last_name VARCHAR2(25), email
VARCHAR2(25), phone_number VARCHAR2(20), hire_date DATE,
job_id VARCHAR2(10), salary NUMBER(8,2), commission_pct
NUMBER(2,2), manager_id NUMBER(6,0), department_id
NUMBER(4,0));

```

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

```
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;  
/
```

# Agenda

- SQL statements in PL/SQL blocks
- Manipulating data with PL/SQL
- Introducing SQL cursors

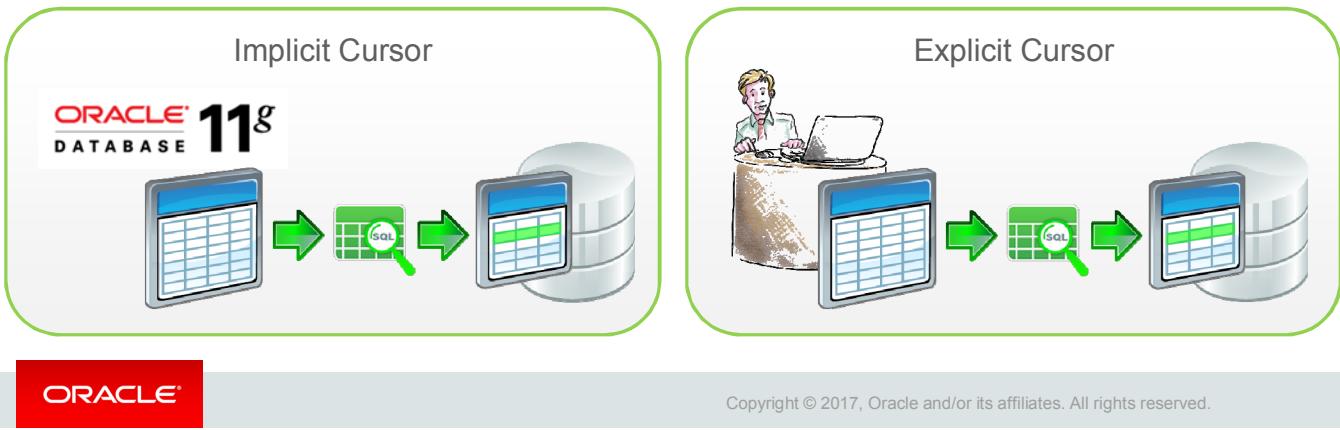


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## SQL Cursor

- A cursor is a pointer to the private memory area that stores information about processing a specific SELECT or DML statement.
- There are two types of cursors:
  - **Implicit:** Created and managed by PL/SQL
  - **Explicit:** Created and managed explicitly by the programmer



### 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 that is required for processing and the information that is 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. The result set of the SQL statement is stored in the context area for further processing.

## 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.
- **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. A cursor that is declared by programmers is called an *explicit cursor*. You declare such a cursor in the `DECLARE` section of a PL/SQL block.

PL/SQL blocks execute on the database server. Cursors declared in the PL/SQL block point to the memory area on the server while executing. When a client initiates a PL/SQL block , a corresponding cursor is created for the `SELECT` statements in the PL/SQL block and the result set is retrieved into the memory area pointed by the cursor.

## SQL Cursor Attributes for Implicit Cursors

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

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

SQL cursor attributes enable you to evaluate what happened when an implicit cursor was last used. You can 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 execution of the appropriate DML command. 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.

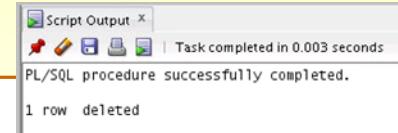
Observe that the attributes are prefixed with `SQL`. These cursor attributes are used with implicit cursors that are automatically created by PL/SQL and for which you do not know the names. Therefore, you use `SQL` instead of the cursor name.

## 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 := 165;
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 © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide deletes a row with `employee_id` 165 from the `employees` table. Using the `SQL%ROWCOUNT` attribute, you can print the number of rows deleted.

`SQL%ROWCOUNT` will return the number of rows in the table that satisfy the given condition. In this case, `SQL%ROWCOUNT` will return 1 because there is only one row in the table whose `employee_id` value is 165.

# Quiz



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

- a. True
- b. False



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Answer: b

### The `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` statement. 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

The `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.

## 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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 5: 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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.





6

# Writing Control Structures

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Course Road Map

Lesson 1: Course Overview

Unit 1: Introducing PL/SQL

**Unit 2: Programming with PL/SQL**

Unit 3: Working with PL/SQL Code

▶ Lesson 6: Writing Control Structures

You are here!

▶ Lesson 7: Working with Composite Data Types

▶ Lesson 8: Using Explicit Cursors

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You are in lesson 6, which is part of Unit 2: Programming with PL/SQL.

## 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®

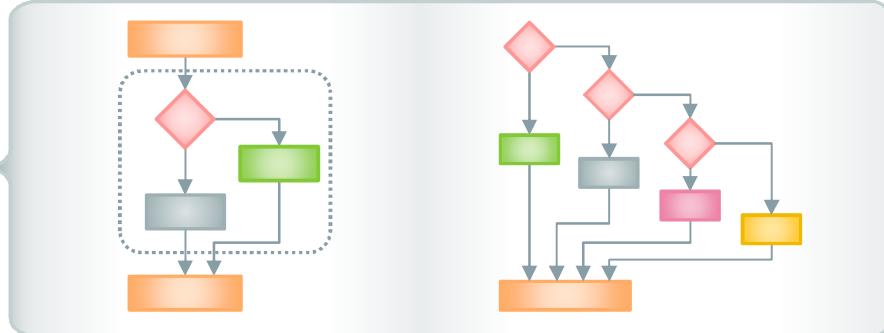
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

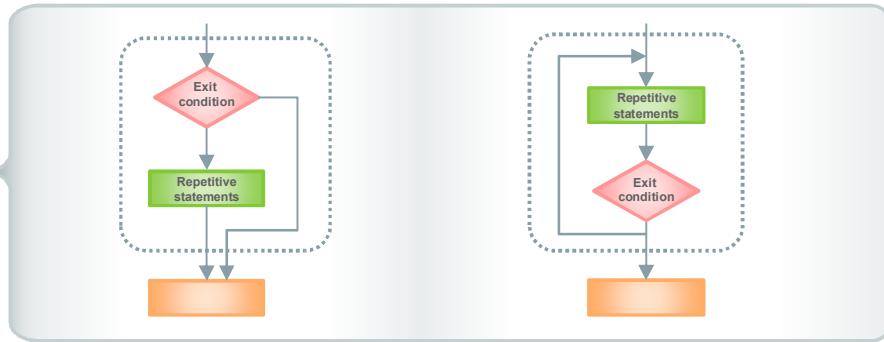
In this lesson, you learn how to use control structures such as `IF` statements, `CASE` expressions, and `LOOP` structures in a PL/SQL block.

## PL/SQL Control Structures

Conditional statements



Loop statements



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can change the logical flow of statements within a 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` statement, `LOOP` control structures.

# Agenda

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



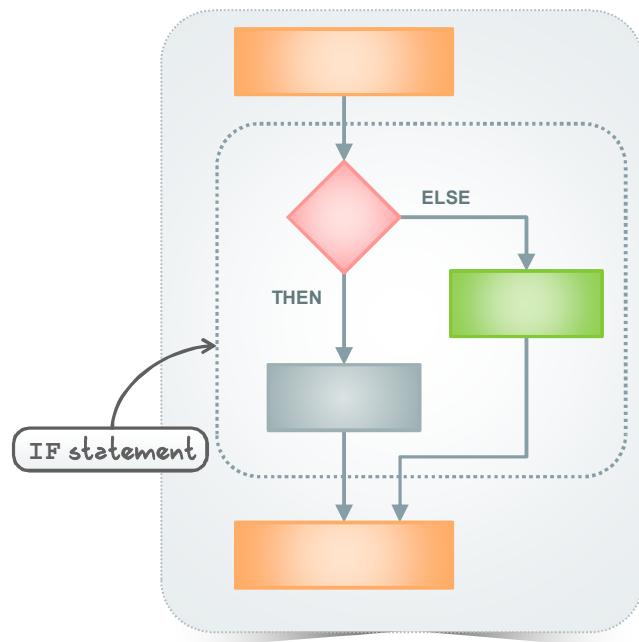
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# IF Statement

Syntax:

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

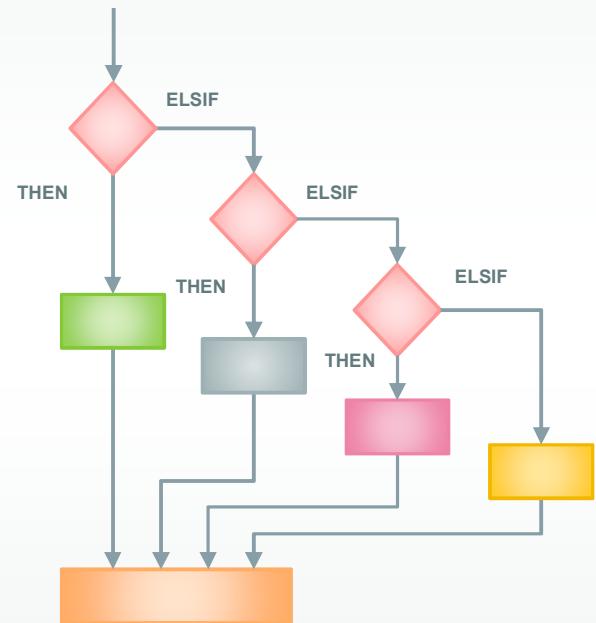
The structure of a PL/SQL IF statement is similar to the structure of IF statements in other procedural languages. It allows PL/SQL to perform actions selectively based on conditions.

In the syntax:

<i>condition</i>	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
<i>statements</i>	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-ELSIF Statements

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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 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 pre-empted 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 is allowed in your IF statement. END IF marks the end of an IF statement and must be terminated by a semicolon.

## Simple IF Statement

```
DECLARE
    v_myage  NUMBER := 10;
BEGIN
    IF v_myage < 11
    THEN
        DBMS_OUTPUT.PUT_LINE(' I am a child ');
    END IF;
END;
/
```

```
PL/SQL procedure successfully completed.
I am a child
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the slide, you see an example of a simple IF statement with the THEN clause.

- The v\_myage variable is initialized to 10.
- The condition for the IF statement returns TRUE because v\_myage is less than 11.
- Therefore, the control reaches the THEN clause.

### Adding Conditional Expressions

You can relate multiple conditional expressions in an IF statement through 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.

You can have any number of conditional expressions for a single IF statement. 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;
/
```

```
PL/SQL procedure successfully completed.
I am not a child
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

An `ELSE` clause is added to the code from the previous slide. The condition has not changed, but the value of the variable has changed from 10 to 31. The condition now evaluates to `FALSE`. Recall that 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;
/
```

```
PL/SQL procedure successfully completed.
I am in my thirties
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can have multiple ELSIF clauses in an IF statement and a final ELSE clause. The example in the slide illustrates the following characteristics of these clauses:

- The ELSIF clauses can have conditions, unlike the ELSE clause. You can check additional conditions through the ELSIF clauses if the condition in the IF clause evaluates to FALSE or NULL.
- 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 an 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;
/
```

```
PL/SQL procedure successfully completed.
I am not a child
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the example 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



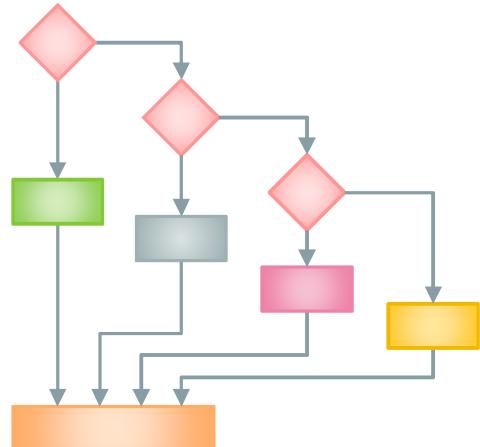
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## 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 © 2017, Oracle and/or its affiliates. All rights reserved.

You can use a CASE expression to return a result from many alternatives based on the value of a selector. The selector is followed by one or more WHEN clauses that are checked sequentially. Whenever the value of the selector equals the value of a WHEN expression, the statements in the WHEN clause are executed and that result is returned.

PL/SQL also provides a searched CASE expression, which has the following 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.

## 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;
/
```



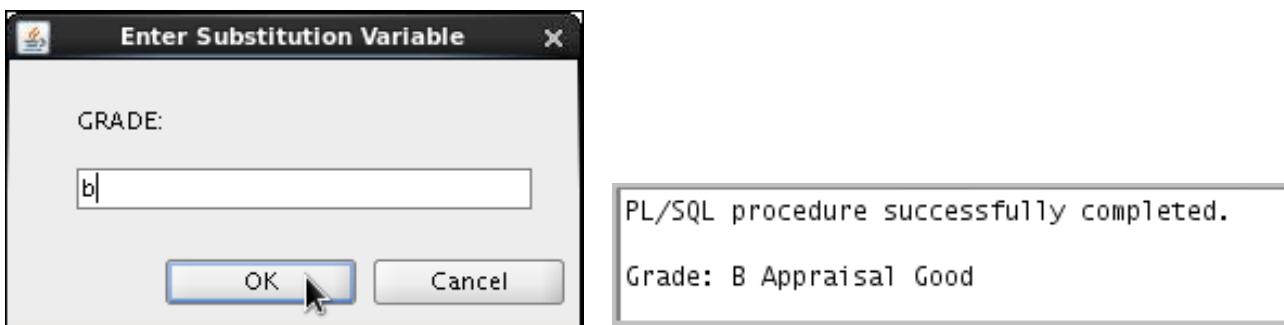
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the previous example, you saw a single test expression, the `v_grade` variable. The `WHEN` clause compares 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`:



## Searched CASE Expressions

```
SET SERVEROUTPUT ON

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;
/
```



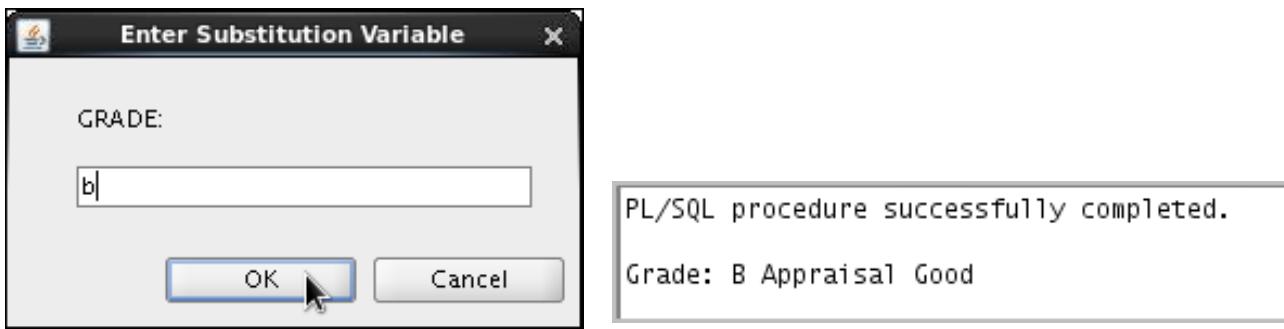
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the previous example, you saw a single test expression, the `v_grade` variable. The `WHEN` clause compares 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;   ← Uses END CASE instead of END
    DBMS_OUTPUT.PUT_LINE ('You are working in the '|| v_deptname |||
    ' department. There are '||v_emps ||' employees in this
    department');
END;
/

```

If 108 matches with the value of v\_mngid, an action is performed instead of just returning a value.

The action defined can modify the database also.

**PL/SQL procedure successfully completed.**  
You are working in the Finance department. There are 6 employees in this department



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 a condition and returns a value, whereas a `CASE` statement evaluates a 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;`

**Note:** Whereas an `IF` statement is able to do nothing (the conditions could all be false and the `ELSE` clause is not mandatory), a `CASE` statement must execute some PL/SQL statement.

## Handling Nulls

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

- Simple comparisons involving `NULL`s 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.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

**NOT**

NOT	
TRUE	FALSE
FALSE	TRUE
NULL	NULL

The negation of NULL (NOT NULL) results in a null value because null values are indeterminate.

TRUE takes precedence in an OR condition.

FALSE takes precedence in an AND condition.

**OR**

OR	TRUE	FALSE	NULL
TRUE	TRUE	TRUE	TRUE
FALSE	TRUE	FALSE	NULL
NULL	TRUE	NULL	NULL

**AND**

AND	TRUE	FALSE	NULL
TRUE	TRUE	FALSE	NULL
FALSE	FALSE	FALSE	FALSE
NULL	NULL	FALSE	NULL



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can build a simple Boolean condition by combining number, character, or date expressions with comparison operators.

You can build a complex Boolean condition by combining simple Boolean conditions with the logical operators AND, OR, or 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 Expression 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 © 2017, Oracle and/or its affiliates. All rights reserved.

The AND logic table can help you to evaluate 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 © 2017, Oracle and/or its affiliates. All rights reserved.

## 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 © 2017, Oracle and/or its affiliates. All rights reserved.

Loops are programming language constructs that are used to execute a set of statements repeatedly until an exit condition is reached. It is mandatory to have an exit condition in a loop; otherwise, the loop executes for infinite number of iterations. PL/SQL provides three types of loop structures:

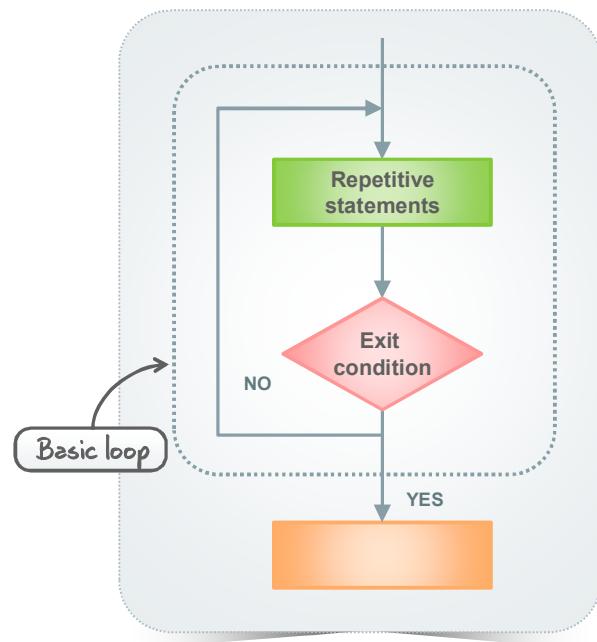
- Basic loop that performs repetitive actions till the exit condition is met
- FOR loops that perform iterative actions based on a count
- WHILE loops that perform iterative actions based on a condition

**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 © 2017, Oracle and/or its affiliates. All rights reserved.

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, the exit condition is checked. If the exit condition evaluates to `FALSE`, the statements in the loop are repeated. 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**

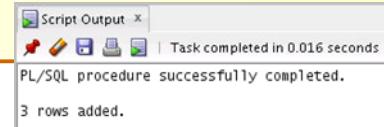
You use the `EXIT` statement to terminate a loop. Control passes to the next statement after the `END LOOP` statement. You can issue `EXIT` either as an action within an `IF` statement or as a stand-alone statement within the loop. The `EXIT` statement must be placed inside a loop. In the latter case, 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;
    DBMS_OUTPUT.PUT_LINE(v_counter-1||' rows added.');
END;
/

```



**ORACLE**

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

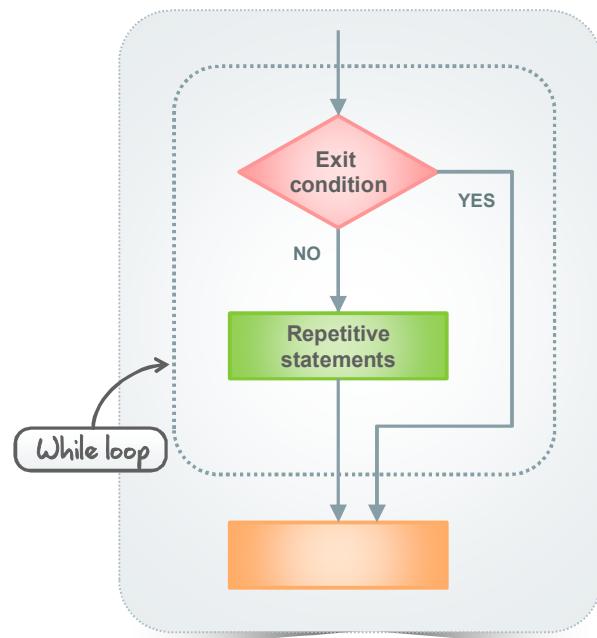
### Results

To view the output, run the code example under slide 23\_sa in `code_ex_06.sql`.

# WHILE Loops

Syntax:

```
WHILE condition LOOP
    statement1;
    statement2;
    ...
END LOOP;
```



**ORACLE®**

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 that are involved in the conditions do not change during the body of the loop, the condition remains TRUE and the loop does not terminate. It is an infinite loop.

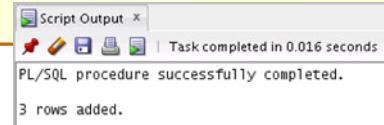
**Note:** If the condition yields NULL, the loop is bypassed and control passes to the next statement.

## 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;
    DBMS_OUTPUT.PUT_LINE(v_counter-1||' rows added.');
END;
/

```



**ORACLE**

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

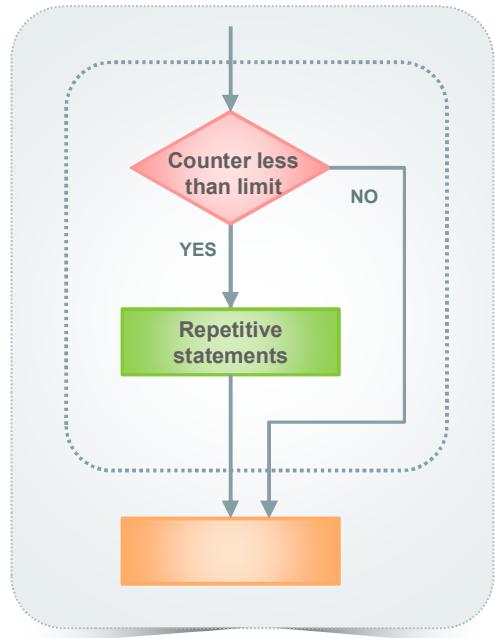
### Results

To view the output, run the code example under slide 25\_sa in `code_ex_06.sql`.

## FOR Loops

- Use a FOR loop when you know 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 © 2017, Oracle and/or its affiliates. All rights reserved.

FOR loops have the same general structure as the basic loop. You can use a FOR loop when you know the number of iterations that a given set of PL/SQL statements should execute.

In the syntax:

<i>counter</i>	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 <b>Note:</b> The lower bound is still referenced first.
<i>lower_bound</i>	Specifies the lower bound for the range of counter values
<i>upper_bound</i>	Specifies the upper bound for the range of counter values

Do not declare the counter. It is declared implicitly as an integer.

**Note:** The sequence of statements is executed each time the counter is incremented, as determined by the two bounds. The lower and upper bounds 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;
```

## 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;
/

```

Result in the LOCATIONS table

	LOCATION_ID	STREET_ADDRESS	POSTAL_CODE	CITY	STATE_PROVINCE	COUNTRY_ID
1	1901	(null)	(null)	Montreal	(null)	CA
2	1902	(null)	(null)	Montreal	(null)	CA
3	1903	(null)	(null)	Montreal	(null)	CA



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

The slide shows the three new rows that are inserted into the LOCATIONS table based on the logic specified.

## 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.
- The loop bound should not be `NULL`.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 © 2017, Oracle and/or its affiliates. All rights reserved.

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.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 functionality to begin the next loop iteration
  - Provides programmers with the ability to transfer control to the next iteration of a loop
- Benefits
  - Eases the programming process
  - May provide a small performance improvement over the previous programming workarounds to simulate the CONTINUE statement



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

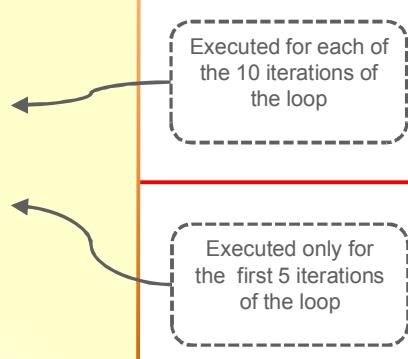
The CONTINUE statement enables you to transfer control within a loop to a new iteration or to leave the loop. Many other programming languages have this functionality. With the Oracle Database 11g release, PL/SQL also offers this functionality. Before the Oracle Database 11g release, you could code a 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
        1 v_total := v_total + i;
        DBMS_OUTPUT.PUT_LINE
            ('Total is: '|| v_total);
        CONTINUE WHEN i > 5;
        2 v_total := v_total + i;
        DBMS_OUTPUT.PUT_LINE
            ('Out of Loop Total is:
             '|| v_total);
    END LOOP;
END;
/
```



```
PL/SQL procedure successfully 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
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, there are two assignments that use 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 only for the first five iterations of the loop. The `CONTINUE` statement transfers control within the loop back to the beginning of the loop for 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.

