



Integrated Cloud Applications & Platform Services

# Custom CW: Oracle Database 12c R2 - Six Payment

Student Guide

X103804GC10

Edition 1.0 | November 2019

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



**Copyright © 2019, 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.

# Table of Contents

<b>Chapter 1: Lesson 10 - Creating Compound, DDL, and Event Database Triggers .....</b>	<b>13</b>
Objectives .....	14
Course Road Map .....	15
Lesson Agenda .....	16
What is a Compound