## PL/SQL CONTINUE Statement: Example 2

```

DECLARE
  v_total NUMBER := 0;
BEGIN
  <><BeforeTopLoop>>
  FOR i IN 1..5 LOOP
    v_total := v_total + 1;
    DBMS_OUTPUT.PUT_LINE
      ('v_total(in BeforeTopLoop) : ' || v_total);
    FOR j IN 1..5 LOOP
      CONTINUE BeforeTopLoop WHEN i + j > 5;
      v_total := v_total + 1;
      DBMS_OUTPUT.PUT_LINE('v_total(in the inner loop) : ' || v_total);
    END LOOP;
  END LOOP BeforeTopLoop;
END;

```

```

Script Output X | Task completed in 0.003 sec
PL/SQL procedure successfully completed.

v_total(in BeforeTopLoop) : 1
v_total (in the inner loop) : 2
v_total (in the inner loop) : 3
v_total (in the inner loop) : 4
v_total (in the inner loop) : 5
v_total(in BeforeTopLoop) : 6
v_total (in the inner loop) : 7
v_total (in the inner loop) : 8
v_total (in the inner loop) : 9
v_total(in BeforeTopLoop) : 10
v_total (in the inner loop) : 11
v_total (in the inner loop) : 12
v_total(in BeforeTopLoop) : 13
v_total (in the inner loop) : 14
v_total(in BeforeTopLoop) : 15

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use the `CONTINUE` statement to jump to the next iteration of an outer loop.

To do this, provide the outer loop with 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, which is labeled `BeforeTopLoop` in this example.

When the pair of loops completes, the value of the `v_total` variable is 15.

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.

- a. True
- b. False



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

**Answer: a**

## 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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 6: Overview

This practice covers the following topics:

- Performing conditional actions by using `IF` statements
- Performing iterative steps by using `LOOP` structures



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.



7

# Working with Composite Data Types

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Course Road Map

Lesson 1: Course Overview

Unit 1: Introducing PL/SQL

**Unit 2: Programming with  
PL/SQL**

Unit 3: Working with  
PL/SQL Code

ORACLE®

▶ Lesson 6: Writing Control Structures

▶ **Lesson 7: Working with Composite Data Types**

You are here!

▶ Lesson 8: Using Explicit Cursors

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You are in lesson 7, which is part of Unit 2: Programming with PL/SQL.

# 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 © 2017, Oracle and/or its affiliates. All rights reserved.

You have already been introduced to composite data types. In this lesson, you learn more about composite data types and their uses.

# Agenda

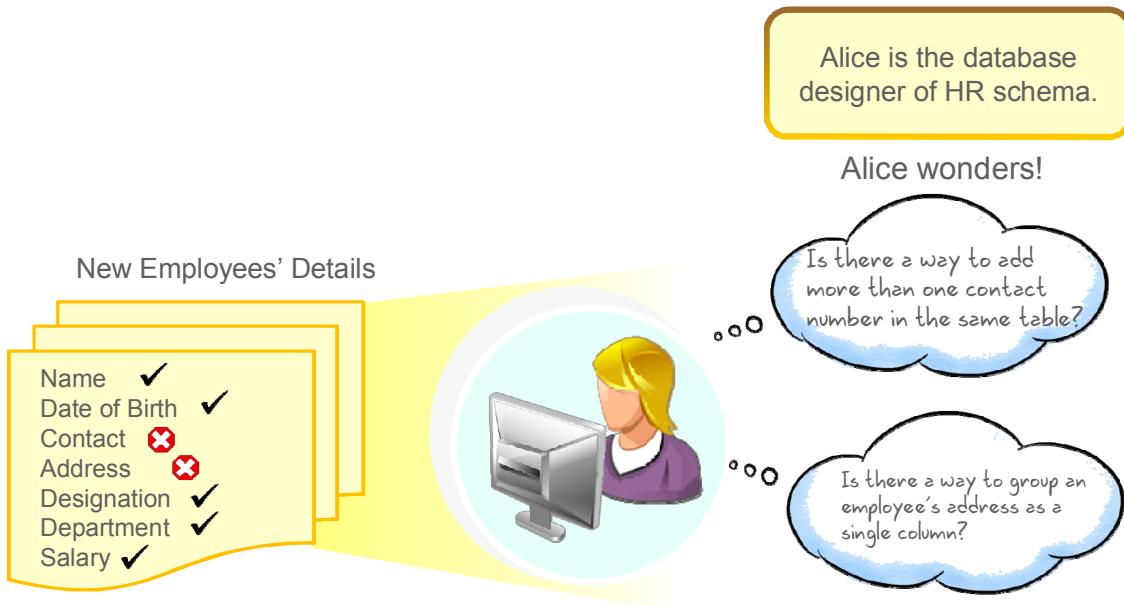
- Understanding composite data types
- Using PL/SQL records
  - Manipulating data with PL/SQL records
  - Advantages of the %ROWTYPE attribute
- Using PL/SQL collections
  - Examining associative arrays
  - Introducing nested tables
  - Introducing VARRAY



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Composite Data Types



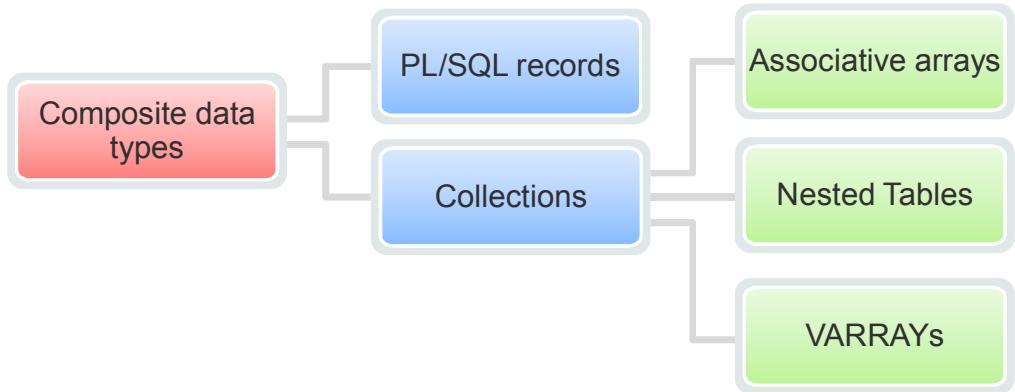
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Consider a scenario where you must store the address of an employee in a table and you may have to refer to each field of the address occasionally. In such a case, storing the complete address as a single record comprising the street name, door number, and so on would be an effective solution.

There are instances where an employee may have more than one contact number. At times, the employee may have only one. The number of contact numbers is not definite, yet you still want to have all the contact numbers in the database. Storing all the contact numbers as a collection would be an effective solution for this situation.

# Composite Data Types



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Composite data types hold data that may further have logical internal components. The internal components can be manipulated, and can be 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.

Each internal component of a record is termed as a field. You can access each component of the record through the following syntax:

```
variable_name.field_name
```

- **PL/SQL collections:** Collections allow you to store multiple rows of data in memory, enabling developers to treat data as a single unit. Collections are of three types:

- Associative array
- Nested table
- VARRAY

Each internal component of a collection is known as an element. You can refer to each element through an index:

```
variable_name(index)
```

## PL/SQL Records Versus Collections

PL/SQL Records	Collections
These are used to store related but dissimilar data as a logical unit.	These are used to store data as a single unit.
Use when you want to store values of different data types that are logically related.	Use when you want to store values of the same data type.
Each element can be accessed as: record_name.field_name.	Each element can be accessed by its unique subscript.
Example: A record to hold employee details that are related because they provide information about a particular employee	Example: A collection to hold the emails of all employees. It may store $n$ email IDs; however, email 1 is not related to email 2, and so on.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 emails of all employees. You may have stored  $n$  emails in the collection; however, email 1 is not related to email 2. The relation between these emails is only that they are employee emails. These collections are similar to arrays in programming languages such as C, C++, and Java.

# Agenda

- Understanding 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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## PL/SQL Records



- Update the job\_id of each employee.
- Add data to the job\_history table to reflect promotion.
- Modify the salary of the employee.

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

It is appraisal time and the HR manager has the job of updating all the organizational changes in the system. To reflect a promotion in the organization, the following operations are to be carried out in the database:

1. Update the job\_id in the employees table according to the promotion.
2. Add the old job details to the job\_history table in relation to the employee.
3. Update the salary field in the employees table according to the new job.

When you implement this task in PL/SQL, it would be a better option to retrieve all the details of the employee into the PL/SQL block, update or manipulate them, and then write the details back into the database.

Using a structure that can hold all the data from multiple tables would be an efficient way to execute this task.

A PL/SQL record allows you to hold data that is logically related. In this case, data relevant to an employee such as employee\_id, job\_id, salary, and so on are of different data types but they are logically connected because they belong to a single employee.

# 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
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

PL/SQL records are user-defined composite types. You must create the record type first, and then declare an identifier by using that type. 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 also use the %TYPE and %ROWTYPE attributes to declare the data types.)

*expr* Is the initial value

The NOT NULL constraint prevents assigning of nulls to the specified fields. Be sure to initialize the NOT NULL fields.

The field declarations that are used in defining a record are like variable declarations. Each field has a unique name and a specific data type.

## Creating a PL/SQL Record: Example

```

DECLARE
  TYPE t_rec IS RECORD
  (
    v_first_name employees.first_name%type,
    v_sal number(8),
    v_hire_date employees.hire_date%type,
  );
  v_myrec t_rec;
BEGIN
  SELECT first_name,salary, hire_date INTO v_myrec
    FROM employees WHERE employee_id = 100;
  DBMS_OUTPUT.PUT_LINE('First Name:
  '||v_myrec.v_first_name ||'Salary:
  '||v_myrec.v_sal ||'Hire Date:
  '|| v_myrec.v_hire_date);
END;

```

PL/SQL procedure successfully completed.

```

First Name:Steven
Salary: 24000
Hire Date: 17-JUN-11

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, a PL/SQL record is created by 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.

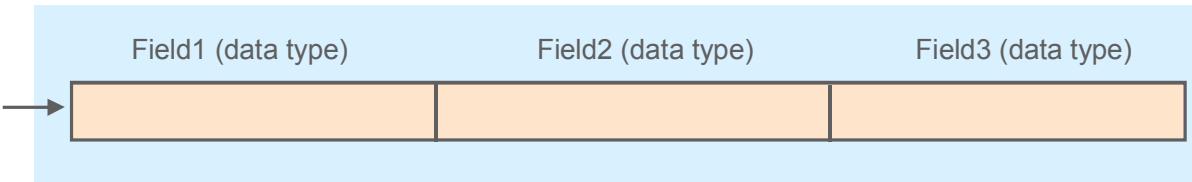
You retrieve values from the `employees` table through a `SELECT` statement into the record, and then access the fields of the record.

### Notes

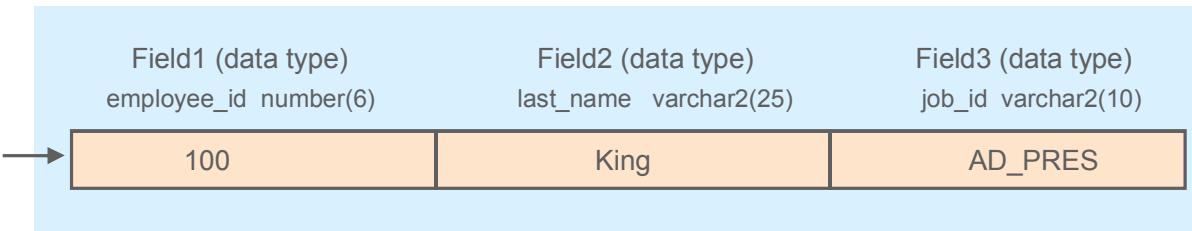
- The record contains three fields: `v_first_name`, `v_sal`, and `v_hire_date`.
- PL/SQL record fields are referenced by using the `<record>. <field>` notation.
- You can add the `NOT NULL` constraint to any field declaration to prevent assigning of nulls to that field. Remember that fields that are declared as `NOT NULL` must be initialized.

## PL/SQL Record Structure

Field declarations:



Example:



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You access the fields in a record 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.

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 one or more than one fields with different data types.
- You can assign initial values to the fields and also apply a NOT NULL constraint.
- Fields without initial values are initialized to NULL.
- You can use the DEFAULT keyword or := to initialize the fields.
- You can define the RECORD types and declare the 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.

## %ROWTYPE Attribute

- Declare a variable according to a collection of columns in a 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;
```

Prefix %ROWTYPE with the database table or view.

Example:

```
DECLARE
    employees    employees%ROWTYPE;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When you declare a record, you may want to have the record fields to be exactly same as the definition of some table in the database. You can use %ROWTYPE attribute in this scenario.

The %ROWTYPE attribute is to a record what the %TYPE attribute is to a variable type.

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 by using %ROWTYPE as a data type specifier:

```
DECLARE
    emp_record    employees%ROWTYPE;
    ...
    ...
```

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 only 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,
 job_id           VARCHAR2(10),
 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 that has 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.

## 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 || ' ' ||
    v_myrec.v_hire_date || ' '|| v_myrec.v_sal);
END;

```

PL/SQL procedure successfully completed.

King 11-JUL-16 1500



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows the usage of the ROWTYPE declaration. The code in the slide has a new field added, `v_rec1`, which is a row from the `employees` table.

The `v_rec1` field in the record declaration is a row from the `employees` table. The `v_myrec` variable is a record variable of type `t_rec`.

**Note:** A record can have other records as its fields.

In the executable block, we set the field values of the record `v_myrec`:

1. `v_myrec.v_sal` is set to `v_myrec.v_minsal + 500` where the value of `v_minsal` was set to a default of 1,000.
2. The value of `v_hire_date` is set to the current system date.
3. The record variable `v_rec1` is set to the row with `employee_id = 100` in the `employees` table through a `SELECT` statement. In the output, you access the last name of the employee record by identifying with the record variable, `v_myrec`, and then the composite field variable, `v_rec1`, and then the field `last_name`.

Observe the difference in accessing the scalar fields in the record and the values of the composite field in the record.

## 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
  - The row-level INSERT and UPDATE statements



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

Using %ROWTYPE simplifies maintenance, which is its main advantage. 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;								
EMP_NO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-15	11-JUL-16	5800	(null)



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 by 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)
)

```

### Notes

- The record that is inserted into the `retired_emps` table is shown in the slide.
- To see the output shown in the slide, execute the `SELECT` statement shown in the slide in SQL Developer.
- The complete code example is found under slide 14\_s-n in `code_ex_07.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;

```

EMP_NO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO	
1	124	Hourgos	ST_MAN	100	16-NOV-15	16-NOV-15	5800	(null)	50

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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`. Practically 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, execute the `SELECT` statement shown in the slide in SQL Developer.

## Updating a Row in a Table by Using a Record

```
DECLARE
    v_employee_number number:= 124;
    v_emp_rec    retired_emps%ROWTYPE;
BEGIN
    SELECT * INTO v_emp_rec FROM retired_emps WHERE
        empno = v_employee_number;
    v_emp_rec.leavedate:= CURRENT_DATE;
    UPDATE retired_emps SET ROW = v_emp_rec WHERE
        empno=v_employee_number;
END ;
/
SELECT * FROM retired_emps;
```

EMP_NO	ENAME	JOB	MGR	HIREDATE	LEAVEDATE	SAL	COMM	DEPTNO
1	124	Mourgos	ST_MAN	100	16-NOV-15	11-JUL-16	5800	(null)

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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` field of the employee.
- The record is updated as shown in the slide.

**Note:** To see the output shown in the slide, run the `SELECT` statement shown at the end of the code in the slide, in SQL Developer.

# Agenda

- Understanding 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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

As stated previously, PL/SQL collections are used when you want to store values of the same data type, such as a single dimensional array.

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 types of collections, the associative array is the focus of this lesson. The nested table and VARRAY are introduced only for comparative purposes.

## 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	Value
1	JONES
2	HARDEY
3	MADURO
4	KRAMER



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

An associative array is a type of PL/SQL collection. It is a single dimensional array where each element in the collection is referred to through an index value. You can store data in an associative array by using a primary key value as the index, where the key values are not necessarily sequential.

Associative arrays are sets of key-value pairs. Each key is a unique index, which is used to locate the associated value with the syntax `variable_name(index)`.

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.

You use associative arrays in PL/SQL blocks. You cannot store the data from associative arrays onto the database after completing execution of the PL/SQL block.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Consider a scenario where the HR Manager must generate a report of expenses per annum for each department to present it in the executive meeting. An expense for a department is the total salary paid to all its employees. While generating the report, the HR manager must extrapolate the expenses by 10% to include the inflation factor for that year. This is just a tentative report; therefore, it need not be retained in the database.

When you execute this task through a PL/SQL block, you may perform the following operations:

1. Execute a select query to find the sum of salaries paid to all employees, department-wise.
2. The `SUM` on each employee's salary would be the department expense. This value can be stored in an associative array that is indexed by the department number.
3. Multiply each entry in the array with 12 to find the expense per annum.
4. Add 10% to each entry in the array.
5. Display the associative array as output.

Using an associative array in this scenario is meaningful because you manipulate the data in the PL/SQL block. Also, the data that is generated in the PL/SQL block need not be retained in the database. You use associative arrays when you have to manipulate or process certain data in the PL/SQL block and need not persist the data in the database.

## Associative Array Structure

1

Unique key column
...
1
5
3
...

PLS\_INTEGER

OR

Value
...
Jones
Smith
Maduro
...

SCALAR

2

Value		
...	...	...
110	ADMIN	Jones
103	ADMIN	Smith
176	IT_PROG	Maduro
...	...	...

RECORD



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Associative arrays have two columns: a key column and a value column. The first column is a key column, which is used to index the value column. The key column helps you to uniquely access a certain value column. The second column holds a value.

**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 the scalar data type can hold only one value per row, whereas a column with the 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 an 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 either positive or negative.

# Steps to Create an Associative Array

Syntax:

```
TYPE type_name IS TABLE OF
{ column_type [NOT NULL] | variable%TYPE [NOT NULL]
| table.column%TYPE [NOT NULL]
| table%ROWTYPE }
INDEX BY { PLS_INTEGER | BINARY_INTEGER
| VARCHAR2(<size>) } ;
identifier type_name;
```

Example:

```
...
TYPE ename_table_type IS TABLE OF
employees.last_name%TYPE
INDEX BY PLS_INTEGER;
...
ename_table ename_table_type;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Two steps are involved in creating an associative array:

1. Declare a TABLE data type by using the INDEX BY option.
2. Declare a variable of that data type.

## Syntax

<i>type_name</i>	Is the name of the TABLE type (This name is used in the subsequent declaration of the array identifier.)
<i>column_type</i>	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.)
<i>identifier</i>	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 in the slide, 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 email_table IS TABLE OF
        employees.email%TYPE
        INDEX BY PLS_INTEGER;
    email_list      email_table;
BEGIN
    email_list(100) = 'SKING';
    email_list(105) = 'DAUSTIN';
    email_list(110) = 'JCHEN';
    DBMS_OUTPUT.PUT_LINE(email_list(100));
    DBMS_OUTPUT.PUT_LINE(email_list(105));
    DBMS_OUTPUT.PUT_LINE(email_list(110));
END;
/
...
```

PL/SQL procedure successfully completed.

SKING  
DAUSTIN  
JCHEN



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide creates an associative array with the identifier, `email_list`. In the executable section, the associative array is initialized with three elements at indexes 100, 105, and 110.

The index of each associative array element is used to access an element in the array, by using the following syntax:

```
identifier(index)
```

In the `DBMS_OUTPUT.PUT_LINE` statement, you access the elements of the associative array and display them as output.

**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.

## Associative Arrays with Record values

Define an associative array to hold an entire row from a table.

```

DECLARE
  TYPE dept_table_type
  IS
    TABLE OF departments%ROWTYPE INDEX BY VARCHAR2(20);
  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;
/

```

PL/SQL procedure successfully completed.  
10 Administration 200



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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. Associative arrays enable you to hold rows of data from a table in the value column.

### **Creating and referencing an associative array of rows from a table:**

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:

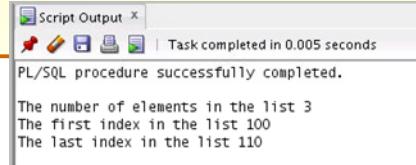
- PL/SQL record types can be user-defined, whereas %ROWTYPE implicitly defines the record.
- PL/SQL records enable you to specify the fields and their data types while declaring them. When you use %ROWTYPE, you cannot specify the fields. The %ROWTYPE attribute represents a table row with all the fields 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.

## Using Collection Methods

```

DECLARE
  TYPE email_table IS TABLE OF
    employees.email%TYPE
    INDEX BY PLS_INTEGER;
  email_list      email_table;
BEGIN
  email_list(100) = 'SKING';
  email_list(105) = 'DAUSTIN';
  email_list(110) = 'JCHEM';
  DBMS_OUTPUT.PUT_LINE('The number of elements in the list' || 
    email_list.COUNT);
  DBMS_OUTPUT.PUT_LINE('The first index in the list ' || email_list.FIRST);
  DBMS_OUTPUT.PUT_LINE('The last index in the list ' || email_list.LAST);
END;
/

```



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A collection method is a built-in procedure or function that operates on an associative array, and is called by using the dot notation as shown in the slide. Following are some of the commonly used functions:

Method	Description
EXISTS	<ul style="list-style-type: none"> <li>This returns TRUE if any elements exist in the collection.</li> <li>EXISTS(<i>n</i>) returns true if an element exists at the given index.</li> </ul>
COUNT	<ul style="list-style-type: none"> <li>This returns the number of elements that a collection currently contains.</li> </ul>
FIRST	<ul style="list-style-type: none"> <li>This returns the first (smallest) index number in a collection.</li> <li>This returns NULL if the collection is empty.</li> </ul>
LAST	<ul style="list-style-type: none"> <li>This returns the last (largest) index number in a collection.</li> <li>This returns NULL if the collection is empty.</li> </ul>
PRIOR ( <i>n</i> )	<ul style="list-style-type: none"> <li>This returns the index number that precedes index <i>n</i> in the collection.</li> </ul>
NEXT ( <i>n</i> )	<ul style="list-style-type: none"> <li>This returns the index number that succeeds index <i>n</i>.</li> </ul>
DELETE	<ul style="list-style-type: none"> <li>DELETE removes all the elements from a collection.</li> <li>DELETE(<i>n</i>) removes the index <i>n</i> from a collection.</li> <li>DELETE(<i>m</i>, <i>n</i>) removes all the elements in the range <i>m</i> ... <i>n</i> from a collection.</li> </ul>

In the slide we use the COUNT, FIRST, LAST, and EXISTS methods for the collection email\_list.

**Note:** There are EXTEND and TRIM procedures, which can be used only with varrays and nested tables.

## Using Collection Methods with Associative Arrays

```

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;
/

```

PL/SQL procedure successfully completed.

King
Kochhar
De Haan
Hunold
Ernst



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide declares an associative array, 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 is retrieved from the `EMPLOYEES` table 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. However, you can do the same more efficiently by using cursors. Cursors are explained in the lesson titled “Using Explicit Cursors.”

The results of the code example are as follows:

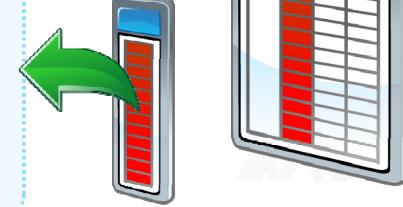
PL/SQL procedure successfully completed.

King
Kochhar
De Haan
Hunold
Ernst

## Nested Tables

- Nested tables represent sets of values.
- Nested tables can be used as column types in database tables.
- Each entry in the column of the database table can hold a set of values.

index	value
1	SKING
2	NKOCHAR
3	LDEHAAN
4	AHUNOLD



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Consider a situation where the HR manager has to store all the email IDs of employees working in a department. In this case, using nested tables would be an appropriate choice.

Unlike associative arrays, nested tables can be defined as columns of a table in the database, where each entry in the column can hold a set of values.

Nested tables represent sets of values. You can think of them as one-dimensional arrays with no declared number of elements.

Recall that in the case of associative arrays, the index value can be explicitly provided in the PL/SQL block. When you retrieve a nested table from the database into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. The indexing of elements is done by PL/SQL.

## Nested Tables: Syntax and Usage

### Syntax:

```
TYPE type_name IS TABLE OF  
{column_type | variable%TYPE  
| table.column%TYPE} [NOT NULL]  
| table.%ROWTYPE
```

### Example:

```
DECLARE  
    TYPE dept_mail IS TABLE OF VARCHAR2(20);  
    mails dept_mail := dept_mail('SKING', 'NKOCHAR', 'LDEHAAN',  
    'AHUNOLD');  
BEGIN  
    FOR i IN mails.FIRST..mails.LAST LOOP  
        DBMS_OUTPUT.PUT_LINE(mails(i));  
    END LOOP;  
END;  
/
```

```
PL/SQL procedure successfully completed.  
SKING  
NKOCHAR  
LDEHAAN  
AHUNOLD
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the slide, you see the syntax for creating nested tables and usage of the nested tables.

In the example, a nested table type, `dept_mail`, is created. The nested table will hold data of type `VARCHAR2`. A variable `mails` of type `dept_mail` is declared in the `DECLARE` section.

The nested table is initialized with values in the `DECLARE` section. In the executable section, you retrieve these values with a `FOR` loop.

The functionality of nested tables is similar to that of associative arrays; however, there are differences in a nested table implementation.

- A 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 in a nested table (unlike in an associative array). Although 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 non-sequential “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.

## Variable-Sized Arrays (Varrays)

```
TYPE emp_mail IS VARRAY(5) OF  
  VARCHAR2(20);  
  mails emp_mail;
```

Declaration

mails(1)	mails(2)	mails(3)	mails(4)	mails(5)
----------	----------	----------	----------	----------

Subscripts to access each varray



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Consider a situation where a company has an upper limit on the number of employees working in each department. In this case, we can use varrays instead of nested tables to hold the email IDs. A variable-size array (VARRAY) is an array with a fixed number of elements. The number of elements that the array would accommodate must be defined during declaration.

- A VARRAY is valid in a schema-level table.
- Items of the VARRAY type are called varrays.
- VARRAYS have a fixed upper bound on the number of elements, and 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.
- To access an element of a varray, you use standard subscripting syntax.

The declaration in the slide shows a VARRAY type, `emp_mail`, of size 5. The identifier `mails` is a variable of type `emp_mail`. Observe that the varray is uninitialized.

The subscripts to access each element of the varray starts with 1, and increments according to the size of the varray.

## VARRAYS: Syntax and Usage

### Syntax:

```
TYPE type_name IS VARRAY(SIZE) OF
{column_type | variable%TYPE
| table.column%TYPE} [NOT NULL]
| table.%ROWTYPE
```

### Example:

```
DECLARE
    TYPE emp_mail IS VARRAY(5) OF VARCHAR2(20);
    mails emp_mail := emp_mail('NKOCHAR', 'LDEHAAN', 'AHUNOLD');
BEGIN
    FOR i IN mails.FIRST..mails.LAST LOOP
        DBMS_OUTPUT.PUT_LINE(mails(i));
    END LOOP;
END;
/
```

PL/SQL procedure successfully completed.  
NKOCHAR  
LDEHAAN  
AHUNOLD



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the slide, you see the syntax for creating a varray and usage of the varray.

In the example in the slide, a varray type, `emp_mail`, is created. It will hold data of type `VARCHAR2` with a maximum size of 5 defined. A variable `mails` of type `emp_mail` is declared in the `DECLARE` section.

The varray is initialized with values in the `DECLARE` section. In the executable section, you retrieve these values with a `FOR` loop.

You will learn more about nested tables and varrays in the *Advanced PL/SQL* course.

## Summarizing Collections

### Associative arrays

- Are key-value pairs
- Are used only in PL/SQL blocks; cannot be persisted to the database

### Nested tables

- Are sequentially indexed data
- Can be persisted to the database as columns of a table

### Varrays

- Are sequentially indexed data with an upper limit
- Can be persisted to the database as columns of a table



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide compares the three types of collections. The following table has more details:

Associative Arrays	Nested Tables	Varrays
These are key-value pairs where each key is a unique index associated with a value.	These store an unspecified number of elements in no particular order.	These store a specified number of elements.
The values in an associative array are indexed according to the key value provided during initialization.	The elements in a nested table are indexed starting from 1.	Varrays are indexed from 1 to the upper limit specified during declaration in the PL/SQL block.
They cannot be persisted to a database, and can be used only in PL/SQL blocks.	They can be a table column and persisted to the database.	They can be a table column and persisted to the database.

# Quiz



Identify situations in which you can use the %ROWTYPE attribute.

- a. When you are not sure about the structure of the underlying database table
- b. When you want to retrieve an entire row from a table
- c. When you want to declare a variable according to another previously declared variable or database column



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

**Answer: a, b**

## 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 how 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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 makes 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 the `TABLE` type that look similar to database tables, but come 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 the `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 7: 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 © 2017, Oracle and/or its affiliates. All rights reserved.

In this practice, you define, create, and use associative arrays and PL/SQL records.



8

# Using Explicit Cursors

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Course Road Map

Lesson 1: Course Overview

Unit 1: Introducing PL/SQL

**Unit 2: Programming with PL/SQL**

Unit 3: Working with PL/SQL Code

▶ Lesson 6: Writing Control Structures

▶ Lesson 7: Working with Composite Data Types

▶ **Lesson 8: Using Explicit Cursors**

You are here!

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You are in lesson 8, which is part of Unit 2: Programming with PL/SQL.

## 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 © 2017, Oracle and/or its affiliates. All rights reserved.

You have learned about implicit cursors that are automatically created by PL/SQL when you execute a SQL SELECT or DML statement. The SELECT statements that are executed with an INTO clause are implicit cursors. 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

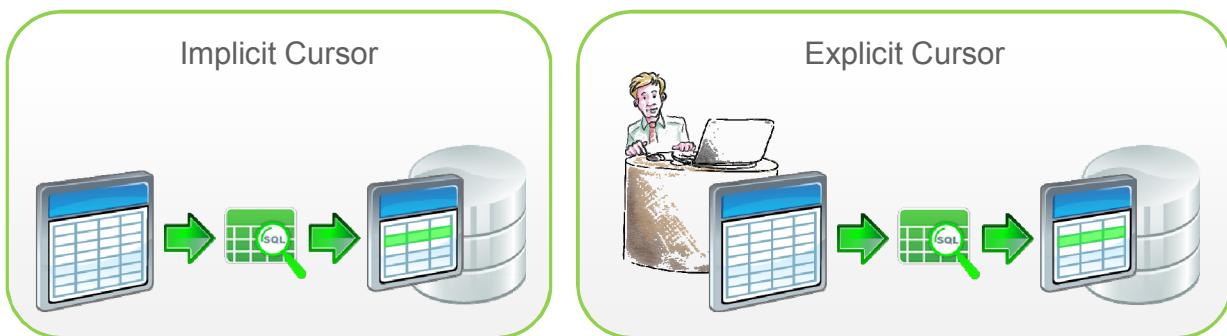


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Cursors

- A cursor is a pointer to a private memory area that is used to execute SQL statements in PL/SQL blocks.
- There are two type of cursors:
  - Implicit cursors
  - Explicit cursors



ORACLE®

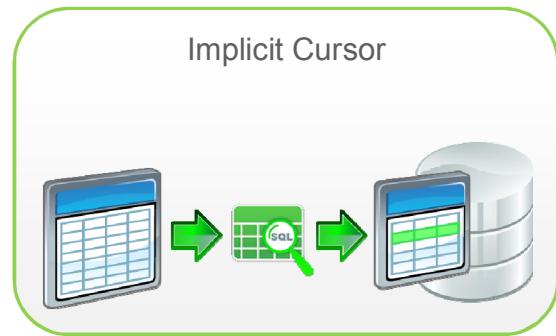
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The Oracle Server uses work areas (called *private SQL areas*) to execute SQL statements, and to store processing information. Whenever you execute a SELECT or DML statement in a PL/SQL block, an implicit cursor is created by the PL/SQL engine. However, you can create and use explicit cursors according to your application requirement.

Cursor Type	Description
Implicit	Implicit cursors are created and managed by PL/SQL. They are also known as session cursors.
Explicit	For queries that return multiple rows, explicit cursors are declared and managed by the programmer.

## Implicit Cursors

- Are created and managed by PL/SQL
- Are created for `SELECT` or DML statements
- Closes after the associated statement is run



ORACLE®

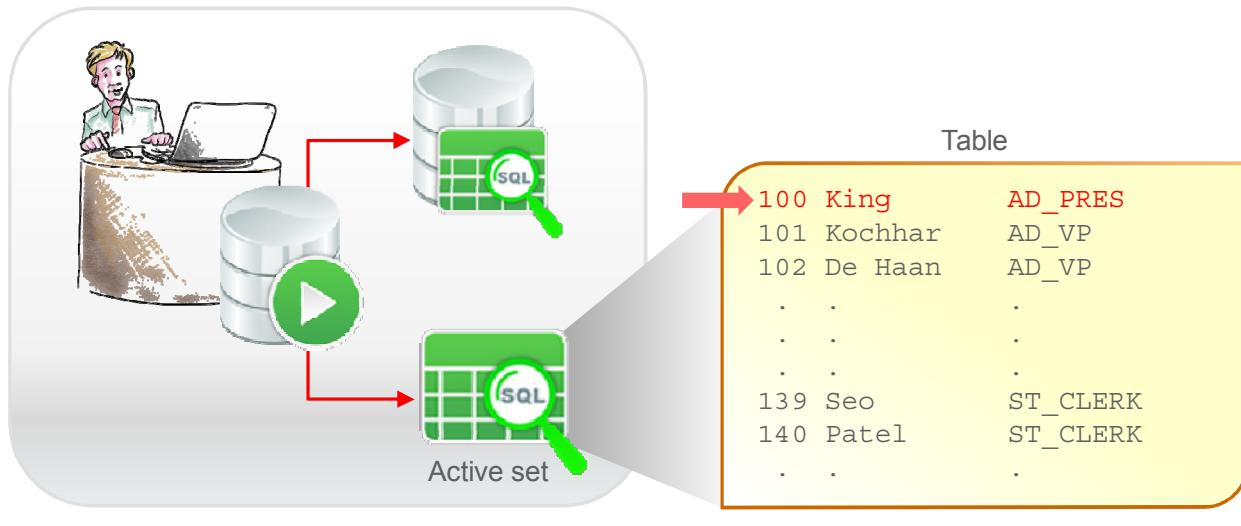
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Implicit cursors are created by the PL/SQL engine when a `SELECT` statement is executed with the `INTO` clause, or when `INSERT`, `UPDATE`, or `DELETE` statements are executed in the PL/SQL block.

The cursor closes after executing the statement associated with it. You cannot control an implicit cursor but you can get information about the implicit cursor through its attributes.

## Explicit Cursor

An explicit cursor is created and managed by a programmer in the PL/SQL block.



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

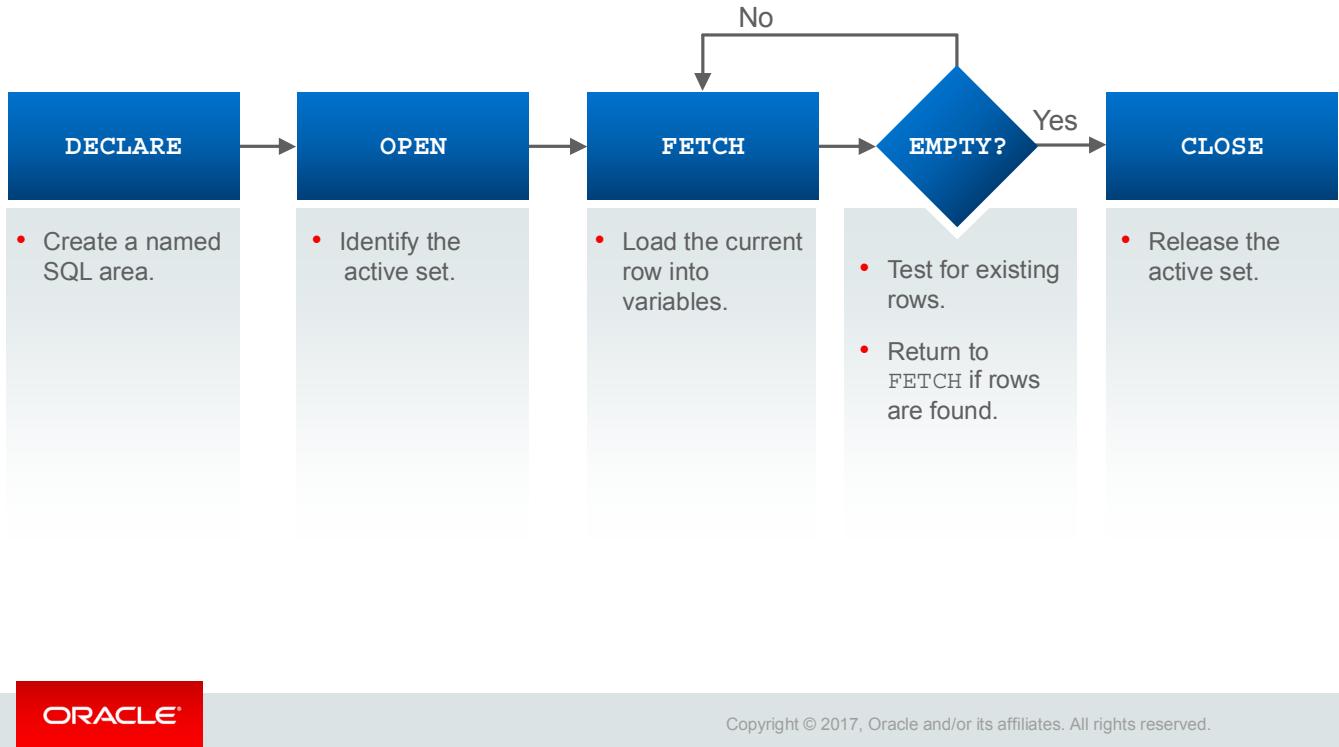
You must declare and define an explicit cursor, giving it a name and associating it with a `SELECT` statement (typically, the query returns multiple rows). You can process each row returned by the `SELECT` statement through a `FOR` loop.

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

## Controlling Explicit Cursors



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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, binds any variables that are referenced, and positions the cursor at the first row. The rows that are 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 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.  
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.
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.

# Agenda

- What are explicit cursors?
- Using explicit cursors
- Using cursors with parameters
- Locking rows and referencing the current row



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# 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;
    ...

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The syntax to declare a cursor is shown in the slide. In the syntax:

<i>cursor_name</i>	Is a PL/SQL identifier
<i>select_statement</i>	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. This is because you are only defining a cursor in the declarative section, and not retrieving any rows into the cursor.

## Notes

- 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.

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.

## 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;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 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.

## 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;
/
```

```
PL/SQL procedure successfully completed.
114  Raphaely
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The `FETCH` statement retrieves rows from a cursor, one at a time. After each fetch, the cursor advances to the next row in the active set.

Consider the example in the slide. Two variables, `v_empno` and `v_lname`, are declared to hold the values that are fetched 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

You should 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. If a fetch acquires no values, there are no rows left to process in the active set and no error is recorded.

## 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;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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:

PL/SQL procedure successfully completed.

```
114  Raphaely
115  Khoo
116  Baida
117  Tobias
118  Himuro
119  Colmenares
```

Apart from `NOTFOUND`, there are other cursor attributes that you can use to check on various aspects of the cursors. We will discuss them in the following slides.

## 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;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 a PL/SQL block without closing cursors, you should make it a practice to close any cursor that you declare to free resources. The amount of memory used by an application is directly proportional to the number of open cursors, which would also affect the performance of the application.

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;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

PL/SQL procedure successfully 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 © 2017, Oracle and/or its affiliates. All rights reserved.

You learned to fetch data from cursors by using simple loops. You now learn to use a cursor FOR loop, which processes the 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:

<i>record_name</i>	Is the name of the implicitly declared record
<i>cursor_name</i>	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;
/
```

PL/SQL procedure successfully completed.

```
114  RaphaeLy
115  Khoo
116  Baida
117  Tobias
118  Himuro
119  Colmenares
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 by 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 number of rows that has been fetched



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

As with implicit cursors, there are four attributes for obtaining status information about 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...
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

- 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 ; /

```

PL/SQL procedure successfully completed.

174	Abel
166	Ande
130	Atkinson
105	Austin
204	Baer
116	Baida
167	Banda
172	Bates
192	Bell
151	Bernstein



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows how the `%ROWCOUNT` and `%NOTFOUND` attributes can be used for the exit conditions in a loop.

In the given code, the exit condition for the loop was either that the cursor displayed 10 rows or the cursor cannot fetch any rows according to the `SELECT` statement.

The `%ROWCOUNT` attribute keeps track of the number of rows retrieved through the cursor. The `%NOTFOUND` attribute checks for the end of the active set of rows retrieved by the cursor.

The output of the code displays 10 rows from the `employees` table.

## 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;
/
```

PL/SQL procedure successfully completed.

```
114  Raphaely
115  Khoo
116  Baida
117  Tobias
118  Hinuro
119  Colmenares
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Note that there is no declarative section in this PL/SQL block. The difference between the cursor FOR loops by using subqueries and the cursor FOR loop lies in the cursor declaration. If you are writing cursor FOR loops by 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 in the slide to illustrate a cursor FOR loop by 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 © 2017, Oracle and/or its affiliates. All rights reserved.

## Cursors with Parameters

- 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.

Syntax:

```
CURSOR cursor_name
      [ (parameter_name datatype, ... ) ]
IS
  select_statement;
```

```
OPEN cursor_name (parameter_value, . . . . .) ;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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. The 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:

<i>cursor_name</i>	Is a PL/SQL identifier for the declared cursor
<i>parameter_name</i>	Is the name of a parameter
<i>datatype</i>	Is the scalar data type of the parameter
<i>select_statement</i>	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;
    v_empno employees.employee_id%TYPE;
    v_lname employees.last_name%TYPE;
BEGIN
  OPEN c_emp_cursor (30);
  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;
  CLOSE c_emp_cursor;
  DBMS_OUTPUT.PUT_LINE('Opening the cursor the second time');
  OPEN c_emp_cursor (10);
  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;
  CLOSE c_emp_cursor;
END;

```

```

PL/SQL procedure successfully completed.

114  Raphaely
115  Khoo
116  Baida
117  Tobias
118  Hinmuro
119  Colmenares
Opening the cursor second time
200  Whalen

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide shows the usage of cursors with parameters. Each time you open a cursor, you can pass a parameter to the cursor to create a new active set of records.

In the example in the slide, you pass `department_id` as a parameter.

The first time, you open the cursor with `department_id` 30. The employee IDs and last names of all employees working in the department with department number 30 are retrieved.

You close the cursor. The second time, you open it with parameter 10. Now the employee ID and last name of the employee working in department number 10 is retrieved.

# Agenda

- What are explicit cursors?
- Using explicit cursors
- Using cursors with parameters
- Locking rows and referencing the current row



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## 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* an update or a delete.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 essential to have locks on the rows before you update or delete rows. In the absence of locks, the data would be updated inconsistently. You can lock rows in databases with the FOR UPDATE clause in the cursor query. Locks are used in databases to maintain data integrity when multiple users are trying to access data.

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

WAIT *n* Waits for *n* seconds of time for the rows to be unlocked if they are locked by other sessions

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)*.

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 execute other steps 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 check 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 you must specify which tables to lock if the statement is a join of multiple tables.

## WHERE CURRENT OF Clause

- 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.

Syntax:

```
WHERE CURRENT OF cursor ;
```

```
UPDATE employees  
    SET salary = ...  
 WHERE CURRENT OF c_emp_cursor;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 to perform an operation such as UPDATE or DELETE on the current row pointed by the cursor. The FOR UPDATE clause is specified in the cursor declaration.

You must include the FOR UPDATE clause in the cursor query so that the rows are locked on OPEN.

In the syntax:

<i>cursor</i>	Is the name of a declared cursor (The cursor must have been declared with the FOR UPDATE clause.)
---------------	---

## WHERE CURRENT OF Clause: Example

```
DECLARE
    CURSOR c_emp_cursor IS
        SELECT employee_id, salary FROM employees
        WHERE department_id =30 FOR UPDATE;
    BEGIN
        FOR emp_record IN c_emp_cursor
        LOOP
            DBMS_OUTPUT.PUT_LINE( emp_record.employee_id
            ||' '||emp_record.salary);
            UPDATE employees
            SET salary = 5000
            WHERE CURRENT OF c_emp_cursor;
        END LOOP;
    END ;
/
SELECT employee_id, salary FROM employees
WHERE department_id =30;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The code in the slide shows the usage of the FOR UPDATE clause and the WHERE CURRENT OF clause in a PL/SQL block.

The cursor retrieves the employee\_id and salary of employees who belong to the department with department\_id 30.

The output of the execution of the code, which shows the value before the UPDATE on the employees table, is as follows:

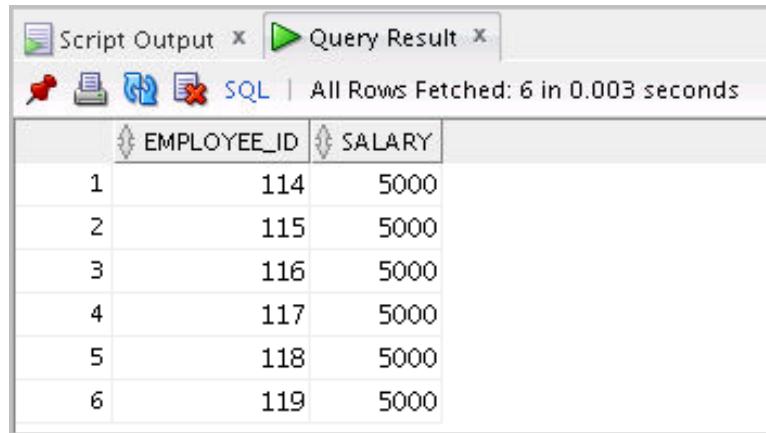
The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output'. It contains a toolbar with icons for script, edit, file, and help. Below the toolbar, a message says 'Task completed in 0.018 seconds'. The main area displays the message 'PL/SQL procedure successfully completed.' followed by a list of employee IDs and salaries:

```
114 11000
115 3100
116 2900
117 2800
118 2600
119 2500
```

After executing the PL/SQL block, you can see the effect of the UPDATE by executing the following SELECT statement:

```
SELECT employee_id, salary FROM employees  
WHERE department_id = 30;
```

The following is the output of the query, where all the salary values are set to 5000.



The screenshot shows the Oracle SQL Developer interface with the 'Query Result' tab selected. The results show six rows of data from the 'employees' table, where all employees in department 30 have been updated to a salary of 5000.

EMPLOYEE_ID	SALARY
114	5000
115	5000
116	5000
117	5000
118	5000
119	5000

## Quiz



Explicit cursor functions enable a programmer to manually control explicit cursors in the PL/SQL block.

- a. True
- b. False



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

**Answer: a**

## Summary

In this lesson, you should have learned how 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 © 2017, Oracle and/or its affiliates. All rights reserved.

The Oracle Server uses work areas to execute SQL statements and to store processing information. You can use a PL/SQL construct called a *cursor* to name a work area and to 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. These are called explicit cursors.

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 each of 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 that 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 8: Overview

This practice covers the following topics:

- Declaring and using explicit cursors to query the 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 © 2017, Oracle and/or its affiliates. All rights reserved.

In this practice, you apply your knowledge of cursors to process a number of rows from a table and to populate another table with the results by using a cursor FOR loop. You also write a cursor with parameters.



9

# Handling Exceptions

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Course Road Map

Lesson 1: Course Overview

Unit 1: Introducing PL/SQL

Unit 2: Programming with PL/SQL

**Unit 3: Working with PL/SQL Code**

▶ Lesson 9: Handling Exceptions

▶ Lesson 10: Creating Procedures and Functions

You are here!

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In Unit 3, you are introduced to exception handling in PL/SQL blocks and programming PL/SQL program units.

## 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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 a PL/SQL block.

# Agenda

- Understanding PL/SQL exceptions
- Trapping exceptions



ORACLE®

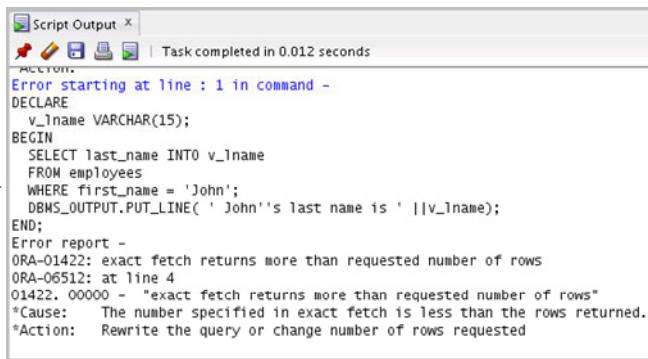
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## 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;

```



The screenshot shows a 'Script Output' window from Oracle SQL Developer. The title bar says 'Script Output X' and 'Task completed in 0.012 seconds'. The content area starts with 'Error starting at line : 1 in command -' followed by the PL/SQL code. Below the code, the error message is displayed: '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'. A red arrow points from the left margin of the slide towards the error message in the screenshot.

```

Error starting at line : 1 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

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Consider the example in the slide. There are no syntax errors in the code, which implies that you must be successfully executing the PL/SQL block. The SELECT statement in the block retrieves the last name of the employee whose first name is “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, for better application performance.

## Handling an Exception: 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;
/
```

PL/SQL procedure successfully completed.  
Your select statement retrieved multiple rows.  
Consider using a cursor



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You 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 occurs, control shifts to the exception section and all the statements in the exception section are executed. The PL/SQL block terminates with a 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



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 define an exception handler to perform final actions before the block ends.

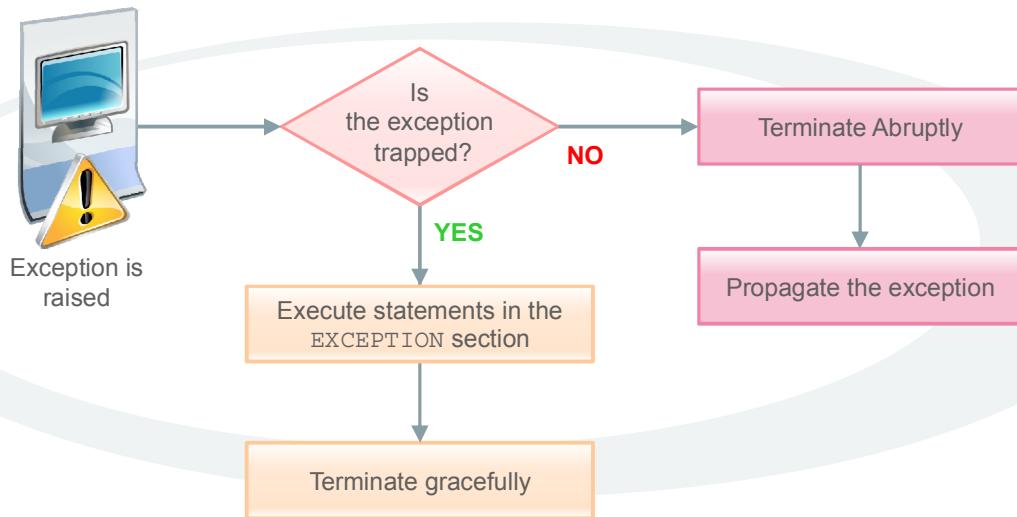
## Two methods for raising an exception

- An Oracle error occurs and the associated exception is raised automatically.  
For example, when you execute a `SELECT` statement in a PL/SQL block that does not retrieve any rows, an ORA-01403 error occurs. These are exceptions raised by the runtime system implicitly.
- Depending on the business functionality that your program implements, you may have to explicitly raise an exception. You raise an exception explicitly by issuing the `RAISE` statement in the block.

## Two methods of handling exceptions

- To handle an exception, you have to anticipate the exception and write code for handling the exception in an `EXCEPTION` block.
- The exception propagates to the host environment when there is no exception handler defined in the PL/SQL block.

# Handling Exceptions



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Trapping an Exception

Include an `EXCEPTION` section in your PL/SQL program to trap exceptions. If an exception is raised in the executable section of a block, processing branches to the corresponding exception handler in the `EXCEPTION` section of the block. If an exception is raised in a block that has no exception handler for it, the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a block has a handler for it. 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 an exception is raised in the executable section of a 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), which determines the outcome of the exception.

## Exception Types

- Internally defined
  - Predefined
- }
- Implicitly raised
- 
- User-defined
- Explicitly raised



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

**There are three types of exceptions:**

Exception	Description	Directions for Handling
Internally defined	ORA-n errors, which are implicitly raised by the runtime system	You can explicitly give names to these exceptions and write the corresponding EXCEPTION blocks.
Predefined	Exceptions declared globally in the package STANDARD. They have predefined names such as INVALID_CURSOR, CASE_NOT_FOUND.	You write an EXCEPTION block in the executable section with the name of the predefined exception.
User-defined error	Exceptions defined by user according to the application context. You have to declare these exceptions and their corresponding names in DECLARE section.	You write an EXCEPTION block based on the name that you used to declare the exception in the declarative section.

# Agenda

- Understanding PL/SQL exceptions
- Trapping exceptions



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Syntax to Trap Exceptions

```
EXCEPTION
  WHEN exception1 [OR exception2 . . .] THEN
    statement1;
    statement2;
    . . .
  [WHEN exception3 [OR exception4 . . .] THEN
    statement1;
    statement2;
    . . .]
  [WHEN OTHERS THEN
    statement1;
    statement2;
    . . .]
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

The exception-trapping syntax includes the following elements:

<code>exception</code>	Is the standard name of a predefined exception or the name of a user-defined exception that is declared within the declarative section
<code>statement</code>	Is one or more PL/SQL or SQL statements
<code>OTHERS</code>	Is an optional exception-handling clause that traps any exceptions that have not been explicitly handled

## The 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 exceptions that are not yet handled. For this reason, if the OTHERS handler is used, it must be the last exception handler that is defined.

**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 a 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.

## 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.

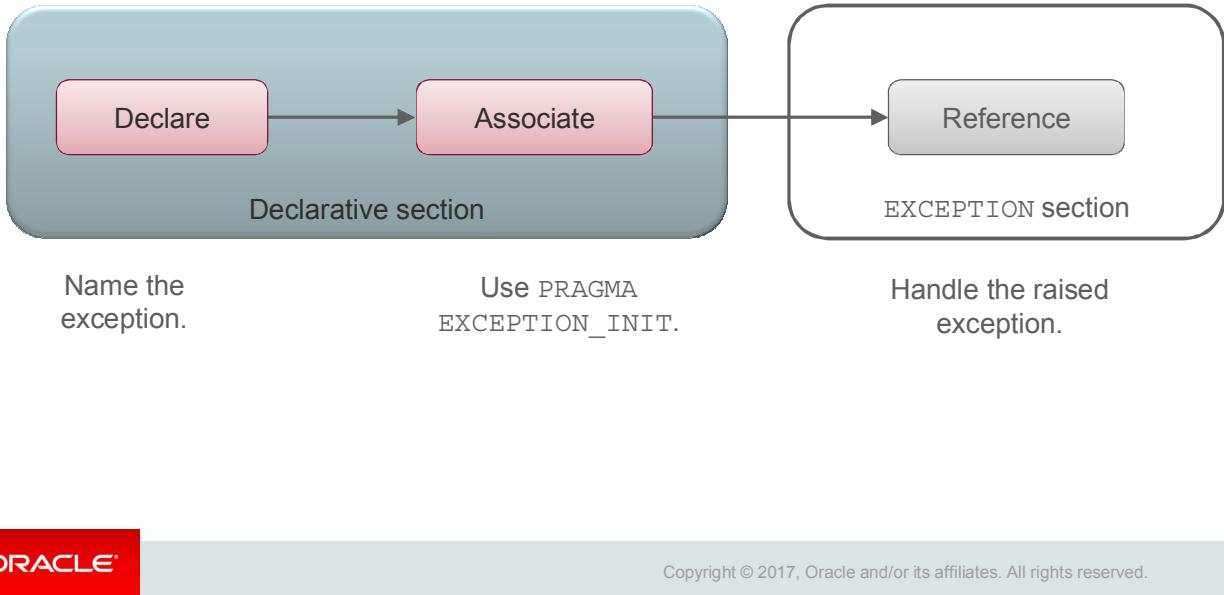


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

- 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 or SQL statements.

## Trapping Internally Predefined Exceptions



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Internally defined exceptions are ORA-n errors. You define EXCEPTION blocks for internally defined exceptions by using the PRAGMA EXCEPTION\_INIT function.

You declare an exception name and associate it with an internally defined exception in the declarative section. PRAGMA EXCEPTION\_INIT associates 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.

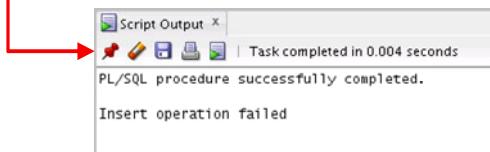
## Internally Defined Exception Trapping: Example

To trap Oracle Server error 01400 (“cannot insert NULL”):

```

DECLARE
  e_insert_excep EXCEPTION;
  PRAGMA EXCEPTION_INIT(e_insert_excep, -01400);
BEGIN
  INSERT INTO departments
    (department_id, department_name) VALUES (280, NULL);
EXCEPTION
  WHEN e_insert_excep THEN
    DBMS_OUTPUT.PUT_LINE('INSERT OPERATION FAILED');
  END;
/

```



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide illustrates the three steps in defining an EXCEPTION block for internally defined exceptions:

1. Declare the name of the exception in the declarative section by using the following 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 a 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.

## Trapping Predefined Exceptions

- Reference the predefined name in the EXCEPTION block.
- Some sample predefined exceptions are:
  - NO\_DATA\_FOUND
  - TOO\_MANY\_ROWS
  - INVALID\_CURSOR
  - ZERO\_DIVIDE
  - DUP\_VAL\_ON\_INDEX

```
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    statement1;
  ...
  WHEN TOO_MANY_ROWS THEN
    statement1;
  ...
  WHEN OTHERS THEN
    statement1;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

The following table shows some of the predefined exceptions:

<b>Exception Name</b>	<b>Oracle Server Error Number</b>	<b>Description</b>
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 were 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	Occurrence of an illegal cursor operation
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 +100	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 the PL/SQL cursor variable involved in an assignment have incompatible return types.

<b>Exception Name</b>	<b>Oracle Server Error Number</b>	<b>Description</b>
STORAGE_ERROR	ORA-06500	PL/SQL ran out of memory, or memory is corrupted.
SELF_IS_NULL	ORA-30625	Function invoked with a <code>NULL</code> object instance
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Referenced a nested table or VARRAY element by using an index number that is 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 failed because the character string does not represent a valid ROWID.
TIMEOUT_ON_RESOURCE	ORA-00051	Timeout occurred while the Oracle Server was waiting for a resource.
TOO_MANY_ROWS	ORA-01422	A single-row <code>SELECT</code> returned multiple rows.
VALUE_ERROR	ORA-06502	Arithmetic, conversion, truncation, or size-constraint error occurred.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero

## Functions for Trapping Exceptions

- **SQLCODE:** Returns the numeric value for the error code
- **SQLERRM:** Returns the message associated with the error number



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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
<code>SQLCODE</code>	Returns the numeric value for the error code (You can assign it to a <code>NUMBER</code> variable.)
<code>SQLERRM</code>	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	<code>NO_DATA_FOUND</code> exception
<i>negative number</i>	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;
    /

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

## Declare

- Name the exception in the `DECLARE` section.

## Raise

- Define the condition when the exception must be raised in executable section.

## Reference

- Handle the exception.

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

PL/SQL enables you to define your own exceptions depending on the requirements of your application.

For example, you may prompt a user to enter the age value in a valid range if the user entered an invalid age value. You must define an exception here, which checks on the range of value of age. If the age is not in the valid range, you may raise a 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

## RAISE Statement

- Stops normal execution of a PL/SQL block or subprogram, and transfers control to an exception handler
- Explicitly raises predefined exceptions or user-defined exceptions
- Syntax:

```
RAISE exception_name ;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the syntax, `exception_name` is the name of a predefined or a user-defined exception. `exception_name` is optional only in an exception handler, where the default is the current exception. Outside an exception handler, you must specify the name of the exception.

# 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;
/

```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 Sub-Block

Sub-blocks 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;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When a sub-block handles an exception, it terminates normally. Control resumes in the enclosing block immediately after the sub-block's `END` statement.

However, if 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 is propagated to the host environment.

When the exception propagates to an enclosing block, the remaining executable actions in that sub-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 (`e_no_rows` and `e_integrity`) are declared in the outer block. In the inner block, when the `e_no_rows` exception is raised, PL/SQL looks for the exception handler in the sub-block. Because the exception is not handled in the sub-block, the exception propagates to the outer block, where PL/SQL finds the handler.

## The 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 © 2017, Oracle and/or its affiliates. All rights reserved.

Use the RAISE\_APPLICATION\_ERROR procedure to communicate a predefined exception interactively by returning a nonstandard error code and an error message. With RAISE\_APPLICATION\_ERROR, you can report errors to your application and avoid returning unhandled exceptions.

In the syntax:

<i>error_number</i>	Is a user-specified number for the exception between –20,000 and –20,999
<i>message</i>	Is a 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.)

## The RAISE\_APPLICATION\_ERROR Procedure

- Is used in two different places:
  - Executable section
  - Exception section
- Returns error conditions to the user in a manner that is consistent with other Oracle Server errors



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

## The RAISE\_APPLICATION\_ERROR Procedure

```
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_APPLICATION_ERROR(-20202, 'Department number does not exist');
  END IF;
END;
```

```
Script Output X | Task completed in 0.009 seconds
-----+
  UPDATE departments
  SET department_name = v_name
  WHERE department_id = v_deptno;
  IF SQL%NOTFOUND THEN
    RAISE_APPLICATION_ERROR(-20202, 'Department number does not exist');
  END IF;
END;
Error report -
ORA-20202: Department number does not exist
ORA-06512: at line 10
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide shows how the RAISE\_APPLICATION\_ERROR procedure can be used in the executable section of a PL/SQL program. When called, RAISE\_APPLICATION\_ERROR ends the subprogram and returns a user-defined error number and message to the application. The error number and message can be trapped like any Oracle error.

You can see in the output that the error displayed is the one defined in the PL/SQL block.

## Quiz



You can trap any error by including a corresponding handler within the exception-handling section of the PL/SQL block.

- a. True
- b. False



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

### Answer: a

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 how 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:
  - Internally defined exceptions
  - Predefined exceptions
  - User-defined exceptions
- Propagate exceptions in nested blocks and call applications



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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. Internally defined exceptions can be any standard Oracle Server errors. Predefined exceptions are error conditions that are defined by the Oracle Server. User-defined exceptions are exceptions that are 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 control is transferred to the exception-handling section of your PL/SQL block. Internal exceptions are raised implicitly (automatically) by the runtime system; however, user-defined exceptions must be raised explicitly. To handle raised exceptions, you write separate routines called exception handlers.

## Practice 9: Overview

This practice covers the following topics:

- Creating and invoking user-defined exceptions
- Handling named Oracle Server exceptions



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In these practices, you create exception handlers for a predefined exception and a standard Oracle Server exception.

10

# Introducing Stored Procedures and Functions

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Course Road Map

Lesson 1: Course Overview

Unit 1: Introducing PL/SQL

Unit 2: Programming with PL/SQL

**Unit 3: Working with PL/SQL Code**

▶ Lesson 9: Handling Exceptions

▶ Lesson 10: Creating Procedures

You are here!

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In Unit 3, you are introduced to the basic program units in PL/SQL such as stored procedures and functions.

## 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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

- PL/SQL subprograms
- Previewing procedures
- Previewing functions



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# What Are PL/SQL Subprograms?

- A PL/SQL subprogram is a named PL/SQL block that can be called with or without a set of parameters.
- A subprogram consists of a specification and a body.
- A subprogram can be a procedure or a function.
- Typically, you use a procedure to perform an action and a function to compute and return a value.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A PL/SQL subprogram is a named, executable PL/SQL block. It can exist as an independent object on the database. You generally write subprograms to perform an operation or to process data and return a value.

## Subprogram Parts

- A subprogram consists of a specification (spec) and a body.
- To declare a subprogram, you must provide the specifications, which include the name of the subprogram and parameters.
- To define a subprogram, you must provide both the specification and the body.
- You can either declare a subprogram first and define it later in the same block or subprogram, or declare and define it at the same time.

## Subprogram Types

- PL/SQL has two types of subprograms: procedures and functions.
- Typically, you use a procedure to perform an action and a function to compute and return a value.

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 © 2017, Oracle and/or its affiliates. All rights reserved.

## Procedure: Syntax

```
CREATE [OR REPLACE] PROCEDURE procedure_name
[(argument1 [mode1] datatype1,
  argument2 [mode2] datatype2,
  . . .)]
IS|AS
procedure_body;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

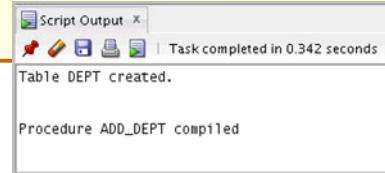
*datatype* Is the data type of the associated parameter. The data type of parameters cannot have explicit size; instead, use %TYPE.

*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: 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 © 2017, Oracle and/or its affiliates. All rights reserved.

In the code example in the slide, 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.

When you compile this script, you see the Script output as shown in the slide. The output shows that a table `DEPT` has been created and that the procedure `ADD_DEPT` is compiled.

When a procedure is compiled, an object is created in the database, which can later be invoked. So here we did not execute the procedure yet.

**Note:** See the following page for more notes on the example.

## Procedure: Example

### Note

- When you create any object, entries are made into 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 * FROM user_objects;
```

OBJECT_NAME	SUBOBJECT_NAME	OBJECT_ID	DATA_OBJECT_ID	OBJECT_TYPE	CREATED	LAST_DDL_TIME	TIMESTAMP
31 SECURE_DML	(null)	73196	(null)	PROCEDURE	07-JUN-16	07-JUN-16	2016-06-07:10:4:
32 SECURE_EMPLOYEES	(null)	73197	(null)	TRIGGER	07-JUN-16	07-JUN-16	2016-06-07:10:4:
33 ADD_JOB_HISTORY	(null)	73198	(null)	PROCEDURE	07-JUN-16	07-JUN-16	2016-06-07:10:4:
34 UPDATE_JOB_HISTORY	(null)	73199	(null)	TRIGGER	07-JUN-16	07-JUN-16	2016-06-07:10:4:
35 RETIRED_EMPS	(null)	73789	73789	TABLE	11-JUL-16	11-JUL-16	2016-07-11:02:20
36 ADD_DEPT	(null)	73831	(null)	PROCEDURE	15-JUL-16	15-JUL-16	2016-07-15:02:1:
37 DEPT	(null)	73830	73830	TABLE	15-JUL-16	15-JUL-16	2016-07-15:02:1:

- 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:

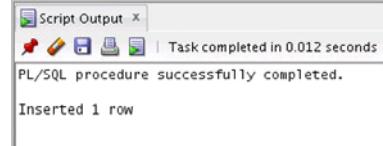
```
SELECT * FROM user_source WHERE name='ADD_DEPT';
```

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;

## Invoking a Procedure

Invoke the procedure in a PL/SQL block

```
...  
BEGIN  
    add_dept;  
END;  
/
```



Run the following query to check the result

```
SELECT department_id,  
       department_name  
  FROM dept  
 WHERE department_id = 280;
```

The image shows a screenshot of an Oracle SQL Developer interface. A red arrow points from the code block above to a window titled "Query Result". The window displays a single row of data from the "dept" table:

DEPARTMENT_ID	DEPARTMENT_NAME
1	280 ST-Curriculum

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 © 2017, Oracle and/or its affiliates. All rights reserved.

## Function: Syntax

```
CREATE [OR REPLACE] FUNCTION function_name
  [(argument1 [mode1] datatype1,
    argument2 [mode2] datatype2,
    . . .)]
  RETURN datatype
  IS|AS
  function_body;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The slide shows the syntax for creating a function. In the syntax:

<i>function_name</i>	Is the name of the function to be created
<i>argument</i>	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.)
<i>mode</i>	Is the type of parameter (Only <code>IN</code> parameters should be declared.)
<i>datatype</i>	Is the data type of the associated parameter
<code>RETURN</code> <i>datatype</i>	Is the data type of the value returned by the function
<i>function_body</i>	Is the PL/SQL block that makes up the function code

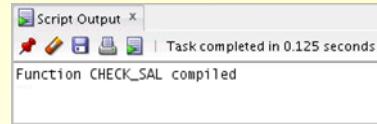
The argument list is optional in a function declaration. The difference between a procedure and a function is that a function must return a value to the calling program. Therefore, the syntax contains `return_type`, which specifies the data type of the value that the function returns. A procedure may 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 © 2017, Oracle and/or its affiliates. All rights reserved.

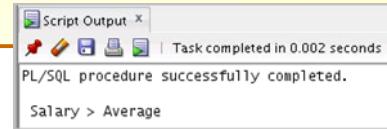
### Function: Example

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 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 © 2017, Oracle and/or its affiliates. All rights reserved.

You include the call to the function in the executable section of the anonymous block. The function is invoked as 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;
```

Task completed in 0.009 seconds			
Argument Name	Type	In/Out	Default
<return value>	PL/SQL BOOLEAN	OUT	unknown

## 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
  ...
END;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 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 *Oracle Database: Develop PL/SQL Program Units* course.

The output of the code example in the slide is as follows:

The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output X'. It displays the following log entries:

- Task completed in 0.054 seconds
- Function CHECK\_SAL dropped.
- Function CHECK\_SAL compiled

## 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;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The code in the slide invokes the function twice by passing parameters. The output of the code is as follows:

The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output'. It displays the message 'PL/SQL procedure successfully completed.' followed by the output of the PL/SQL block. The output includes two lines of text: 'Checking for employee with employee id 205' and 'Salary > Average'. Below that, it shows another two lines: 'Checking for employee with employee id 70' and 'The function returned NULL due to exception'. The window also shows standard toolbar icons for running, stopping, and saving scripts.

```
Script Output X
Task completed in 0.006 seconds
PL/SQL procedure successfully completed.

Checking for employee with employee id 205
Salary > Average
Checking for employee with employee id 70
The function returned NULL due to exception
```

# Quiz



Subprograms:

- a. Are named PL/SQL blocks, and can be invoked by other applications
- b. Are compiled only once
- c. Are stored in the database
- d. Do not have to return values if they are functions
- e. Can take parameters



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

**Answer: a, b, c, e**

## Summary

In this lesson, you should have learned how 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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 10: 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 © 2017, Oracle and/or its affiliates. All rights reserved.



11

# Oracle Cloud Overview

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Lesson Objectives

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

- Describe the salient features of Oracle Cloud
- Discuss the features of Oracle Database Exadata Express Cloud Service



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Lesson Agenda

- Overview of Oracle Cloud
- Working with Oracle Database Exadata Express Cloud Service



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Introduction to Oracle Cloud

- Any business can now use the enterprise cloud provided by Oracle.
- You can access the Oracle Cloud from [cloud.oracle.com](http://cloud.oracle.com).



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The Oracle Cloud is an enterprise cloud for business. Oracle Cloud services are built on Oracle Exalogic Elastic Cloud and Oracle Exadata Database Machine, together offering a platform that delivers extreme performance and scalability.

The top two benefits of cloud computing are speed and cost.

As a result, the applications and databases deployed in the Oracle Cloud are portable and you can easily move them to or from a private cloud or on-premise environment.

- You can request and get the cloud services provisioned through a self-service interface.
- You can either use an integrated development and deployment platform to rapidly extend and create new services.

Using Oracle Cloud services, you can benefit from the following five essential characteristics:

- **On-demand self-service**: You can provision, monitor, and manage cloud on your own.
- **Resource pooling**: You can share resources and maintain a level of abstraction between consumers and services.
- **Rapid elasticity**: You can quickly scale up or down as needed.
- **Measured service**: You pay for what you use with either internal chargeback (private cloud) or external billing (public cloud).
- **Broad network access**: You can access the cloud services through a browser on any networked device.

# Oracle Cloud Services

Oracle Cloud provides three types of services:

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

**SaaS** generally refers to applications that are delivered to end users over the Internet. Oracle CRM On Demand is an example of a SaaS offering that provides both multitenant as well as single-tenant options, depending on the customer's preferences.

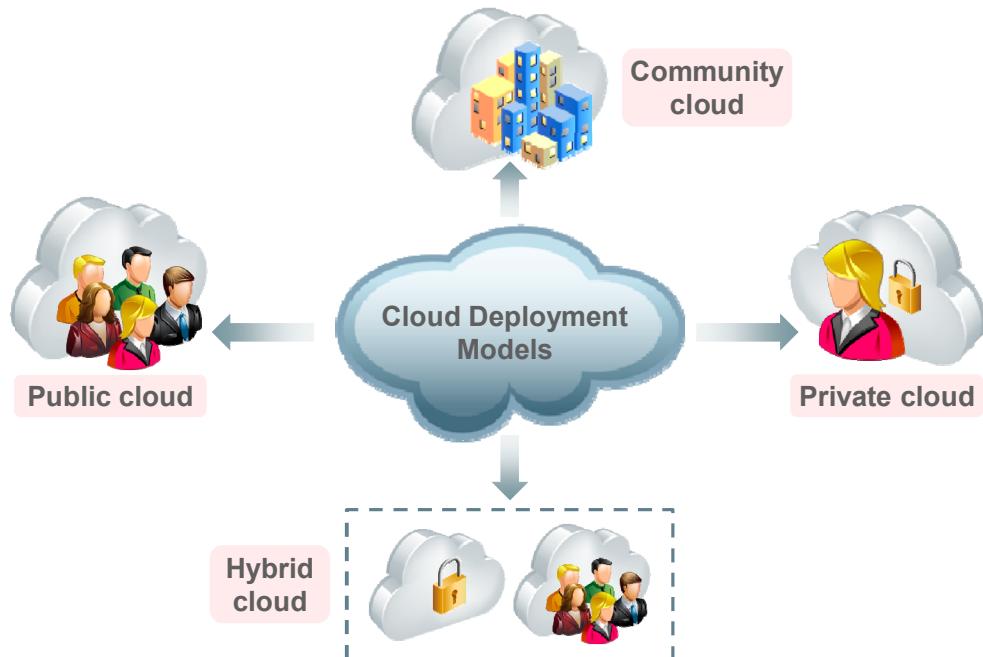
**PaaS** generally refers to an application development and deployment platform that is delivered as a service to developers, enabling them to quickly build and deploy a SaaS application to end users. The platform typically includes databases, middleware, and development tools, all delivered as a service via the Internet.

**IaaS** refers to computing hardware (servers, storage, and network) delivered as a service. This service typically includes the associated software as well as operating systems, virtualization, clustering, and so on. Examples of IaaS in the public cloud include Amazon's Elastic Compute Cloud (EC2) and Simple Storage Service (S3).

The Oracle Cloud Database is built as a PaaS model. It provides on-demand access to database services in a self-service, scalable, and metered manner. You can deploy a database within a virtual machine in an IaaS platform.

You can rapidly deploy Oracle Cloud Database on Oracle Exadata, which is a pre-integrated and optimized hardware platform that supports both online transaction processing (OLTP) and Data Warehouse workloads.

# Cloud Deployment Models



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

- **Private cloud:** A single organization uses a private cloud, which it typically controls, manages, and hosts in private data centers. However, the organization can also outsource hosting and operation to a third-party service provider. Amazon's Virtual Private Cloud is an example of a private cloud in an external provider setting.
- **Public cloud:** Multiple organizations (tenants) use a private cloud on a shared basis. This private cloud is hosted and managed by a third-party service provider. For example: Amazon's Elastic Compute Cloud (EC2), IBM's Blue Cloud, Sun Cloud, and Google AppEngine
- **Community cloud:** A group of related organizations, who want to make use of a common cloud computing environment, uses the community cloud. It is managed by the participating organizations or by a third-party managed service provider. It is hosted internally or externally. For example, a community might consist of the different branches of the military, all the universities in a given region, or all the suppliers to a large manufacturer.
- **Hybrid cloud:** A single organization that wants to adopt both private and public clouds for a single application uses the hybrid cloud. A third model, the hybrid cloud, is maintained by both internal and external providers. For example, an organization might use a public cloud service, such as Amazon Simple Storage Service (Amazon S3), for archived data but continue to maintain in-house (private cloud) storage for operational customer data.

## Lesson Agenda

- Overview of Oracle Cloud
- Working with Oracle Database Exadata Express Cloud Service

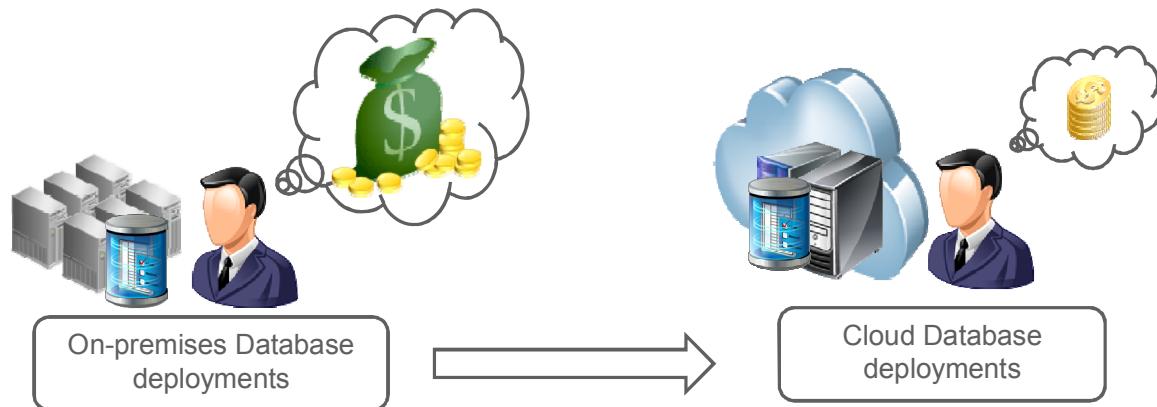


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In this topic, you gain an introduction to Oracle Exadata Express Cloud Service and its features. You will take a tour of its service console and also learn about the different database clients such as Oracle SQL Developer, SQL CL, SQL Workshop and SQL \* Plus that can be used to connect to Oracle Exadata Express Cloud Service.

## Evolving from On-premises to Exadata Express



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Cloud deployments provide end users and enterprises with different capabilities to store and process data. They enable users to have high performance and huge computing resources at a lower price as compared to traditional on-premises deployments.

Exadata Express is a powerful database machine, extended as a cloud service. End users can use it for Oracle 12c database deployments. It delivers a complete database experience for developers and enterprises.

Exadata express being a cloud deployment provides high scalability, performance and availability to its users.

It is fully managed database, therefore you need not worry about patching, upgrading or other DBA tasks.

## What is in Exadata Express?

- A fully managed database service
- Provides powerful yet elastic database cloud service for developers
- Provides on-demand access to a shared pool of database resources
- Comes with built-in tools for rapid application development
  - APEX for web application development
  - Compatibility with clients such as SQL Developer, SQLcl



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Exadata Express is a fully managed database service where end users need not worry about upgrades to the database and other components of the service. All enhancements are automatically managed by the cloud service.

Being a cloud deployment, Exadata Express, allows end users to scale their data virtually to unlimited size.

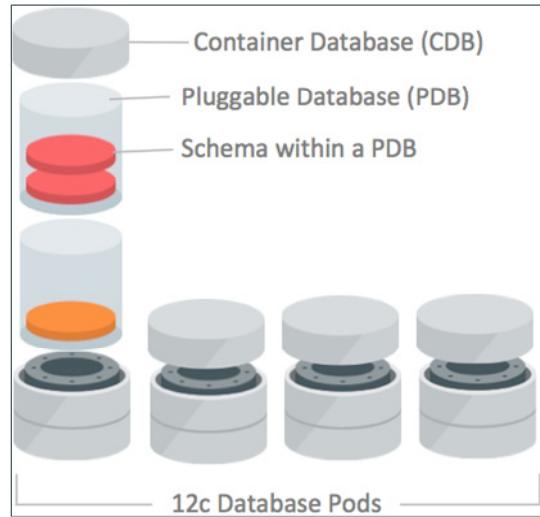
Dynamic provisioning of resources allows users to access huge amount of compute and storage resources in no time.

For developers, it provides built-in application development tool – APEX. APEX(Application Express) is a rapid web application development tool for Oracle database. Developers with minimal development experience can develop and deploy professional applications through web browser using APEX.

Oracle makes a variety of database client drivers and tools available for use with Oracle Database Exadata Express Cloud Service. You can use Exadata Express with Oracle SQL Developer, an IDE used for SQL, PL/SQL development and Oracle SQLcl, an enhanced command line interface.

## Exadata Express for Users

- Oracle manages the service as multiple Container databases(CDBs), also known as database pods
- Each CDB can accommodate upto 1000 Pluggable databases(PDBs).
- Each user is provisioned with a PDB on subscribing to the service, where the user can create several schemas.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Exadata Express is ideal for production applications that consist of small to medium sized data as well as developers, testers, evaluators and other users who are looking for a full Oracle Database experience at an affordable entry-level price. It is a fully managed database service, is organized into Container databases(CDBs). These container databases are also known as database pods.

Each container database in turn can contain several Pluggable databases(PDBs). When a user subscribes to the Exadata Service, a pluggable database is provisioned. Within the PDB, the user can create several schemas. However, PDB Services are constrained by CPU, storage and memory.

# Exadata Express for Developers

- Developers can connect with a wide range of data sources for their applications
  - JSON Document Storage
  - Document Style data access
  - Oracle Rest Data Services



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

**JSON Document Storage** - Oracle Database in Exadata Express provides direct storage, access and management of JSON documents. See JSON Support in Oracle Database New Features Guide 12c Release 2 (12.2).

**Document-Style Data Access** - Oracle Database in Exadata Express gives you the ability to store and access data as schema-less documents and collections using the Simple Oracle Document Access (SODA) API. See Working with JSON and Other Data Using SODA in Using Oracle Database Exadata Express Cloud Service.

**Oracle REST Data Services 3** - Exadata Express includes the newest Oracle REST Data Services (ORDS). With ORDS 3, it's easy to develop modern RESTful interfaces for relational data and now JSON documents stored in Oracle Database.

# Getting Started with Exadata Express

1. Purchase a subscription.
2. Activate and verify the service.
3. Verify activation.
4. Learn about users and roles.
5. Create accounts for your users and assign them appropriate privileges and roles.
6. Set the password for the database user authorized to perform administrative tasks for your service (PDB\_ADMIN).

**Note:** You can refer to Using Oracle Database Exadata Express Cloud Service (<https://docs.oracle.com/cloud/latest/exadataexpress-cloud/CSDBP/toc.htm>) for details on the subscription process.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The steps in the slide provide you a brief overview of how to get started with Exadata Express. In the following slides, you will learn in detail how to perform the aforementioned steps.

# Oracle Exadata Express Cloud Service

You can refer to Working with Oracle Database Exadata Express Cloud Service (<http://oukc.oracle.com/public/redir.html?type=player&offid=1984115860>) to gain an introduction to the service and its features.



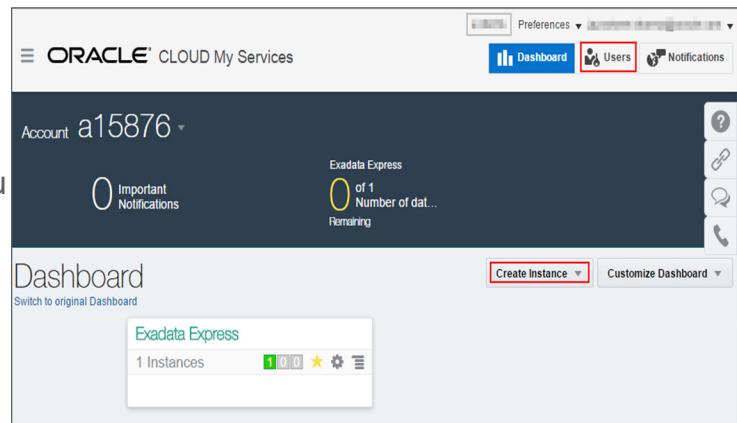
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In this video, you gain an introduction to Oracle Exadata Express Cloud Service and its features. You will take a tour of its service console and also learn about the different database clients such as Oracle SQL Developer, SQL CL, SQL Workshop and SQL \* Plus that can be used to connect to Oracle Exadata Express Cloud Service.

**Note:** This demonstration has audio, which cannot be played inside OU Classroom. However participants can access this link on open internet and review at their convenience.

## Getting Started with Exadata Express

- On signing into the service, you get access to the dashboard.
- Dashboard allows you to create database instances and users.
- The number of instances you create is limited by the amount of resources you have access to.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

After successful subscription to the service, you can login to your account and access the dashboard. Based on the type of subscription you can create instances.

The instances would appear on the dashboard. You can see an instance created in the image . To manage the instance, click on the instance.

# Managing Exadata

The diagram illustrates the navigation from the Service Instances dashboard to the Service Console. On the left, the Service Instances dashboard shows a list of service instances, with one instance named 'exa4' selected. An arrow points from this instance to the right, leading to the 'Service Console: exa4' interface. The Service Console interface is divided into several sections: 'Web Access' (with links to SQL Workshop, REST Data Services, and App Builder), 'Client Access' (with links to Client Credentials, Disable Client Access, and Download Drivers), 'Administration' (with links to Database Schema, Set Administrator Password, and Document Store), and 'Download Tools' (with links to SQL\*Plus and Application Express).



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

On clicking the instance on the dashboard, you see various details about the dashboard. You can access the services by clicking on ‘Open Service Console’.

The service console provides you access to tools for Web Access, Client Access and Administration.

## Service Console

- Service Console is the interface to use and manage the Exadata service
- It provides three different perspectives of the instance
  - Web Access
  - Client Access
  - Administration

The screenshot shows the Service Console interface for an Exadata system named 'exa4'. The interface is divided into three main sections:

- Web Access:** Includes links to 'Go to SQL Workshop' (Run SQL commands, execute SQL scripts and browse database objects), 'Define REST Data Services' (Create and manage RESTful web service interfaces to your database), and 'Install Productivity Apps' (Browse and install productivity apps). It also features a 'Develop with App Builder' section for declarative development.
- Client Access:** Includes links to 'Download Client Credentials' (Download a zip file containing security credentials and network configuration files), 'Disable Client Access' (Disable SQL\*Net access and invalidate all existing client credential files), and 'Download Drivers' (Get database drivers for Java, .NET, Node.js, Python, PHP, Ruby, C, C++, Instant Client and more). It also features a 'Download Tools' section for SQL\*Plus command-line and developer tools.
- Administration:** Includes links to 'Create Database Schema' (Create a new schema for database objects), 'Set Administrator Password' (Set or reset your database's privileged user (PDB\_ADMIN) account password), and 'Create Document Store' (Enable or disable a schema-less document-style interface, with JSON storage and access). It also features a 'Manage Application Express' section for Oracle Application Express administrative options.



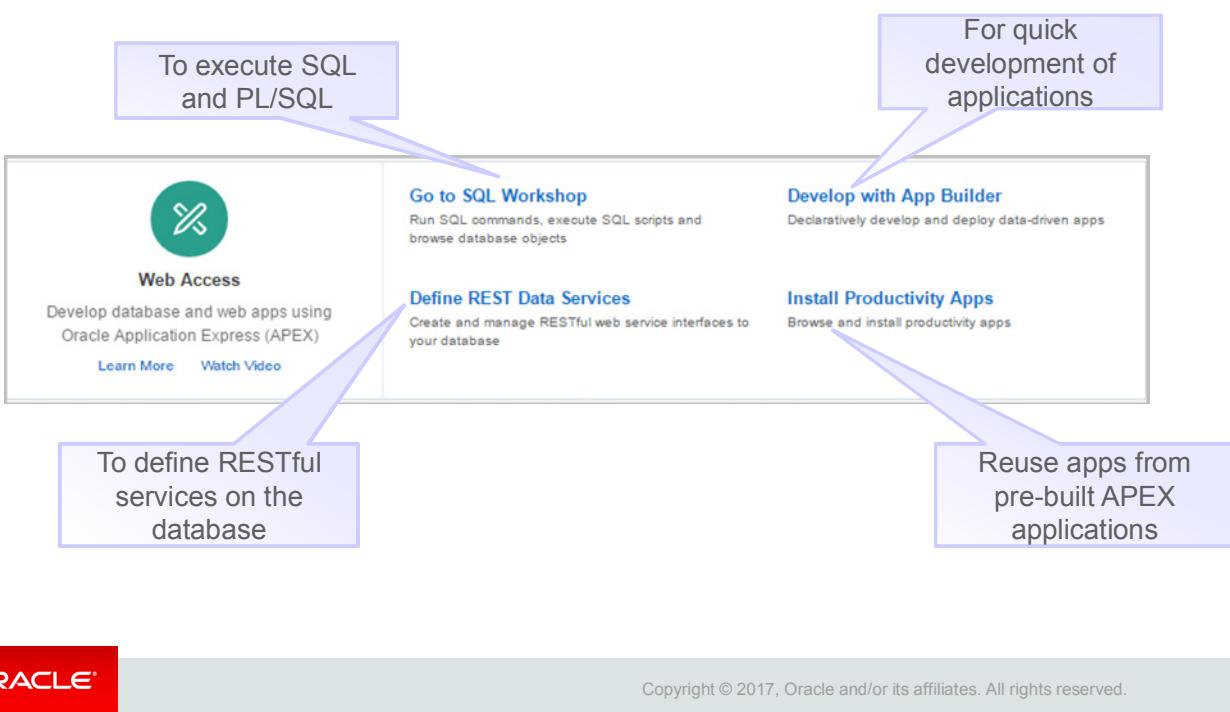
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Web access provides utilities which enable you to develop database and web applications using Oracle Application Express(APEX).

Database clients can connect to Exadata Express service using SQL \*Net Access. Some examples of supported database clients are SQLcl, SQL Developer, SQL \*Plus, JDBC Thin client, ODP.NET, OCI and Instant Client. Client Access in the Service Console allows you to configure with the client you use.

Administration in the Service Console provides for performing administration tasks such as create new database schemas for database objects, set or reset administration password, create a schema-less documents and collections interface, and use administrative options to manage Oracle Application Express.

## Web Access through Service Console

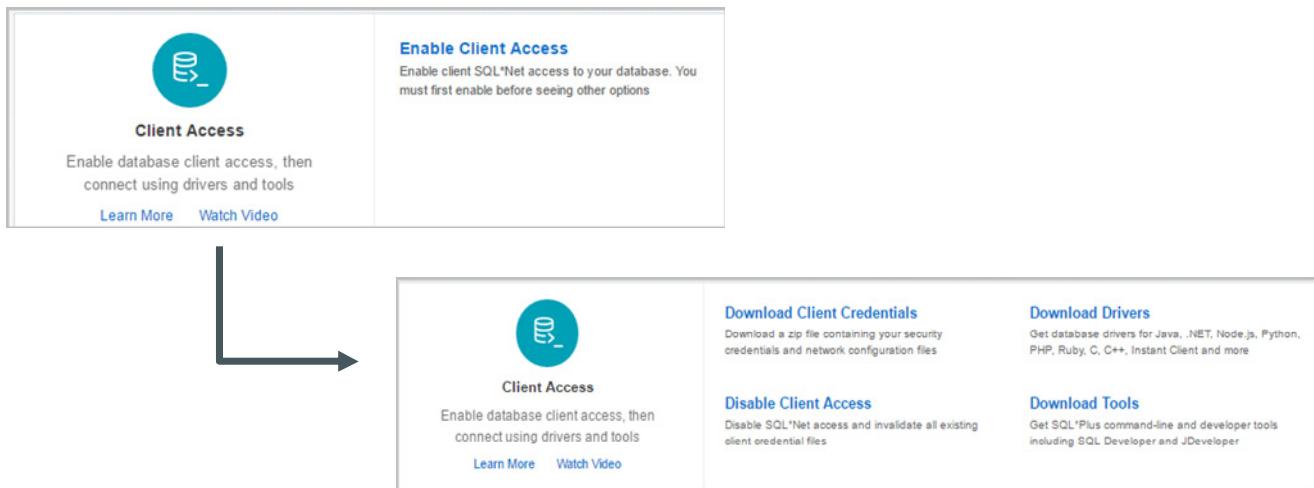


ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Option	Description
Go to SQL Workshop	Allows you go directly to browser-based SQL Workshop, where you can run SQL statements, execute scripts and explore database objects.
Develop with App Builder	Quickly declaratively develop database and websheet applications. You can import files such as database applications and plug-ins. There is a dashboard showing metrics about your applications and workspace utilities to manage defaults, themes, metadata, exports, and more.
Define REST Data Services	Directly access the page to define and manage RESTful web services that view and manipulate data objects within your database.
Install Productivity Apps	Install from a gallery of pre-built Oracle Application Express Productivity Apps.

# Client Access Configuration through Service Console



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You have to enable client access to allow SQL\*Net Access to your service. Using SQL \*Net Access software you can connect the Exadata instance to different clients. This option is only available when client access has not yet been enabled. Once you enable the client access, four options appear in the console:

Option	Description
Download Client Credentials	Download client credentials needed for clients to access your service.
Download Drivers	Go directly to the Oracle Technology Network page to download and install database drivers including for Java, Instant Client, C, C++, Microsoft .NET, Node.js, Python, PHP, Ruby, and more.
Disable Client Access	Use this option to disallow SQL*Net access to your service. This option is only available when client access has been enabled.
Download tools	Go directly to the Oracle Technology Network page to download and install tools such as SQL*Plus, SQLcl, command-line and integrated development environments such as Oracle SQL Developer, JDeveloper, Oracle JET, and more

# Database Administration through Service Console

The screenshot shows the 'Administration' section of the Oracle Service Console. It includes:

- Create Database Schema**: Create a new schema for database objects.
- Set Administrator Password**: Set or reset your database's privileged user (PDB\_ADMIN) account password.
- Create Document Store**: Enable or disable a schema-less document-style interface, with JSON storage and access.
- Manage Application Express**: Use Application Express (APEX) administrative options.

Callout boxes with arrows point to specific sections:

- An arrow points from the text 'Create a new schema for database objects' to the 'Create Database Schema' option.
- An arrow points from the text 'To create a document store using a schema' to the 'Create Document Store' option.
- An arrow points from the text 'Set or reset password for admin' to the 'Set Administrator Password' option.
- An arrow points from the text 'To manage tasks such as archiving APEX schemas and association among APEX schema' to the 'Manage Application Express' option.

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can perform various administration tasks through 'Administration' in the service console.

Option	Description
Create Database Schema	Create a new schema for database objects. Schema is the set of database objects, such as tables and views that belong to that user account.
Create Document Store	This option enables you to create a document store, using either an existing schema or new schema, and to enable SODA for REST, which enables REST-based operations on the schema using Oracle's SODA for REST API. It also enables SODA for Java, which is Oracle's SODA for Java API for use with Java programs.
Set Administrator Password	Use this option to set the password for the PDB_ADMIN database user that is authorized to perform administrative tasks.
Manage Application Express	Options here allow you to enable application archiving to archive your Oracle Application Express applications to database tables, manage the association between schemas and Oracle Application Express, and manage messages and set preferences for the workspace.

# SQL Workshop

Web Access  
Develop database and web apps using Oracle Application Express (APEX)  
Learn More Watch Video

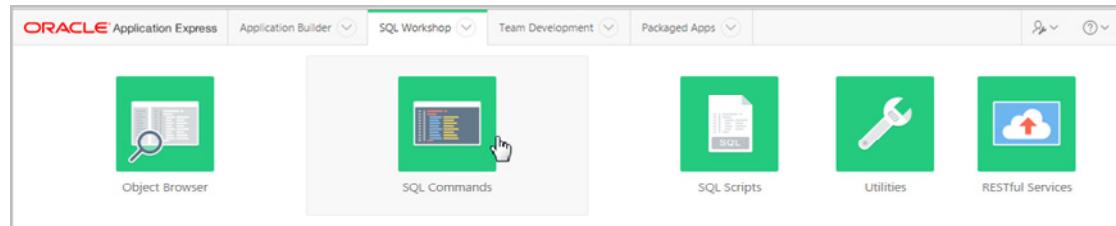
Go to SQL Workshop  
Run SQL commands, execute SQL scripts and browse database objects

Define REST Data Services  
Create and manage RESTful web service interfaces to your database

Develop with App Builder  
Declaratively develop and deploy data-driven apps

Install Productivity Apps  
Browse and install productivity apps

1 Clicking on SQL workshop will lead you to APEX interface



2 To run SQL or PL/SQL you can use the SQL commands utility

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# SQL Workshop

You can run SQL statements in the editor.

The screenshot shows the Oracle Application Express SQL Workshop interface. At the top, there are tabs for Application Builder, SQL Workshop (which is selected), Team Development, and Packaged Apps. Below the tabs, a toolbar includes a magnifying glass icon, a help icon, and a user icon. The main area is titled "SQL Commands" and shows the schema "BZJNMKSSA". A dropdown menu indicates "Rows: 10". Buttons for "Clear Command" and "Find Tables" are present. A "Run" button is highlighted in blue. The SQL command entered is "SELECT \* FROM emp;". The results tab is selected, displaying a grid of data from the EMP table:

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7839	KING	PRESIDENT	-	11/17/1981	5000	-	10
7698	BLAKE	MANAGER	7839	05/01/1981	2850	-	30
7782	CLARK	MANAGER	7839	06/09/1981	2450	-	10
7566	JONES	MANAGER	7839	04/02/1981	2975	-	20
7788	SCOTT	ANALYST	7566	12/09/1982	3000	-	20
7902	FORD	ANALYST	7566	12/03/1981	3000	-	20

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

SQL Workshop in APEX allows you to run SQL statements using SQL commands editor. The results of the SQL query entered will be displayed in the tab below.

## Connecting through Database Clients

You can connect to Exadata Express through various database clients.

Some of the database clients include:

- SQL\*Plus
- SQLcl
- SQL Developer
- .Net and Visual Studio
- JDBC Thin Client



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You must first configure database clients and Oracle Database Exadata Express Cloud Service to communicate with each other.

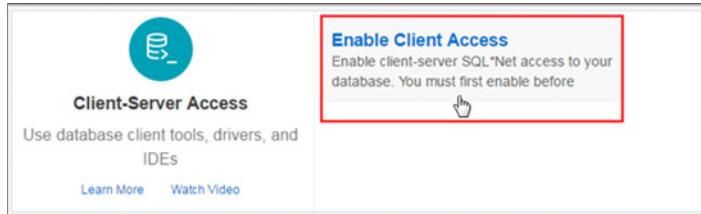
Prerequisite tasks for database client connectivity require you to:

- Enable SQL\*Net access to your service.
- Download client credentials.
- Follow set-up instructions for the specific database client you want to connect with.

In the following topics, you will learn how to enable SQL\*Net access, download client credentials and make connections using Oracle SQL Developer and SQLcl.

# Enabling SQL\*Net Access for Client Applications

Enable SQL\*Net Access in the Service Console to obtain the various Database Client options.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can connect to your Exadata Express cloud service from diverse database clients over SQL\*Net also called as Oracle Database Net Services. Some examples of supported clients include SQL\*Plus, SQLcl, SQL Developer, JDBC Thin, ODP.NET, OCI, and Instant Client.

Database drivers for all popular programming and scripting languages such as Python, PHP, Node.js, C/C++, Ruby and Perl are supported. SQL\*Net access has to be enabled as a prerequisite for all clients and drivers connecting over SQL\*Net.

The Service Administrator must do the following to enable SQL\*Net:

- Navigate to the Service Console for **Exadata Express** and open the service console.
- Click **Enable Client Access**.
- Download the client credentials.
- Now, depending on the client-side application and driver being used, you need to configure the application connection string for that application.

## Downloading Client Credentials

The screenshot illustrates the process of downloading client credentials from the Oracle Cloud Service Console. Step 1 highlights the 'Download Client Credentials' button. Step 2 shows the resulting dialog box where a password is entered and confirmed.

You can easily download the zip files for client credentials from service console. Its contents include Oracle Wallet and Java Keystore as well as essential client configuration files. While downloading the zip file, you are prompted to enter a password.

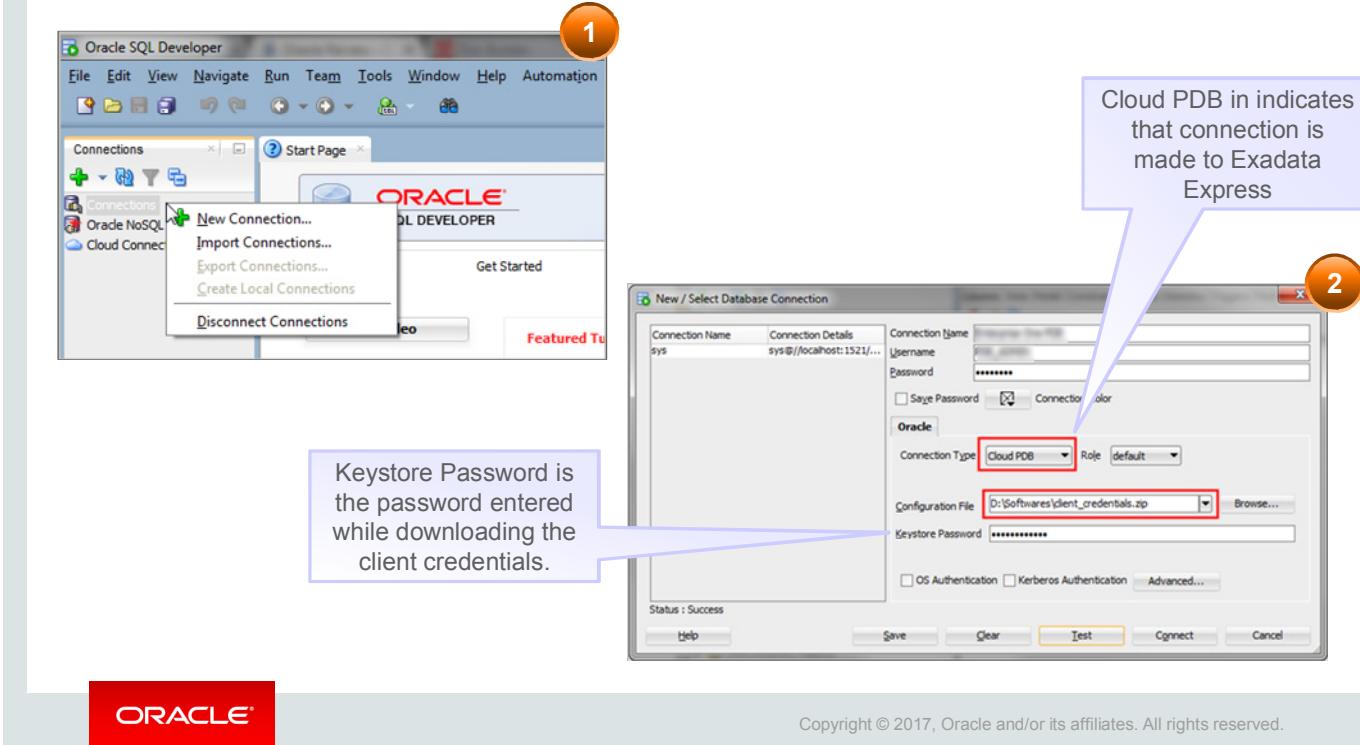
Note that client credential zip files should be carefully managed. Please remember to keep the file secure to avoid unauthorized database access. If you believe the security of this file has been compromised, then immediately disable client access using the cloud service console.

The steps to download the client credentials are as follows.

1. Navigate to Exadata Express and open the service console.
2. Click Download Client Credentials to download a zip file containing your security credentials and network configuration files.
3. Enter a password to create a password-protected Oracle Wallet and Java Keystore files for the service.
4. Click Download and save the downloaded zip file to a secure location that is accessible by your database client(s).

In the following topics, you will learn how to use these credentials to connect to the cloud database.

# Connecting Oracle SQL Developer



In order to connect to Oracle Database Exadata Express cloud service, you need to download and install Oracle SQL Developer Release 4.1.5 or later. You should have also downloaded the Client Credentials from Oracle Exadata Express service console. You should then configure an Oracle Cloud connection in the Oracle SQL Developer.

The connection can be created as follows:

1. Run Oracle SQL Developer locally.
2. Under Connections, right click Connections and select **New Connection**.
3. Enter the following details:
  - Connection Name – Enter a name for this cloud connection.
  - Username – Enter username required to sign into Exadata Express.
  - Password – Enter password required to sign into Exadata Express.
  - Connection type – Select **Cloud PDB**.
  - Configuration File - Click Browse and select the **Client Credentials** zip file that you previously downloaded from the Exadata Express service console.
  - Keystore Password – Enter the password provided while downloading the Client Credentials from the Exadata Express service console.
4. Click Test. If the status is Success, click Connect.

If you have connected successfully, the tables and other objects from Exadata Express display under the new connection.

## Connecting Oracle SQLcl

```
D:\PDB Service\SQL CL\sqlcl-no-jre-latest\sqlcl\bin>sql /nolog
SQLcl: Release 4.2.0.16.160.2007 RC on Thu Sep 08 12:18:07 2016
Copyright (c) 1982, 2016, Oracle. All rights reserved.
SQL>
```

1

```
SQL> set cloudconfig client_credentials.zip
Wallet Password: *****
Using temp directory:C:\Users\APOTHU~1.ORA\AppData\Local\Temp\
oracle_cloud_config6707346342028726502
```

2

```
SQL> conn pdb_admin/welcome1@dbaccess
Connected.
SQL>
```

3



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use Oracle SQLcl which is a powerful command-line interface(CLI) to connect to the cloud database. In order to connect, you need to first download and setup Oracle SQLcl locally. You should have also downloaded the Client Credentials from Oracle Exadata Express service console.

To create an Oracle SQLcl cloud connection:

- Navigate to the sqlcl/bin directory from where you unzipped the SQLcl installation files, and run sql /nolog, to startup Oracle SQLcl. The Oracle SQLcl starts displaying the date and time, the SQLcl version and copyright information, before the SQLcl prompt appears.
- At the SQLcl prompt, type set cloudconfig <name of your wallet zip file>, and press the Enter key.
- Enter the Password provided for downloading the Client Credentials from the Exadata Express service console, and press the Enter key.
- To connect to the Exadata Express, type conn <username>/<password>@<servicename>, and press the Enter key. The username and password are the credentials of your database account.

You should now be connected to Exadata Express.

## Summary

In this lesson, you should have learned about:

- The salient features of Oracle Cloud
- Oracle Database Exadata Express Cloud Service



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Relational database management systems are composed of objects or relations. They are managed by operations and governed by data integrity constraints.

Oracle Corporation produces products and services to meet your RDBMS needs. The main products are the following:

- Oracle Database, which you use to store and manage information by using SQL
- Oracle Fusion Middleware, which you use to develop, deploy, and manage modular business services that can be integrated and reused
- Oracle Enterprise Manager Grid Control, which you use to manage and automate administrative tasks across sets of systems in a grid environment

## SQL

The Oracle server supports ANSI-standard SQL and contains extensions. SQL is the language that is used to communicate with the server to access, manipulate, and control data.



A

# Using SQL Developer

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## 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
- Browse the Data Modeling options in SQL Developer



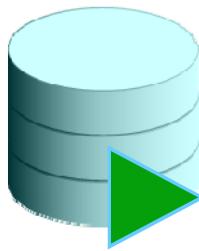
ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 debug stored procedures, test SQL statements, and view optimizer plans.

SQL Developer, which is 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.

SQL Developer is the interface to administer the Oracle Application Express Listener. The new interface enables you to specify global settings and multiple database settings with different database connections for the Application Express Listener. SQL Developer provides the option to drag and drop objects by table or column name onto the worksheet. It provides improved DB Diff comparison options, GRANT statements support in the SQL editor, and DB Doc reporting. Additionally, SQL Developer includes support for 12c Database features.

## Specifications of SQL Developer

- Is shipped along with Oracle Database 12c Release 1
- Is developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Enables default connectivity using the JDBC Thin driver
- Connects to Oracle Database version 9.2.0.1 and later



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

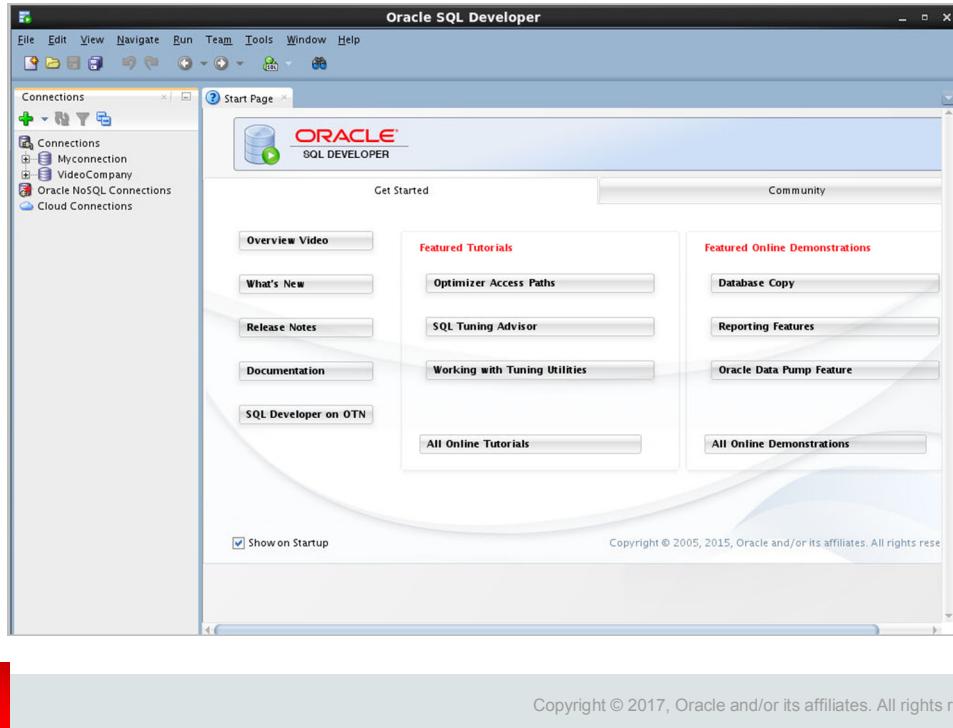
Oracle SQL Developer is shipped along with Oracle Database 12c Release 1 by default. SQL Developer is developed in Java, leveraging the Oracle JDeveloper integrated development environment (IDE). Therefore, it is a cross-platform tool. The tool runs on Windows, Linux, and Mac operating system (OS) X platforms.

The 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 and you need to simply 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 12c Release 1, you will have to download and install SQL Developer. SQL Developer can be downloaded for free from the following link:  
<http://www.oracle.com/technetwork/developer-tools/sql-developer/downloads/index.html>
- For instructions on how to install SQL Developer, see the website at:  
<http://www.oracle.com/technetwork/developer-tools/sql-developer/overview/index.html>

# SQL Developer 4.1.3 Interface



The SQL Developer 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.
- **Reports tab:** Identified by the Reports icon, this tab enables you to run predefined reports or create and add your own reports.
- **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.

## 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.

## Menus

The following menus contain standard entries, plus 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 for executing subprograms
- **Run:** Contains the Run File and Execution Profile options that are relevant when a function or procedure is selected, and also debugging options
- **Versioning:** Provides integrated support for the following versioning and source control systems – Concurrent Versions System (CVS) and Subversion
- **Tools:** Invokes SQL Developer tools such as SQL\*Plus, Preferences, and SQL Worksheet. It also contains options related to migrating third-party databases to Oracle.

**Note:** The Run menu also contains options that are relevant when a function or procedure is selected for debugging.

## 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.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

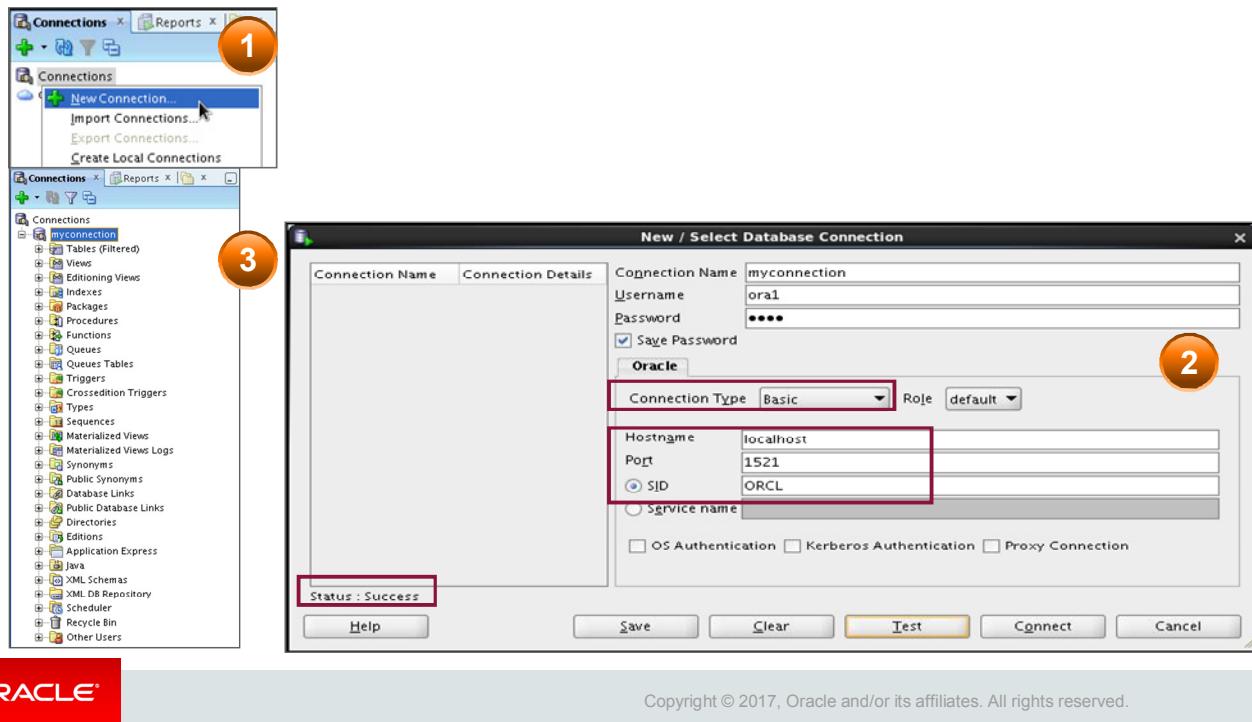
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory, but it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and open the Database Connections dialog box, SQL Developer automatically 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.

You can create additional connections as different users to the same database or to connect to the different databases.

# Creating a Database Connection



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 list, 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 host name and SID for the database that you want to connect to. Port is already set to 1521. 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.

**Local/Bequeath:** If the client and database exist on the same computer, a client connection can be passed directly to a dedicated server process without going through the listener.

- c. Click Test to ensure that the connection has been set correctly.
- d. Click Connect.

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.

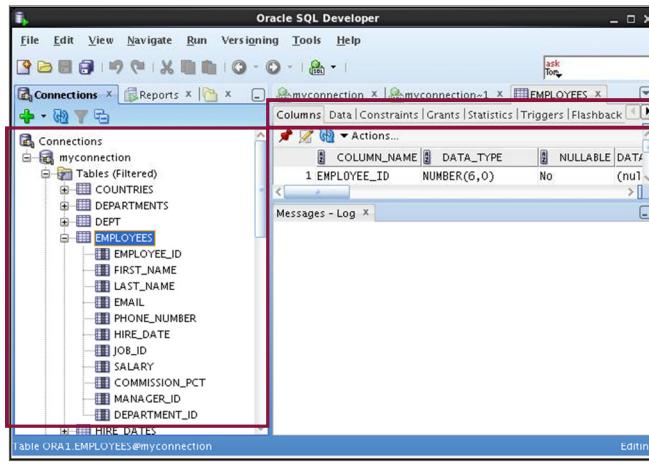
3. The connection gets added in the Connections Navigator. You can expand the connection to view the database objects and view object definitions—for example, dependencies, details, statistics, and so on.

**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.

## 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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.

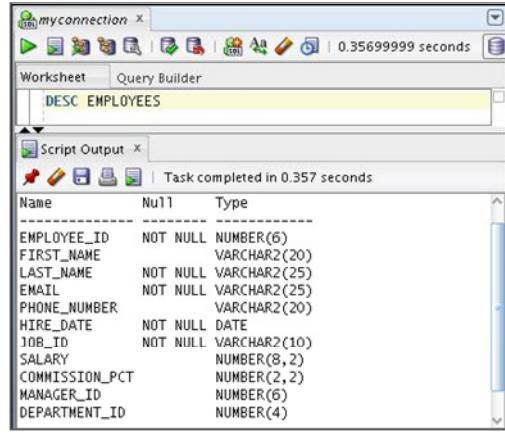
You can see the definition of the objects broken into tabs of information that is pulled out of the data dictionary. For example, if you select a table in the Navigator, details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

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:



The screenshot shows the Oracle SQL Developer interface. In the top-left corner, there's a red button with the word "ORACLE". The main window has a title bar "myconnection X" and tabs "Worksheet" and "Query Builder". The "Worksheet" tab is active, showing the command "DESC EMPLOYEES" in a yellow-highlighted input field. Below it, a "Script Output" window displays the results of the command. The output is a table with three columns: "Name", "Null", and "Type". The data rows are as follows:

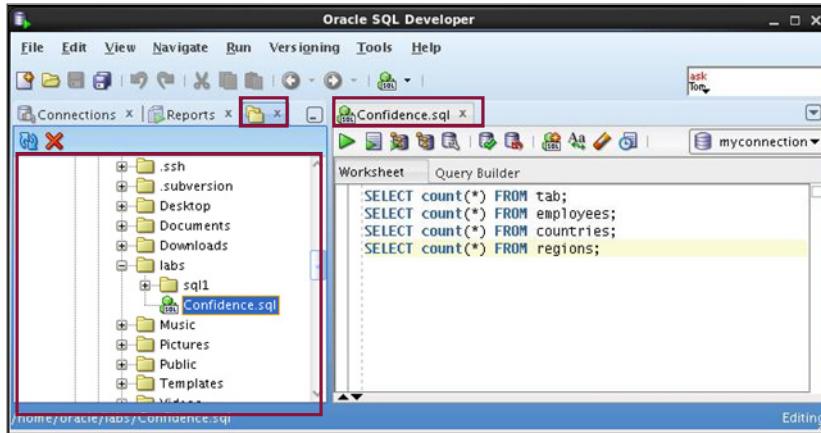
Name	Null	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 if a column must contain data.

## Browsing Files

Use the File Navigator to explore the file system and open system files.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

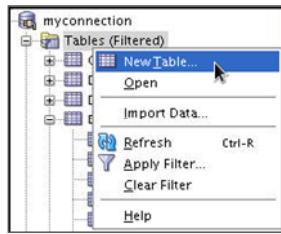
## Browsing Database Objects

You can use the File Navigator to browse and open system files.

- To view the File Navigator, click the View tab and select Files, 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.

## 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®

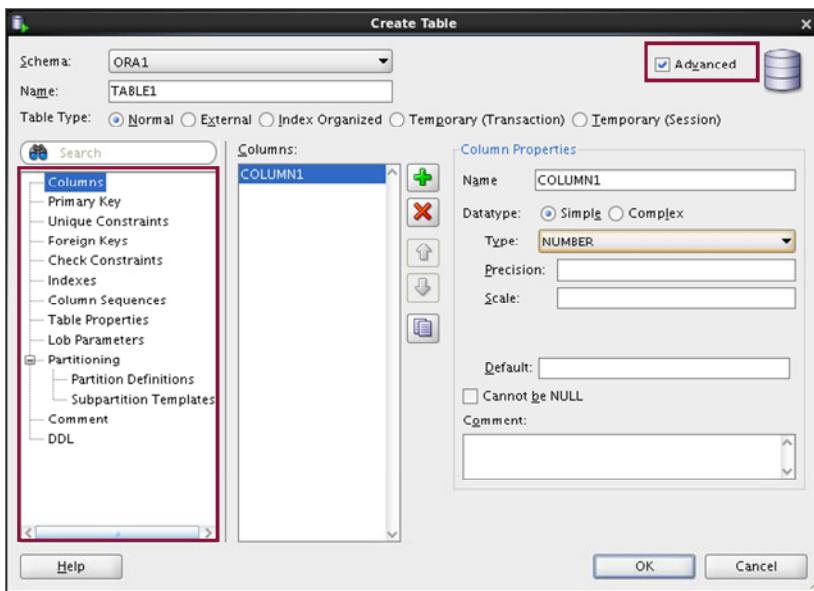
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 menus. When created, you can edit objects 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 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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the Create Table dialog box, if you do not select the Advanced check box, you can create a table quickly by specifying 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.

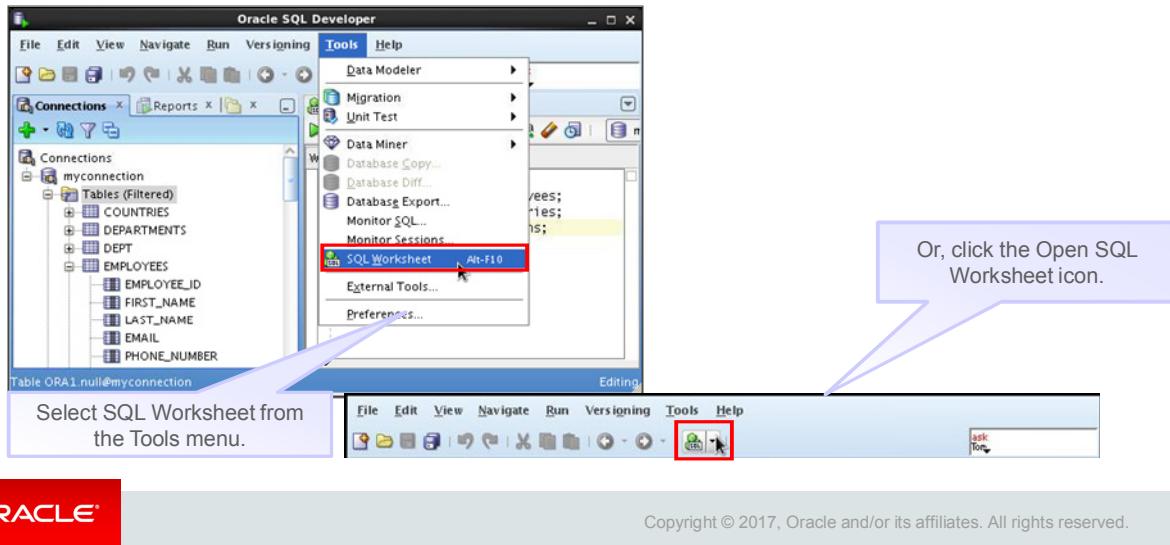
To create a new table, perform the following steps:

1. In the Connections Navigator, right-click Tables and select Create TABLE.
2. In the Create Table dialog box, select Advanced.
3. Specify the column information.
4. 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 have created; to do so, right-click the table in the Connections Navigator and select Edit.

## 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.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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. SQL\*Plus statements that are not supported by the SQL Worksheet are ignored and not passed to the database.

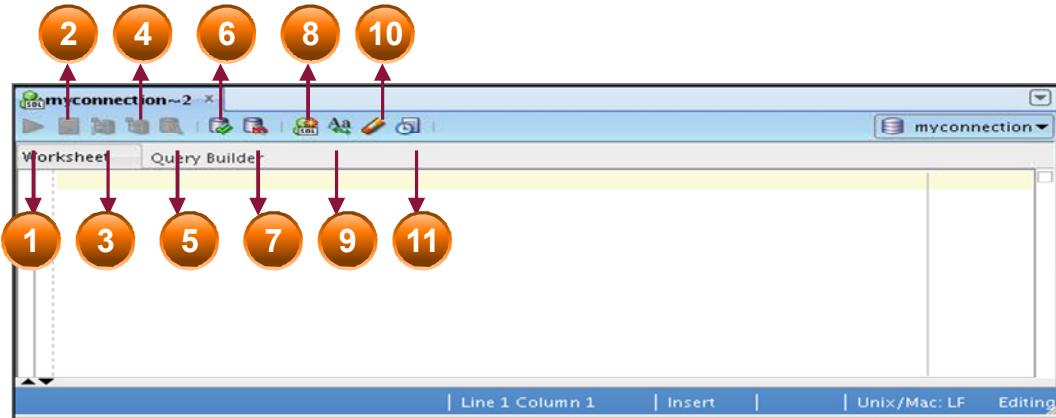
You can specify the actions that can be processed by the database connection associated with the worksheet, such as:

- 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.

## Using the SQL Worksheet



ORACLE®

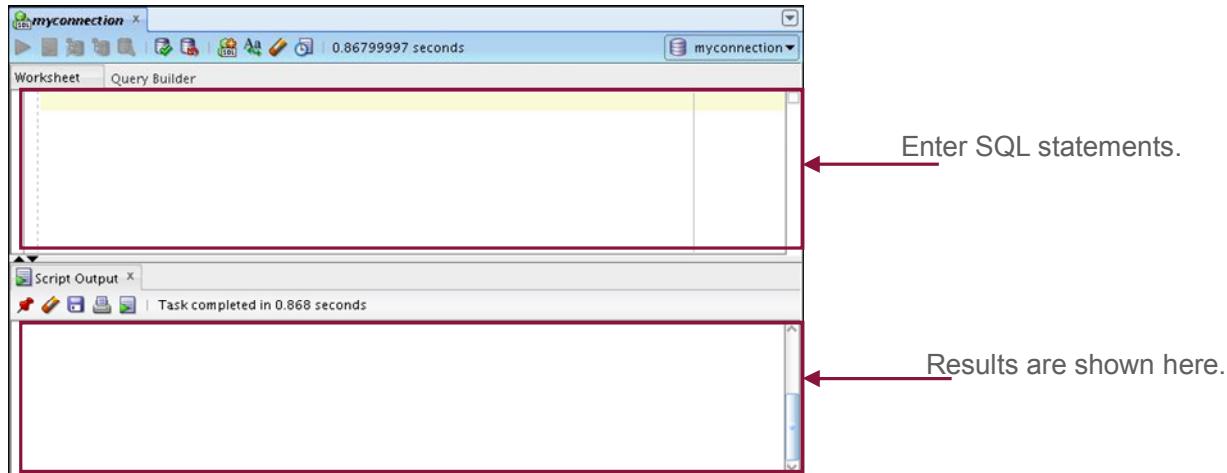
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 the SQL statements that you have executed. You can use the SQL Worksheet toolbar that contains icons to perform the following tasks:

1. **Run Statement:** Executes the statement where the cursor is located in the Enter SQL Statement box. You can use bind variables in the SQL statements, but not substitution variables.
2. **Run Script:** Executes all the statements in the Enter SQL Statement box by using the Script Runner. You can use substitution variables in the SQL statements, but not bind variables.
3. **Autotrace:** Generates trace information for the statement
4. **Explain Plan:** Generates the execution plan, which you can see by clicking the Explain tab
5. **SQL Tuning Advisory:** Analyzes high-volume SQL statements and offers tuning recommendations
6. **Commit:** Writes any changes to the database and ends the transaction
7. **Rollback:** Discards any changes to the database, without writing them to the database, and ends the transaction

8. **Unshared SQL Worksheet:** Creates a separate unshared SQL Worksheet for a connection
9. **To Upper/Lower/InitCap:** Changes the selected text to uppercase, lowercase, or initcap, respectively
10. **Clear:** Erases the statement or statements in the Enter SQL Statement box
11. **SQL History:** Displays a dialog box with information about the SQL statements that you have executed

## Using the SQL Worksheet



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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. The SQL\*Plus commands that are used in SQL Developer must be interpreted by the SQL Worksheet before being passed to the database.

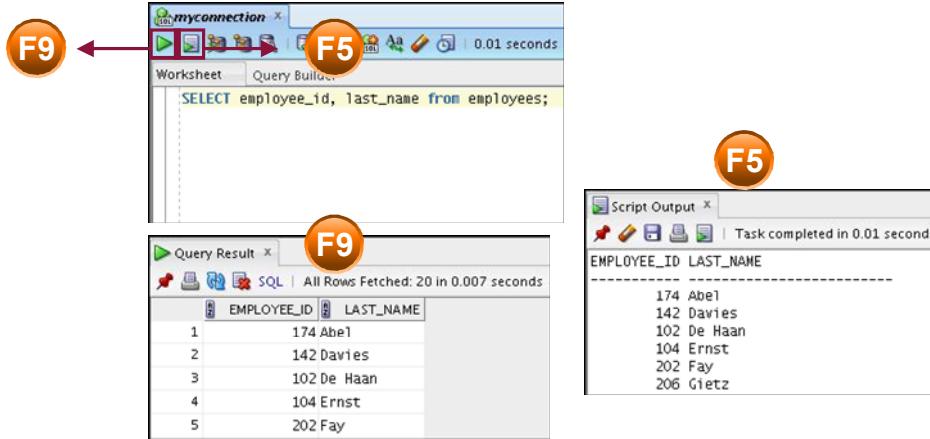
The SQL Worksheet currently supports a number of SQL\*Plus commands. Commands that are not supported by the SQL Worksheet are ignored and not sent to the Oracle database. Through the SQL Worksheet, you can execute the SQL statements and some of the SQL\*Plus commands.

You can display a SQL Worksheet by using any of the following options:

- Select Tools > SQL Worksheet.
- Click the Open SQL Worksheet icon.

# Executing SQL Statements

Use the Enter SQL Statement box to enter single or multiple SQL statements.

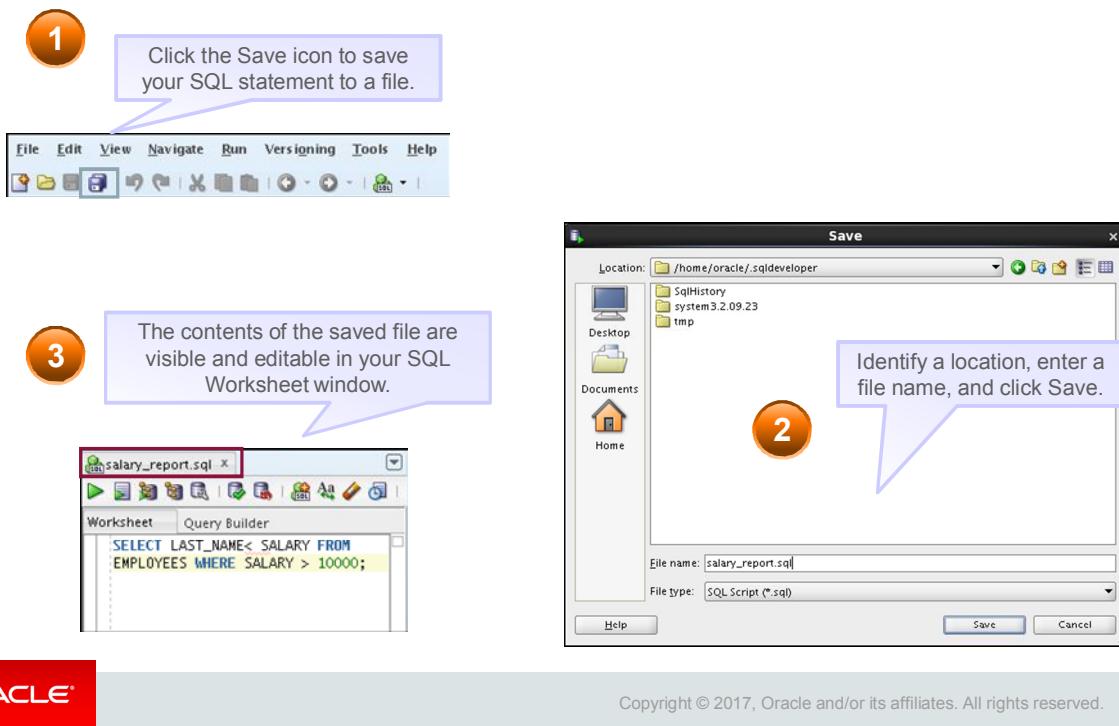


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide shows the difference in output for the same query when the F9 key or Execute Statement is used versus the output when F5 or Run Script is used.

# Saving SQL Scripts



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can save your SQL statements from the SQL Worksheet to a text file. To save the contents of the Enter SQL Statement box, perform the following steps:

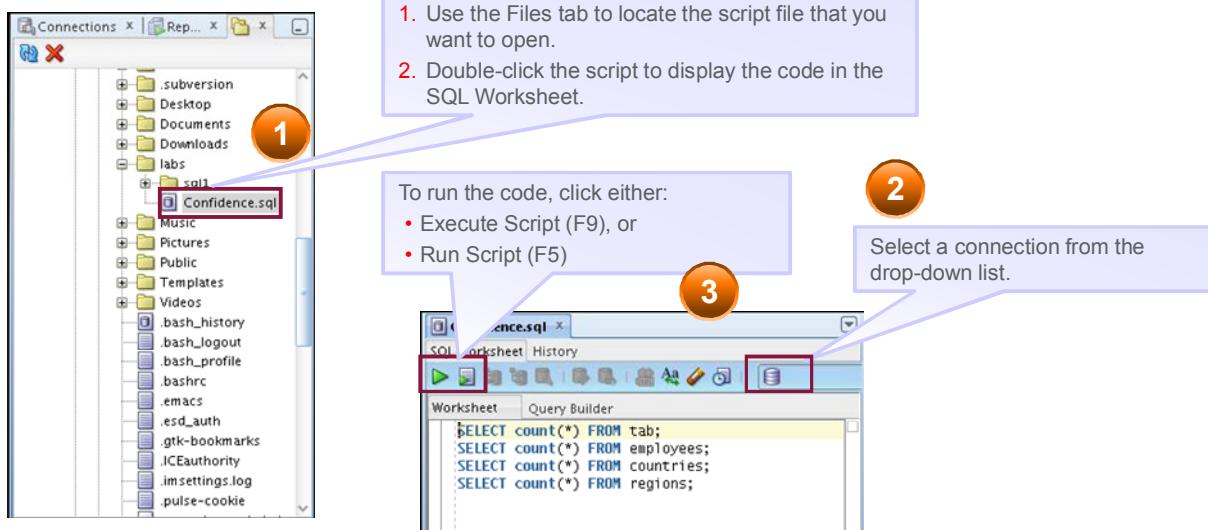
1. Click the Save icon or use the File > Save menu item.
2. In the Save dialog box, enter a file name and the location where you want the file saved.
3. Click Save.

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 is displayed 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.

## Executing Saved Script Files: Method 1



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

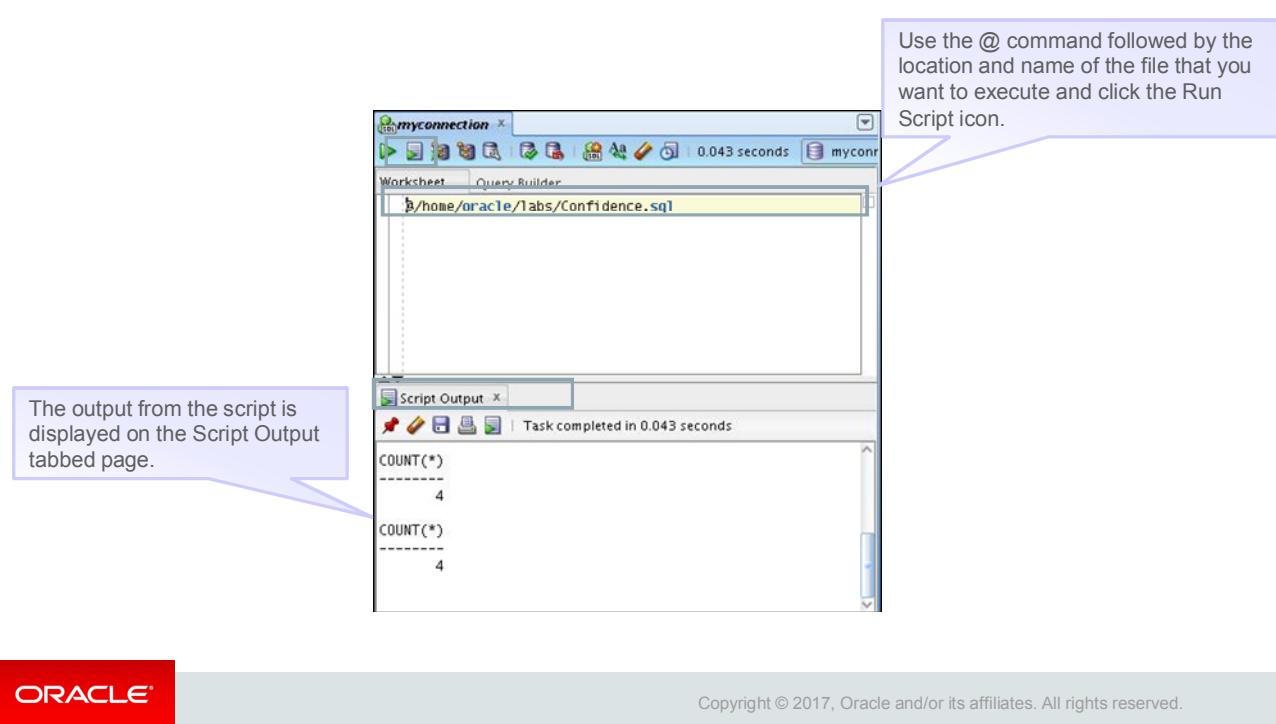
To open a script file and display the code in the SQL Worksheet area, perform the following steps:

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 do the following:

1. Select File > Open. The Open dialog box is displayed.
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.

## Executing Saved Script Files: Method 2



ORACLE®

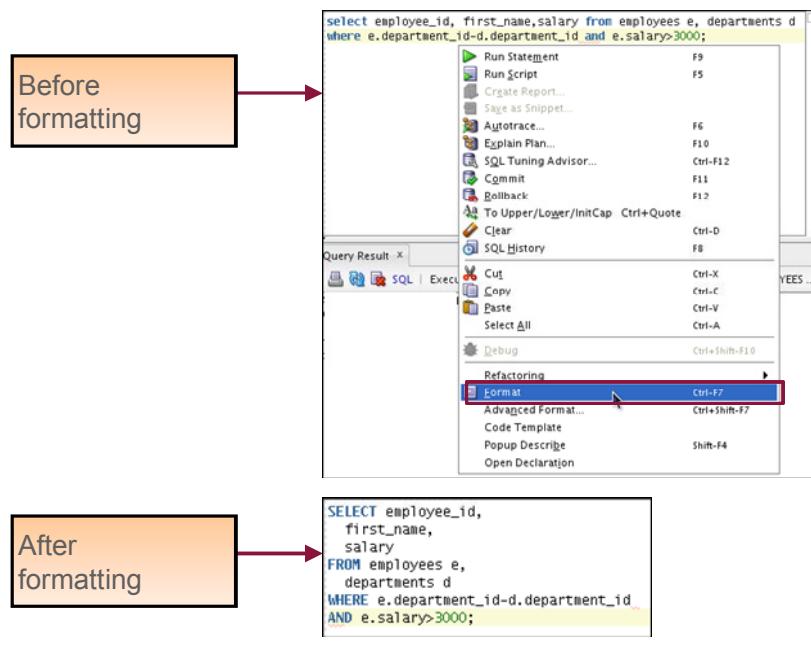
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To run a saved SQL script, perform the following steps:

1. Use the @ command followed by the location and the 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 File Save dialog box appears and you can identify a name and location for your file.

## Formatting the SQL Code



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

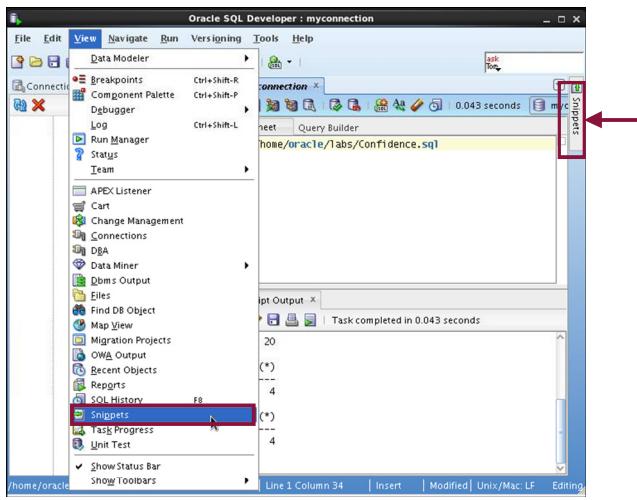
You may want to format the indentation, spacing, capitalization, and line separation of the SQL code. SQL Developer has a feature for formatting SQL code.

To format the SQL code, right-click in the statement area and select Format.

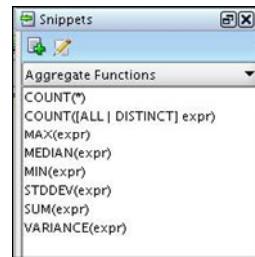
In the example in the slide, before formatting, the SQL code has the keywords not capitalized and the statement not properly indented. After formatting, the SQL code is beautified with the keywords capitalized and the statement properly indented.

# Using Snippets

Snippets are code fragments that may be just syntax or examples.



When you place your cursor here, it shows the Snippets window. From the drop-down list, you can select the functions category that you want.



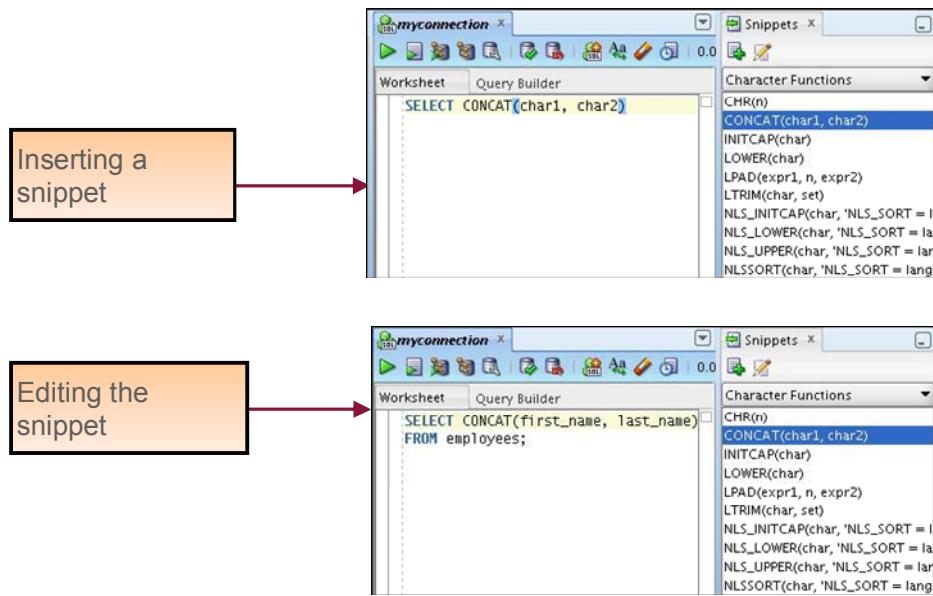
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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. Snippets are code fragments such as SQL functions, optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag snippets to the Editor window.

To display Snippets, select View > Snippets.

The Snippets window is displayed on 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.

## Using Snippets: Example



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

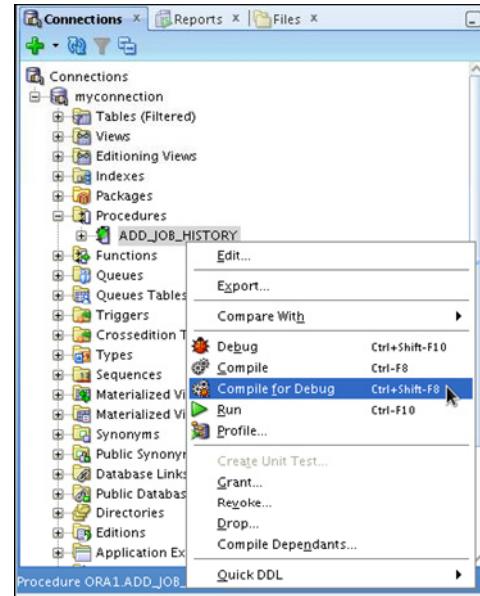
To insert a Snippet into your code in a SQL Worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window to the desired place in your code. Then you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function 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 in the following:

```
SELECT CONCAT(first_name, last_name)
FROM employees;
```

# 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 the Debug menu options to set breakpoints, and to perform step into, step over tasks.



**ORACLE**

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 tab of the output window.

# Database Reporting

SQL Developer provides a number of predefined reports about the database and its objects.

Owner	Name	Type	Referenced_Owner	Referenced_Name	Referenced_Type
APEX_040100 APEX	PROCEDURE APEX_040100		MVV_FLOW		PACKAGE
APEX_040100 APEX	PROCEDURE APEX_040100		MVV_FLOW_ISC		PACKAGE
APEX_040100 APEX	PROCEDURE APEX_040100		MVV_FLOW_SECURITY		PACKAGE
APEX_040100 APEX	PROCEDURE SYS		STANDARD		PACKAGE
APEX_040100 APEX	PROCEDURE SYS		SYS_STUB_FOR_PURITY_ANALYSIS		PACKAGE
APEX_040100 APEXWS	PACKAGE SYS		STANDARD		PACKAGE
APEX_040100 APEXADMIN	PROCEDURE APEX_040100	F			PROCEDURE
APEX_040100 APEXADMIN	PROCEDURE SYS		STANDARD		PACKAGE
APEX_040100 APEXADMIN	PROCEDURE SYS		SYS_STUB_FOR_PURITY_ANALYSIS		PACKAGE
APEX_040100 APEX_APPLICATIONS	VIEW APEX_040100	NV			FUNCTION
APEX_040100 APEX_APPLICATIONS	VIEW APEX_040100		MVV_FLOWS		TABLE
APEX_040100 APEX_APPLICATIONS	VIEW APEX_040100		MVV_FLOW_APPLICATION_GROUPS		TABLE
APEX_040100 APEX_APPLICATIONS	VIEW APEX_040100		MVV_FLOW_AUTHENTICATIONS		TABLE
APEX_040100 APEX_APPLICATIONS	VIEW APEX_040100		MVV_FLOW_COMPANIES		TABLE
APEX_040100 APEX_APPLICATIONS	VIEW APEX_040100		MVV_FLOW_COMPANY_SCHEMAS		TABLE
APEX_040100 APEX_APPLICATIONS	VIEW APEX_040100		MVV_FLOW_COMPUTATIONS		TABLE
APEX_040100 APEX_APPLICATIONS	VIEW APEX_040100		MVV_FLOW_ICON_BAR		TABLE
APEX_040100 APEX_APPLICATIONS	VIEW APEX_040100		MVV_FLOW_INSTALL_SCRIPTS		TABLE
APEX_040100 APEX_APPLICATIONS	VIEW APEX_040100		MVV_FLOW_ITEMS		TABLE
APEX_040100 APEX_APPLICATIONS	VIEW APEX_040100		MVV_FLOW_LANGUAGE_MAP		TABLE

ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

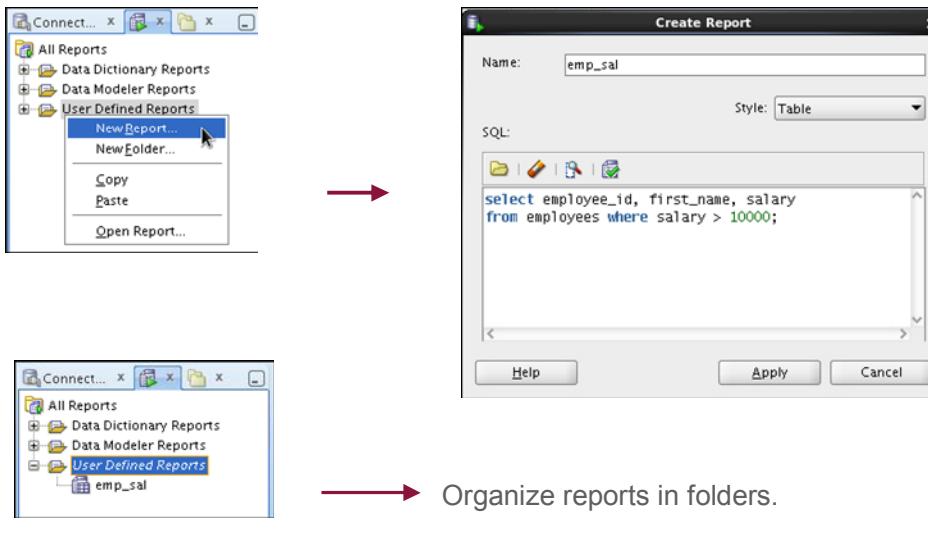
SQL Developer provides many reports about the database and its objects. These reports can be grouped into the following categories:

- 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 on the left of the window. Individual reports are displayed in tabbed panes on the right of the window; and 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 the Owner. You can also create your own user-defined reports.

# Creating a User-Defined Report

Create and save user-defined reports for repeated use.



**ORACLE**

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

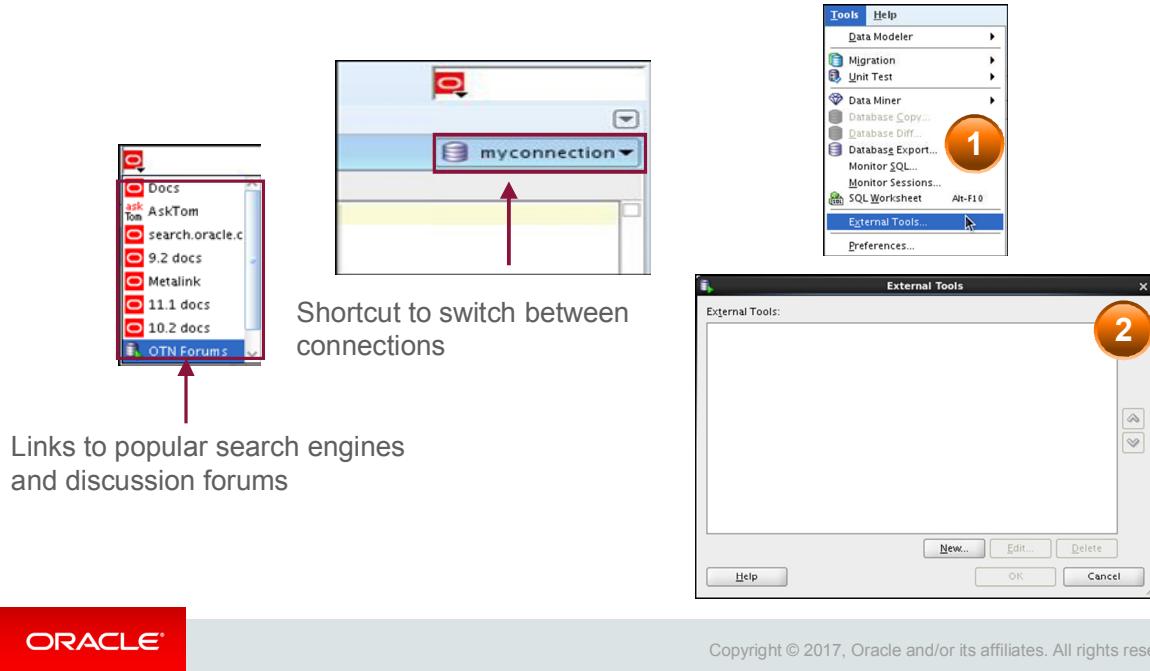
User-defined reports are reports created by SQL Developer users. To create a user-defined report, perform the following steps:

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 stays briefly 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` in the directory for user-specific information.

# Search Engines and External Tools



ORACLE

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

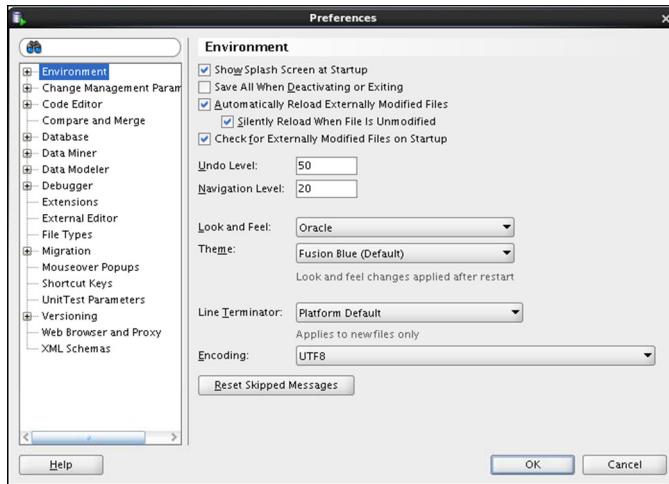
To enhance the productivity of developers, SQL Developer has added quick links to popular search engines and discussion forums such as AskTom, Google, and so on. Also, you have shortcut icons to some of the frequently-used tools such as Notepad, Microsoft Word, and Dreamweaver, available to you.

You can add external tools to the existing list or even delete shortcuts to the tools that you do not use frequently. To do so, perform the following steps:

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.
- In the Tools menu, select Preferences.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

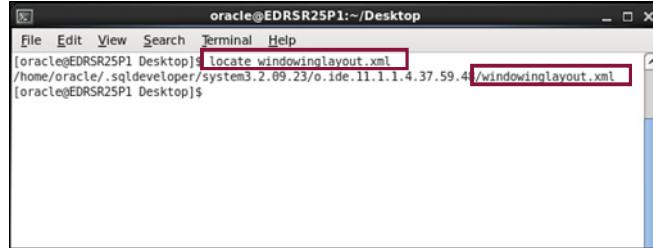
You can customize many aspects of the SQL Developer interface and environment by modifying SQL Developer preferences according to your needs. To modify SQL Developer preferences, select Tools, and then Preferences.

The preferences are grouped into the following categories:

- Environment
- Change Management parameter
- Code Editors
- Compare and Merge
- Database
- Data Miner
- Data Modeler
- Debugger
- Extensions
- External Editor
- File Types
- Migration

- Mouseover Popups
- Shortcut Keys
- Unit Test Parameters
- Versioning
- Web Browser and Proxy
- XML Schemas

## Resetting the SQL Developer Layout



The Oracle logo, which consists of the word "ORACLE" in a white sans-serif font inside a red rectangular box.

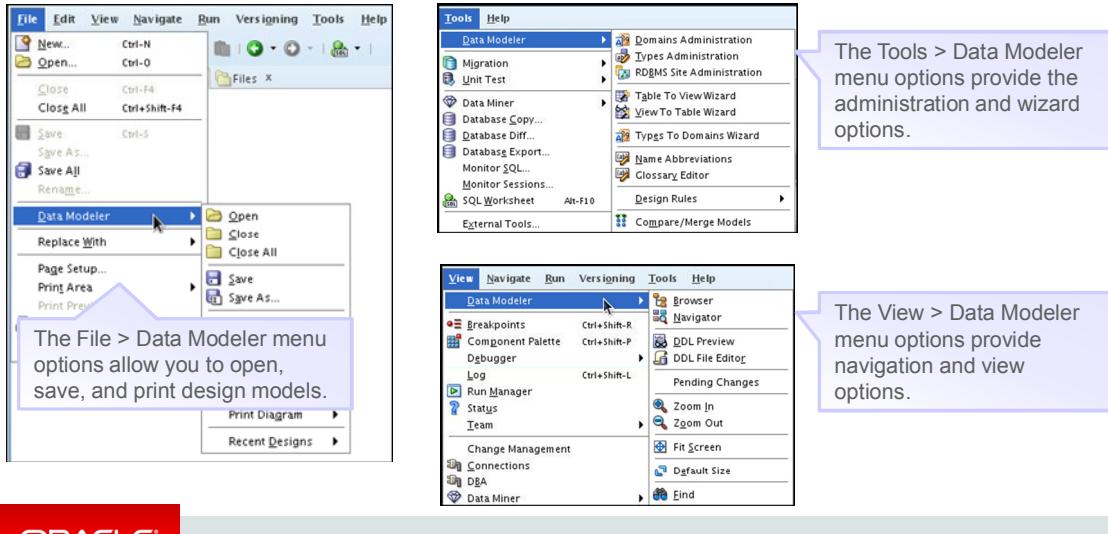
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 from 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.

# Data Modeler in SQL Developer

SQL Developer includes an integrated version of SQL Developer Data Modeler.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Using the integrated version of the SQL Developer Data Modeler, you can:

- Create, open, import, and save a database design
- Create, modify, and delete Data Modeler objects

To display Data Modeler in a pane, click Tools, and then Data Modeler. The Data Modeler menu under Tools includes additional commands, for example, that enable you to specify design rules and preferences.

## 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
- Browse the Data Modeling options in SQL Developer



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.



B

# Using SQL\*Plus

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Objectives

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

- Log in to SQL\*Plus
- Edit SQL commands
- Format the output by using SQL\*Plus commands
- Interact with script files

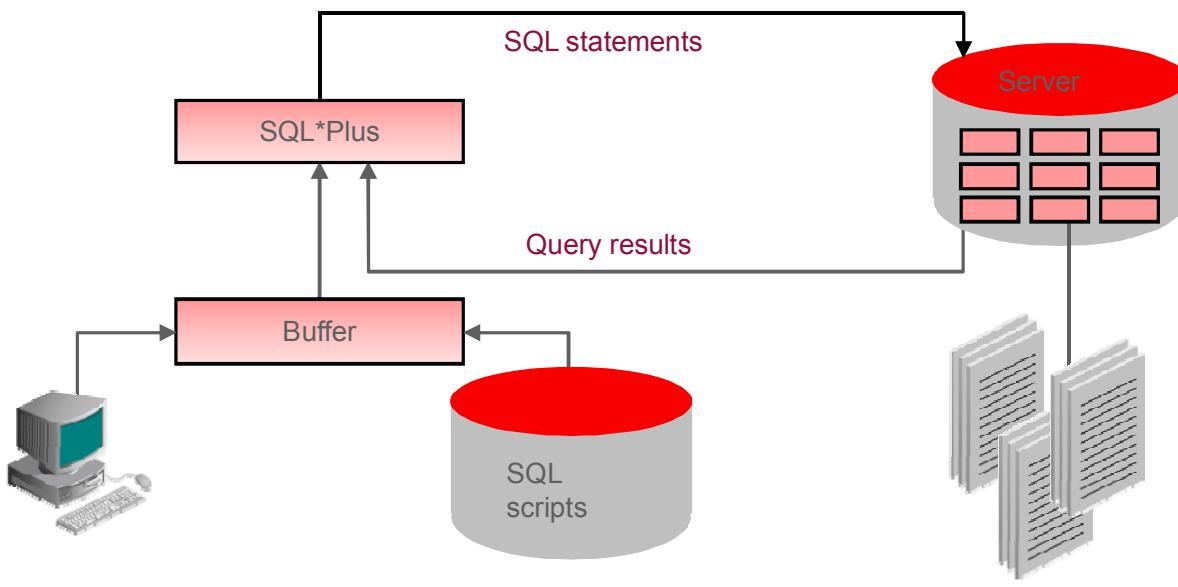


ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You might want to create SELECT statements that can be used repeatedly. This appendix covers the use of SQL\*Plus commands to execute SQL statements. You learn how to format the output by using SQL\*Plus commands, edit SQL commands, and save scripts in SQL\*Plus.

## SQL and SQL\*Plus Interaction



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## 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 Oracle9i 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.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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

## Overview of SQL\*Plus

- 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 buffer to edit.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

### 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

oracle@EDRSR2SP1:~/Desktop\$ sqlplus  
SQL\*Plus: Release 12.1.0.0.2 Beta on Thu Sep 13 02:00:57 2012  
Copyright (c) 1982, 2012, Oracle. All rights reserved.  
Enter user-name: oral  
Enter password:  
Last Successful login time: Wed Sep 2012 23:16:13 +00:00  
Connected to:  
Oracle Database 12c Enterprise Edition Release 12.1.0.0.2 - 64bit Beta  
With the Partitioning, OLAP, Data Mining and Real Application Testing options  
SQL>

`sqlplus [username[/password[@database]]]`

oracle@EDRSR2SP1:~/Desktop\$ sqlplus oral/oral  
SQL\*Plus: Release 12.1.0.0.2 Beta on Thu Sep 13 02:29:51 2012  
Copyright (c) 1982, 2012, Oracle. All rights reserved.  
Last Successful login time: Thu Sep 2012 02:01:21 +00:00  
Connected to:  
Oracle Database 12c Enterprise Edition Release 12.1.0.0.2 - 64bit Beta  
With the Partitioning, OLAP, Data Mining and Real Application Testing options  
SQL>

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 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:

<code>username</code>	Your database username
<code>password</code>	Your database password (Your password is visible if you enter it here.)
<code>@database</code>	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 © 2017, Oracle and/or its affiliates. All rights reserved.

In SQL\*Plus, you can 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 if a column must contain data.

In the syntax:

<b>tablename</b>	The name of any existing table, view, or synonym that is accessible to the user
------------------	---

To describe the DEPARTMENTS table, use the following 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)



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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*



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

SQL\*Plus commands are entered one line at a time and are not stored in the SQL buffer.

Command	Description
A [PPEND] <i>text</i>	Adds <i>text</i> to the end of the current line
C [HANGE] / <i>old</i> / <i>new</i>	Changes <i>old</i> text to <i>new</i> in the current line
C [HANGE] / <i>text</i> /	Deletes <i>text</i> from the current line
CL [EAR] BUFF [ER]	Deletes all lines from the SQL buffer
DEL	Deletes current line
DEL <i>n</i>	Deletes line <i>n</i>
DEL <i>m n</i>	Deletes lines <i>m</i> to <i>n</i> 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 appears.

## SQL\*Plus Editing Commands

- I [NPUT]
- I [NPUT] *text*
- L[IST]
- L[IST] *n*
- L[IST] *m n*
- R[UN]
- *n*
- *n text*
- O *text*



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Command	Description
I [NPUT]	Inserts an indefinite number of lines
I [NPUT] <i>text</i>	Inserts a line consisting of <i>text</i>
L[IST]	Lists all lines in the SQL buffer
L[IST] <i>n</i>	Lists one line (specified by <i>n</i> )
L[IST] <i>m n</i>	Lists a range of lines ( <i>m</i> to <i>n</i> ) inclusive
R[UN]	Displays and runs the current SQL statement in the buffer
<i>n</i>	Specifies the line to make the current line
<i>n text</i>	Replaces line <i>n</i> with <i>text</i>
O <i>text</i>	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 © 2017, Oracle and/or its affiliates. All rights reserved.

- 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 just their first letter. 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 © 2017, Oracle and/or its affiliates. All rights reserved.

- 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 © 2017, Oracle and/or its affiliates. All rights reserved.

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:

<b>Command</b>	<b>Description</b>
<code>SAV[E] filename [.ext] [REP[LACE] APP[END]]</code>	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.
<code>GET filename [.ext]</code>	Writes the contents of a previously saved file to the SQL buffer. The default extension for the file name is .sql.
<code>STA[RT] filename [.ext]</code>	Runs a previously saved command file
<code>@ filename</code>	Runs a previously saved command file (same as START)
<code>ED[IT]</code>	Invokes the editor and saves the buffer contents to a file named afiedt.buf
<code>ED[IT] [filename [.ext]]</code>	Invokes the editor to edit the contents of a saved file
<code>SPO[OL] [filename [.ext]]   OFF  OUT</code>	Stores query results in a file. OFF closes the spool file. OUT closes the spool file and sends the file results to the printer.
<code>EXIT</code>	Quits SQL*Plus

## Using the SAVE, 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.		



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

### 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 32767 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 | UNLIMITED} ] [FOR [MAT] {WRA[PPED] |  
WOR[D_WRAPPED] | TRU[NATED]} ]
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 done 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 2000 or greater than 1,000,000.
- For additional information about SERVEROUTPUT, see *Oracle Database PL/SQL User's Guide and Reference 12c*.

## Using the SQL\*Plus SPOOL Command

```
SPO [OL]  [file_name[.ext]  [CRE[ATE] | REP[LACE] | APP[END]] | OFF | OUT]
```

Option	Description
<b>file_name[.ext]</b>	Spools output to the specified file name
<b>CRE[ATE]</b>	Creates a new file with the name specified
<b>REP[LACE]</b>	Replaces the contents of an existing file. If the file does not exist, REPLACE creates the file.
<b>APP[END]</b>	Adds the contents of the buffer to the end of the file that you specify
<b>OFF</b>	Stops spooling
<b>OUT</b>	Stops spooling and sends the file to your computer's standard (default) printer



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The SPOOL command stores 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 use SPOOL only 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 screen, use SET TERMOUT OFF. SET TERMOUT OFF does not affect the output from commands that run interactively.

You must use quotation marks around file names that contain white space. To create a valid HTML file using 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]] [STATISTICS]
```

```
SET AUTOTRACE ON
-- The AUTOTRACE report includes both the optimizer
-- execution path and the SQL statement execution
-- statistics
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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 the package and subprograms, refer to *Oracle Database PL/SQL Packages and Types Reference 12c*.
- For additional information about the EXPLAIN PLAN, refer to *Oracle Database SQL Reference 12c*.
- For additional information about Execution Plans and the statistics, refer to *Oracle Database Performance Tuning Guide 12c*.

## Summary

In this appendix, you should have learned how to use SQL\*Plus as an environment to do the following:

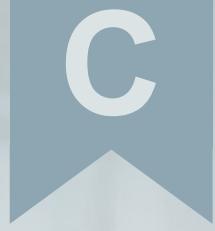
- Execute SQL statements
- Edit SQL statements
- Format the output
- Interact with script files



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.



# Commonly Used SQL Commands

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Objectives

After completing this appendix, you should be able to:

- Execute a basic `SELECT` statement
- Create, alter, and drop a table using the data definition language (DDL) statements
- Insert, update, and delete rows from one or more tables using data manipulation language (DML) statements
- Commit, roll back, and create save points using the transaction control statements
- Perform join operations on one or more tables



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

This lesson explains how to obtain data from one or more tables using the `SELECT` statement, how to use DDL statements to alter the structure of data objects, how to manipulate data in the existing schema objects by using the DML statements, how to manage the changes made by DML statements, and how to use joins to display data from multiple tables using SQL:1999 join syntax.

## Basic SELECT Statement

- Use the SELECT statement to:
  - Identify the columns to be displayed
  - Retrieve data from one or more tables, object tables, views, object views, or materialized views
- A SELECT statement is also known as a query because it queries a database.
- Syntax:

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



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In its simplest form, a SELECT statement must include the following:

- A SELECT clause, which specifies the columns to be displayed
- A FROM clause, which identifies the table containing the columns that are listed in the SELECT clause

In the syntax:

SELECT	Is a list of one or more columns
*	Selects all columns
DISTINCT	Suppresses duplicates
column expression	Selects the named column or the expression
alias	Gives different headings to the selected columns
FROM table	Specifies the table containing the columns

**Note:** Throughout this course, the words *keyword*, *clause*, and *statement* are used as follows:

- A *keyword* refers to an individual SQL element—for example, SELECT and FROM are keywords.
- A *clause* is a part of a SQL statement (for example, SELECT employee\_id, last\_name).
- A *statement* is a combination of two or more clauses (for example, SELECT \* FROM employees).

## SELECT Statement

- Select all columns:

```
SELECT *
FROM job_history;
```

	EMPLOYEE_ID	START_DATE	END_DATE	JOB_ID	DEPARTMENT_ID
1	102	13-JAN-01	24-JUL-06	IT_PROG	60
2	101	21-SEP-97	27-OCT-01	AC_ACCOUNT	110
3	101	28-OCT-01	15-MAR-05	AC_MGR	110
4	201	17-FEB-04	19-DEC-07	MK_REP	20
5	114	24-MAR-06	31-DEC-07	ST_CLERK	50
6	122	01-JAN-07	31-DEC-07	ST_CLERK	50
7	200	17-SEP-95	17-JUN-01	AD_ASST	90
8	176	24-MAR-06	31-DEC-06	SA REP	80
9	176	01-JAN-07	31-DEC-07	SA_MAN	80
10	200	01-JUL-02	31-DEC-06	AC_ACCOUNT	90

- Select specific columns:

```
SELECT manager_id, job_id
FROM employees;
```

	MANAGER_ID	JOB_ID
1	(null)	AD_PRES
2	100	AD_VP
3	100	AD_VP
4	102	IT_PROG
5	103	IT_PROG
6	103	IT_PROG
7	100	ST_MAN
8	124	ST_CLERK
9	124	ST_CLERK
10	124	ST_CLERK

...



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can display all columns of data in a table by following the `SELECT` keyword with an asterisk (\*) or by listing all the column names after the `SELECT` keyword. The first example in the slide displays all the rows from the `job_history` table. Specific columns of the table can be displayed by specifying the column names, separated by commas. The second example in the slide displays the `manager_id` and `job_id` columns from the `employees` table.

In the `SELECT` clause, specify the columns in the order in which you want them to appear in the output. For example, the following SQL statement displays the `location_id` column before displaying the `department_id` column:

```
SELECT location_id, department_id FROM departments;
```

**Note:** You can enter your SQL statement in a SQL Worksheet and click the Run Statement icon or press F9 to execute a statement in SQL Developer. The output displayed on the Results tabbed page appears as shown in the slide.

## WHERE Clause

- Use the optional WHERE clause to:
  - Filter rows in a query
  - Produce a subset of rows
- Syntax:

```
SELECT * FROM table  
[WHERE condition];
```

- Example:

```
SELECT location_id from departments  
WHERE department_name = 'Marketing';
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The WHERE clause specifies a condition to filter rows, producing a subset of the rows in the table. A condition specifies a combination of one or more expressions and logical (Boolean) operators. It returns a value of TRUE, FALSE, or NULL. The example in the slide retrieves the location\_id of the marketing department.

The WHERE clause can also be used to update or delete data from the database.

For example:

```
UPDATE departments  
SET department_name = 'Administration'  
WHERE department_id = 20;  
and  
DELETE from departments  
WHERE department_id =20;
```

## ORDER BY Clause

- Use the optional ORDER BY clause to specify the row order.
- Syntax:

```
SELECT * FROM table  
[WHERE condition]  
[ORDER BY {<column>}<position> } [ASC|DESC] [, ...] ;
```

- Example:

```
SELECT last_name, department_id, salary  
FROM employees  
ORDER BY department_id ASC, salary DESC;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The ORDER BY clause specifies the order in which the rows should be displayed. The rows can be sorted in ascending or descending fashion. By default, the rows are displayed in ascending order. The example in the slide retrieves rows from the EMPLOYEES table ordered first by ascending order of department\_id, and then by descending order of salary.

## GROUP BY Clause

- Use the optional GROUP BY clause to group columns that have matching values into subsets.
- Each group has no two rows having the same value for the grouping column or columns.
- Syntax:

```
SELECT <column1, column2, ... column_n>
  FROM table
 [WHERE condition]
 [GROUP BY <column> [, ...] ]
 [ORDER BY <column> [, ...] ] ;
```

- Example:

```
SELECT department_id, MIN(salary), MAX (salary)
  FROM employees
 GROUP BY department_id ;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The GROUP BY clause is used to group selected rows based on the value of `expr(s)` for each row. The clause groups rows but does not guarantee order of the result set. To order the groupings, use the ORDER BY clause.

Any SELECT list elements that are not included in aggregation functions must be included in the GROUP BY list of elements. This includes both columns and expressions. The database returns a single row of summary information for each group.

The example in the slide returns the minimum and maximum salaries for each department in the EMPLOYEES table.

## Data Definition Language

- DDL statements are used to define, structurally change, and drop schema objects.
- The commonly used DDL statements are:
  - CREATE TABLE, ALTER TABLE, and DROP TABLE
  - GRANT, REVOKE
  - TRUNCATE



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

DDL statements enable you to alter the attributes of an object without altering the applications that access the object. You can also use DDL statements to alter the structure of objects while database users are performing work in the database. These statements are most frequently used to:

- Create, alter, and drop schema objects and other database structures, including the database itself and database users
- Delete all the data in schema objects without removing the structure of these objects
- Grant and revoke privileges and roles

Oracle Database implicitly commits the current transaction before and after every DDL statement.

## CREATE TABLE Statement

- Use the CREATE TABLE statement to create a table in the database.
- Syntax:

```
CREATE TABLE tablename (
  {column-definition | Table-level constraint}
  [ , {column-definition | Table-level constraint} ] * )
```

- Example:

```
CREATE TABLE teach_dept (
  department_id NUMBER(3) PRIMARY KEY,
  department_name VARCHAR2(10));
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Use the CREATE TABLE statement to create a table in the database. To create a table, you must have the CREATE TABLE privilege and a storage area in which to create objects.

The table owner and the database owner automatically gain the following privileges on the table after it is created:

- INSERT
- SELECT
- REFERENCES
- ALTER
- UPDATE

The table owner and the database owner can grant the preceding privileges to other users.

## ALTER TABLE Statement

- Use the ALTER TABLE statement to modify the definition of an existing table in the database.
- Example1:

```
ALTER TABLE teach_dept  
ADD location_id NUMBER NOT NULL;
```

- Example 2:

```
ALTER TABLE teach_dept  
MODIFY department_name VARCHAR2(30) NOT NULL;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The ALTER TABLE statement allows you to make changes to an existing table.

You can:

- Add a column to a table
- Add a constraint to a table
- Modify an existing column definition
- Drop a column from a table
- Drop an existing constraint from a table
- Increase the width of the VARCHAR and CHAR columns
- Change a table to have read-only status

Example 1 in the slide adds a new column called LOCATION\_ID to the TEACH\_DEPT table.

Example 2 updates the existing DEPARTMENT\_NAME column from VARCHAR2 (10) to VARCHAR2 (30), and adds a NOT NULL constraint to it.

## DROP TABLE Statement

- The `DROP TABLE` statement removes the table and all its data from the database.
- Example:

```
DROP TABLE teach_dept;
```

- `DROP TABLE` with the `PURGE` clause drops the table and releases the space that is associated with it.

```
DROP TABLE teach_dept PURGE;
```

- The `CASCADE CONSTRAINTS` clause drops all referential integrity constraints from the table.

```
DROP TABLE teach_dept CASCADE CONSTRAINTS;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The `DROP TABLE` statement allows you to remove a table and its contents from the database, and pushes it to the recycle bin. Dropping a table invalidates dependent objects and removes object privileges on the table.

Use the `PURGE` clause along with the `DROP TABLE` statement to release back to the tablespace the space allocated for the table. You cannot roll back a `DROP TABLE` statement with the `PURGE` clause, nor can you recover the table if you have dropped it with the `PURGE` clause.

The `CASCADE CONSTRAINTS` clause allows you to drop the reference to the primary key and unique keys in the dropped table.

## GRANT Statement

- The GRANT statement assigns privilege to perform the following operations:
  - Insert or delete data.
  - Create a foreign key reference to the named table or to a subset of columns from a table.
  - Select data, a view, or a subset of columns from a table.
  - Create a trigger on a table.
  - Execute a specified function or procedure.
- Example:

```
GRANT SELECT any table to PUBLIC;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can use the GRANT statement to:

- Assign privileges to a specific user or role, or to all users, to perform actions on database objects
- Grant a role to a user, to PUBLIC, or to another role

Before you issue a GRANT statement, check that the `derby.database.sqlAuthorization` property is set to True. This property enables the SQL Authorization mode. You can grant privileges on an object if you are the owner of the database.

You can grant privileges to all users by using the PUBLIC keyword. When PUBLIC is specified, the privileges or roles affect all current and future users.

## Privilege Types

Assign the following privileges using the GRANT statement:

- ALL PRIVILEGES
- DELETE
- INSERT
- REFERENCES
- SELECT
- UPDATE



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Oracle Database provides a variety of privilege types to grant privileges to a user or role:

- Use the ALL PRIVILEGES privilege type to grant all privileges to the user or role for the specified table.
- Use the DELETE privilege type to grant permission to delete rows from the specified table.
- Use the INSERT privilege type to grant permission to insert rows into the specified table.
- Use the REFERENCES privilege type to grant permission to create a foreign key reference to the specified table.
- Use the SELECT privilege type to grant permission to perform SELECT statements on a table or view.
- Use the UPDATE privilege type to grant permission to use the UPDATE statement on the specified table.

## REVOKE Statement

- Use the REVOKE statement to remove privileges from a user to perform actions on database objects.
- Revoke a *system privilege* from a user:

```
REVOKE DROP ANY TABLE  
  FROM hr;
```

- Revoke a *role* from a user:

```
REVOKE dw_manager  
  FROM sh;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The REVOKE statement removes privileges from a specific user (or users) or role to perform actions on database objects. It performs the following operations:

- Revokes a role from a user, from PUBLIC, or from another role
- Revokes privileges for an object if you are the owner of the object or the database owner

**Note:** To revoke a role or system privilege, you must have been granted the privilege with the ADMIN OPTION.

## TRUNCATE TABLE Statement

- Use the TRUNCATE TABLE statement to remove all the rows from a table.
- Example:

```
TRUNCATE TABLE employees_demo;
```

- By default, Oracle Database performs the following tasks:
  - Deallocates space used by the removed rows
  - Sets the NEXT storage parameter to the size of the last extent removed from the segment by the truncation process



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The TRUNCATE TABLE statement deletes all the rows from a specific table. Removing rows with the TRUNCATE TABLE statement can be more efficient than dropping and re-creating a table. Dropping and re-creating a table:

- Invalidates the dependent objects of the table
- Requires you to regrant object privileges
- Requires you to re-create indexes, integrity constraints, and triggers
- Requires you to respecify its storage parameters

The TRUNCATE TABLE statement spares you from these efforts.

**Note:** You cannot roll back a TRUNCATE TABLE statement.

# Data Manipulation Language

- DML statements query or manipulate data in the existing schema objects.
- A DML statement is executed when:
  - New rows are added to a table by using the `INSERT` statement
  - Existing rows in a table are modified using the `UPDATE` statement
  - Existing rows are deleted from a table by using the `DELETE` statement
- A *transaction* consists of a collection of DML statements that form a logical unit of work.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Data Manipulation Language (DML) statements enable you to query or change the contents of an existing schema object. These statements are most frequently used to:

- Add new rows of data to a table or view by specifying a list of column values or using a subquery to select and manipulate existing data
- Change column values in the existing rows of a table or view
- Remove rows from tables or views

A collection of DML statements that forms a logical unit of work is called a transaction. Unlike DDL statements, DML statements do not implicitly commit the current transaction.

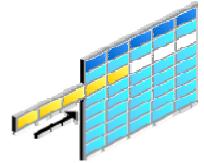
## INSERT Statement

- Use the `INSERT` statement to add new rows to a table.
- Syntax:

```
INSERT INTO table [(column [, column...])]
VALUES      (value [, value...]);
```

- Example:

```
INSERT INTO departments
VALUES      (200, 'Development', 104, 1400);
1 rows inserted.
```



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The `INSERT` statement adds rows to a table. Make sure to insert a new row containing values for each column and to list the values in the default order of the columns in the table. Optionally, you can also list the columns in the `INSERT` statement.

For example:

```
INSERT INTO job_history (employee_id, start_date, end_date, job_id)
VALUES (120, '25-JUL-06', '12-FEB-08', 'AC_ACCOUNT');
```

The syntax discussed in the slide allows you to insert a single row at a time. The `VALUES` keyword assigns the values of expressions to the corresponding columns in the column list.

## UPDATE Statement Syntax

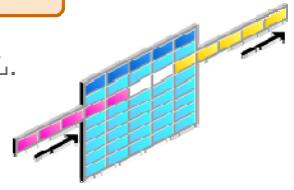
- Use the UPDATE statement to modify the existing rows in a table.
- Update more than one row at a time (if required).

```
UPDATE      table
SET        column = value [, column = value, ...]
[WHERE      condition];
```

- Example:

```
UPDATE      copy_emp
SET
[22 rows updated]
```

- Specify SET *column\_name*= NULL to update a column value to NULL.



**ORACLE**

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The UPDATE statement modifies the existing values in a table. Confirm the update operation by querying the table to display the updated rows. You can modify a specific row or rows by specifying the WHERE clause.

For example:

```
UPDATE employees
SET      salary = 17500
WHERE    employee_id = 102;
```

In general, use the primary key column in the WHERE clause to identify the row to update. For example, to update a specific row in the EMPLOYEES table, use *employee\_id* to identify the row instead of *employee\_name*, because more than one employee may have the same name.

**Note:** Typically, the condition keyword is composed of column names, expressions, constants, subqueries, and comparison operators.

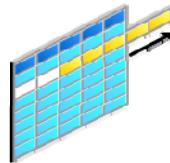
## DELETE Statement

- Use the DELETE statement to delete the existing rows from a table.
- Syntax:

```
DELETE [FROM] table  
[WHERE condition];
```

- Write the DELETE statement using the WHERE clause to delete specific rows from a table.

```
DELETE FROM departments  
WHERE department_name = 'Finance';  
1 rows deleted
```



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The DELETE statement removes existing rows from a table. You must use the WHERE clause to delete a specific row or rows from a table based on the condition. The condition identifies the rows to be deleted. It may contain column names, expressions, constants, subqueries, and comparison operators.

The first example in the slide deletes the finance department from the DEPARTMENTS table. You can confirm the delete operation by using the SELECT statement to query the table.

```
SELECT *  
FROM departments  
WHERE department_name = 'Finance';
```

If you omit the WHERE clause, all rows in the table are deleted. For example:

```
DELETE FROM copy_emp;
```

The preceding example deletes all the rows from the COPY\_EMP table.

## Transaction Control Statements

- Transaction control statements are used to manage the changes made by DML statements.
- The DML statements are grouped into transactions.
- Transaction control statements include:
  - COMMIT
  - ROLLBACK
  - SAVEPOINT



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A transaction is a sequence of SQL statements that Oracle Database treats as a single unit. Transaction control statements are used in a database to manage the changes made by DML statements and to group these statements into transactions.

Each transaction is assigned a unique `transaction_id` and it groups SQL statements so that they are either all committed, which means they are applied to the database, or all rolled back, which means they are undone from the database.

## COMMIT Statement

- Use the COMMIT statement to:
  - Permanently save the changes made to the database during the current transaction
  - Erase all savepoints in the transaction
  - Release transaction locks
- Example:

```
INSERT INTO departments
VALUES      (201, 'Engineering', 106, 1400);
COMMIT;

1 rows inserted.
committed.
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The COMMIT statement ends the current transaction by making all the pending data changes permanent. It releases all row and table locks, and erases any savepoints that you may have marked since the last commit or rollback. The changes made using the COMMIT statement are visible to all users.

Oracle recommends that you explicitly end every transaction in your application programs with a COMMIT or ROLLBACK statement, including the last transaction, before disconnecting from Oracle Database. If you do not explicitly commit the transaction and the program terminates abnormally, the last uncommitted transaction is automatically rolled back.

**Note:** Oracle Database issues an implicit COMMIT before and after any data definition language (DDL) statement.

## ROLLBACK Statement

- Use the ROLLBACK statement to undo changes made to the database during the current transaction.
- Use the TO SAVEPOINT clause to undo a part of the transaction after the savepoint.
- Example:

```
UPDATE      employees
SET         salary = 7000
WHERE       last_name = 'Ernst';
SAVEPOINT   Ernst_sal;

UPDATE      employees
SET         salary = 12000
WHERE       last_name = 'Mourgos';

ROLLBACK TO SAVEPOINT Ernst_sal;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The ROLLBACK statement undoes work done in the current transaction. To roll back the current transaction, no privileges are necessary.

Using ROLLBACK with the TO SAVEPOINT clause performs the following operations:

- Rolls back only the portion of the transaction after the savepoint
- Erases all savepoints created after that savepoint. The named savepoint is retained, so you can roll back to the same savepoint multiple times.

Using ROLLBACK without the TO SAVEPOINT clause performs the following operations:

- Ends the transaction
- Undoes all the changes in the current transaction
- Erases all savepoints in the transaction

## SAVEPOINT Statement

- Use the SAVEPOINT statement to name and mark the current point in the processing of a transaction.
- Specify a name to each savepoint.
- Use distinct savepoint names within a transaction to avoid overriding.
- Syntax:

```
SAVEPOINT savepoint;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The SAVEPOINT statement identifies a point in a transaction to which you can later roll back. You must specify a distinct name for each savepoint. If you create a second savepoint with the same identifier as an earlier savepoint, the earlier savepoint is erased.

After a savepoint has been created, you can continue processing, commit your work, roll back the entire transaction, or roll back to the savepoint.

A simple rollback or commit erases all savepoints. When you roll back to a savepoint, any savepoints marked after that savepoint are erased. The savepoint to which you have rolled back is retained.

When savepoint names are reused within a transaction, the Oracle Database moves (overrides) the save point from its old position to the current point in the transaction.

## Joins

Use a join to query data from more than one table:

```
SELECT    table1.column, table2.column
FROM      table1, table2
WHERE     table1.column1 = table2.column2;
```

- Write the join condition in the WHERE clause.
- Prefix the column name with the table name when the same column name appears in more than one table.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When data from more than one table in the database is required, a *join* condition is used. Rows in one table can be joined to rows in another table according to common values that exist in the corresponding columns (usually primary and foreign key columns).

To display data from two or more related tables, write a simple join condition in the WHERE clause.

In the syntax:

*table1.column*  
*table1.column1* =  
*table2.column2*

Denotes the table and column from which data is retrieved  
 Is the condition that joins (or relates) the tables together

### Guidelines

- When writing a SELECT statement that joins tables, precede the column name with the table name for clarity and to enhance database access.
- If the same column name appears in more than one table, the column name must be prefixed with the table name.
- To join  $n$  tables together, you need a minimum of  $n-1$  join conditions. For example, to join four tables, a minimum of three joins is required. This rule may not apply if your table has a concatenated primary key, in which case more than one column is required to uniquely identify each row.

## Types of Joins

- Natural join
- Equijoin
- Nonequijoin
- Outer join
- Self-join
- Cross join



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

To join tables, you can use Oracle's join syntax.

**Note:** Before the Oracle9*i* release, the join syntax was proprietary. The SQL:1999-compliant join syntax does not offer any performance benefits over the Oracle-proprietary join syntax.

# Qualifying Ambiguous Column Names

- Use table prefixes to qualify column names that are in multiple tables.
- Use table prefixes to improve performance.
- Use table aliases, instead of full table name prefixes.
- Table aliases give a table a shorter name.
  - This keeps SQL code smaller and uses less memory.
- Use column aliases to distinguish columns that have identical names, but reside in different tables.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

When joining two or more tables, you need to qualify the names of the columns with the table name to avoid ambiguity. Without the table prefixes, the `DEPARTMENT_ID` column in the `SELECT` list could be from either the `DEPARTMENTS` table or the `EMPLOYEES` table. Therefore, it is necessary to add the table prefix to execute your query. If there are no common column names between the two tables, there is no need to qualify the columns. However, using a table prefix improves performance, because you tell the Oracle server exactly where to find the columns.

Qualifying column names with table names can be very time consuming, particularly if table names are lengthy. Therefore, you can use *table aliases*, instead of table names. Just as a column alias gives a column another name, a table alias gives a table another name. Table aliases help to keep SQL code smaller, thereby using less memory.

The table name is specified in full, followed by a space, and then the table alias. For example, the `EMPLOYEES` table can be given an alias of `e`, and the `DEPARTMENTS` table an alias of `d`.

## Guidelines

- Table aliases can be up to 30 characters in length, but shorter aliases are better than longer ones.
- If a table alias is used for a particular table name in the `FROM` clause, that table alias must be substituted for the table name throughout the `SELECT` statement.
- Table aliases should be meaningful.
- A table alias is valid only for the current `SELECT` statement.

## Natural Join

- The NATURAL JOIN clause is based on all the columns in the two tables that have the same name.
- It selects rows from tables that have the same names and data values of columns.
- Example:

```
SELECT country_id, location_id, country_name, city
FROM countries NATURAL JOIN locations;
```

	COUNTRY_ID	LOCATION_ID	COUNTRY_NAME	CITY
1	US	1400	United States of America	Southlake
2	US	1500	United States of America	South San Francisco
3	US	1700	United States of America	Seattle
4	CA	1800	Canada	Toronto
5	UK	2500	United Kingdom	Oxford



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You can join tables automatically based on the columns in the two tables that have matching data types and names. You do this by using the NATURAL JOIN keywords.

**Note:** The join can happen only on those columns that have the same names and data types in both tables. If the columns have the same name but different data types, the NATURAL JOIN syntax causes an error.

In the example in the slide, the COUNTRIES table is joined to the LOCATIONS table by the COUNTRY\_ID column, which is the only column of the same name in both tables. If other common columns were present, the join would have used them all.

# Equijoins

EMPLOYEES

	EMPLOYEE_ID	DEPARTMENT_ID
1	200	10
2	201	20
3	202	20
4	205	110
5	206	110
6	100	90
7	101	90
8	102	90
9	103	60
10	104	60

DEPARTMENTS

	DEPARTMENT_ID	DEPARTMENT_NAME
1	10	Administration
2	20	Marketing
3	50	Shipping
4	60	IT
5	80	Sales
6	90	Executive
7	110	Accounting
8	190	Contracting

Foreign key

Primary key



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

An **equijoin** is a join with a join condition containing an equality operator. An equijoin combines rows that have equivalent values for the specified columns. To determine an employee's department name, you compare the values in the `DEPARTMENT_ID` column in the `EMPLOYEES` table with the `DEPARTMENT_ID` values in the `DEPARTMENTS` table. The relationship between the `EMPLOYEES` and `DEPARTMENTS` tables is an **equijoin**; that is, values in the `DEPARTMENT_ID` column in both tables must be equal. Often, this type of join involves primary and foreign key complements.

**Note:** Equijoins are also called *simple joins*.

## Retrieving Records with Equijoins

```
SELECT e.employee_id, e.last_name, e.department_id,
       d.department_id, d.location_id
  FROM employees e JOIN departments d
 WHERE e.department_id = d.department_id;
```

EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_ID_1	LOCATION_ID
1	200 Whalen	10	10	1700
2	201 Hartstein	20	20	1800
3	202 Fay	20	20	1800
4	144 Vargas	50	50	1500
5	143 Matos	50	50	1500
6	142 Davies	50	50	1500
7	141 Rajs	50	50	1500
8	124 Mourgos	50	50	1500
9	103 Hunold	60	60	1400
10	104 Ernst	60	60	1400
11	107 Lorentz	60	60	1400
...				



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the example in the slide:

- **The SELECT clause specifies the column names to retrieve:**
  - Employee last name, employee ID, and department ID, which are columns in the EMPLOYEES table
  - Department ID and location ID, which are columns in the DEPARTMENTS table
- **The FROM clause specifies the two tables that the database must access:**
  - EMPLOYEES table
  - DEPARTMENTS table
- **The WHERE clause specifies how the tables are to be joined:**

e.department\_id = d.department\_id

Because the DEPARTMENT\_ID column is common to both tables, it must be prefixed with the table alias to avoid ambiguity. Other columns that are not present in both the tables need not be qualified by a table alias, but it is recommended for better performance.

**Note:** When you use the Execute Statement icon to run the query, SQL Developer suffixes a “\_1” to differentiate between the two DEPARTMENT\_IDS.

## Additional Search Conditions Using the AND and WHERE Operators

```
SELECT d.department_id, d.department_name, l.city
FROM departments d JOIN locations l
ON d.location_id = l.location_id
AND d.department_id IN (20, 50);
```

	DEPARTMENT_ID	DEPARTMENT_NAME	CITY
1	20	Marketing	Toronto
2	50	Shipping	South San Francisco

```
SELECT d.department_id, d.department_name, l.city
FROM departments d JOIN locations l
ON d.location_id = l.location_id
WHERE d.department_id IN (20, 50);
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In addition to the join, you may have criteria for your WHERE clause to restrict the rows in consideration for one or more tables in the join. The example in the slide performs a join on the DEPARTMENTS and LOCATIONS tables and, in addition, displays only those departments with ID equal to 20 or 50. To add additional conditions to the ON clause, you can add AND clauses. Alternatively, you can use a WHERE clause to apply additional conditions.

Both queries produce the same output.

	LAST_NAME	DEPARTMENT_ID	DEPARTMENT_NAME
1	Matos	50	Shipping

## Retrieving Records with Nonequijoins

```
SELECT e.last_name, e.salary, j.grade_level
FROM employees e JOIN job_grades j
ON e.salary
BETWEEN j.lowest_sal AND j.highest_sal;
```

	LAST_NAME	SALARY	GRADE_LEVEL
1	Vargas	2500	A
2	Matos	2600	A
3	Davies	3100	B
4	Rajs	3500	B
5	Lorentz	4200	B
6	Whalen	4400	B
7	Fay	6000	C

...



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The example in the slide creates a nonequijoin to evaluate an employee's salary grade. The salary must be *between* any pair of the low and high salary ranges.

It is important to note that all employees appear exactly once when this query is executed. No employee is repeated in the list. There are two reasons for this:

- None of the rows in the job grade table contain grades that overlap. That is, the salary value for an employee can lie only between the low salary and high salary values of one of the rows in the salary grade table.
- All of the employees' salaries lie within the limits that are provided by the job grade table. That is, no employee earns less than the lowest value contained in the `LOWEST_SAL` column or more than the highest value contained in the `HIGHEST_SAL` column.

**Note:** Other conditions (such as `<=` and `>=`) can be used, but `BETWEEN` is the simplest. Remember to specify the low value first and the high value last when using the `BETWEEN` condition. The Oracle server translates the `BETWEEN` condition to a pair of `AND` conditions. Therefore, using `BETWEEN` has no performance benefits, but should be used only for logical simplicity.

Table aliases have been specified in the example in the slide for performance reasons, not because of possible ambiguity.

## Retrieving Records by Using the USING Clause

- You can use the USING clause to match only one column when more than one column matches.
- You cannot specify this clause with a NATURAL join.
- Do not qualify the column name with a table name or table alias.
- Example:

```
SELECT country_id, country_name, location_id, city
FROM   countries JOIN locations
USING (country_id) ;
```

COUNTRY_ID	COUNTRY_NAME	LOCATION_ID	CITY
1 US	United States of America	1400 Southlake	
2 US	United States of America	1500 South San Francisco	
3 US	United States of America	1700 Seattle	
4 CA	Canada	1800 Toronto	
5 UK	United Kingdom	2500 Oxford	



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the COUNTRY\_ID columns in the COUNTRIES and LOCATIONS tables are joined and thus the LOCATION\_ID of the location where an employee works is shown.

## Retrieving Records by Using the ON Clause

- The join condition for the natural join is basically an equijoin of all columns with the same name.
- Use the `ON` clause to specify arbitrary conditions or specify columns to join.
- The `ON` clause makes code easy to understand.

```
SELECT e.employee_id, e.last_name, j.department_id,
FROM   employees e JOIN job_history j
ON     (e.employee_id = j.employee_id);
```

	EMPLOYEE_ID	LAST_NAME	DEPARTMENT_ID
1	101 Kochhar		110
2	101 Kochhar		110
3	102 De Haan		60
4	176 Taylor		80
5	176 Taylor		80
6	200 Whalen		90
7	200 Whalen		90
8	201 Hartstein		20



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Use the `ON` clause to specify a join condition. With this, you can specify join conditions separate from any search or filter conditions in the `WHERE` clause.

In this example, the `EMPLOYEE_ID` columns in the `EMPLOYEES` and `JOB_HISTORY` tables are joined using the `ON` clause. Wherever an employee ID in the `EMPLOYEES` table equals an employee ID in the `JOB_HISTORY` table, the row is returned. The table alias is necessary to qualify the matching column names.

You can also use the `ON` clause to join columns that have different names. The parentheses around the joined columns, as in the example in the slide, `(e.employee_id = j.employee_id)`, is optional. So, even `ON e.employee_id = j.employee_id` will work.

**Note:** When you use the Execute Statement icon to run the query, SQL Developer suffixes a '`_1`' to differentiate between the two `employee_ids`.

## Left Outer Join

- A join between two tables that returns all matched rows, as well as the unmatched rows from the left table is called a LEFT OUTER JOIN.
- Example:

```
SELECT c.country_id, c.country_name, l.location_id, l.city
FROM   countries c LEFT OUTER JOIN locations l
ON    (c.country_id = l.country_id) ;
```

COUNTRY_ID	COUNTRY_NAME	LOCATION_ID	CITY
1 CA	Canada	1800	Toronto
2 DE	Germany	(null)	(null)
3 UK	United Kingdom	2500	Oxford
4 US	United States of America	1400	Southlake
5 US	United States of America	1500	South San Francisco
6 US	United States of America	1700	Seattle



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

This query retrieves all the rows in the COUNTRIES table, which is the left table, even if there is no match in the LOCATIONS table.

## Right Outer Join

- A join between two tables that returns all matched rows, as well as the unmatched rows from the right table is called a **RIGHT OUTER JOIN**.
- Example:

```
SELECT d.department_id, d.department_name, l.location_id,  
l.city  
FROM departments d RIGHT OUTER JOIN locations l  
ON (d.location_id = l.location_id) ;
```

	DEPARTMENT_ID	DEPARTMENT_NAME	LOCATION_ID	CITY
1	10	Administration	1700	Seattle
2	20	Marketing	1800	Toronto
3	30	Purchasing	1700	Seattle
4	40	Human Resources	2400	London
...				
26	260	Recruiting	1700	Seattle
27	270	Payroll	1700	Seattle
28	(null)	(null)	2000	Beijing
29	(null)	(null)	3000	Bern
...				



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

This query retrieves all the rows in the LOCATIONS table, which is the right table, even if there is no match in the COUNTRIES table.

## Full Outer Join

- A join between two tables that returns all matched rows, as well as the unmatched rows from both tables is called a FULL OUTER JOIN.
- Example:

```
SELECT e.last_name, d.department_id, d.manager_id,  
       d.department_name  
  FROM employees e FULL OUTER JOIN departments d  
    ON (e.manager_id = d.manager_id) ;
```

	LAST_NAME	DEPARTMENT_ID	MANAGER_ID	DEPARTMENT_NAME
1	King	(null)	(null)	(null)
2	Kochhar	90	100	Executive
3	De Haan	90	100	Executive
4	Hunold	(null)	(null)	(null)

...
106 Higgins
107 Gietz
108 (null)
109 (null)

...



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

This query retrieves all the rows in the EMPLOYEES table, even if there is no match in the DEPARTMENTS table. It also retrieves all the rows in the DEPARTMENTS table, even if there is no match in the EMPLOYEES table.

## Self-Join: Example

```
SELECT worker.last_name || ' works for '
    || manager.last_name
  FROM employees worker JOIN employees manager
 WHERE worker.manager_id = manager.employee_id
 ORDER BY worker.last_name;
```

WORKER LAST NAME  ' WORKS FOR'  MANAGER LAST NAME
1 Abel works for Zlotkey
2 Davies works for Mourgos
3 De Haan works for King
4 Ernst works for Hunold
5 Fay works for Hartstein
6 Gietz works for Higgins
7 Grant works for Zlotkey
8 Hartstein works for King
9 Higgins works for Kochhar

...

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Sometimes you need to join a table to itself. To find the name of each employee's manager, you need to join the EMPLOYEES table to itself, or perform a self-join. The example in the slide joins the EMPLOYEES table to itself. To simulate two tables in the FROM clause, there are two aliases, namely worker and manager, for the same table, EMPLOYEES.

In this example, the WHERE clause contains the join that means "where a worker's manager ID matches the employee ID for the manager."

## Cross Join

- A CROSS JOIN is a JOIN operation that produces the Cartesian product of two tables.
- Example:

```
SELECT department_name, city
FROM department CROSS JOIN location;
```

DEPARTMENT_NAME	CITY
1 Administration	Oxford
2 Administration	Seattle
3 Administration	South San Francisco
4 Administration	Southlake
5 Administration	Toronto
6 Marketing	Oxford
7 Marketing	Seattle
8 Marketing	South San Francisco
9 Marketing	Southlake
10 Marketing	Toronto
...	



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The CROSS JOIN syntax specifies the cross product. It is also known as a Cartesian product. A cross join produces the cross product of two relations, and is essentially the same as the comma-delimited Oracle Database notation.

You do not specify any WHERE condition between the two tables in the CROSS JOIN.

## Summary

In this appendix, you should have learned how to use:

- The SELECT statement to retrieve rows from one or more tables
- DDL statements to alter the structure of objects
- DML statements to manipulate data in the existing schema objects
- Transaction control statements to manage the changes made by DML statements
- Joins to display data from multiple tables



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

There are many commonly used commands and statements in SQL. It includes the DDL statements, DML statements, transaction control statements, and joins.





D

# REF Cursors

ORACLE®

Copyright © 2017, Oracle and/or its affiliates. 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.



ORACLE®

Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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;
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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
```

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
```

## 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 ...
```

## 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.



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

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`.

## 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.

## 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;
/
```



Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.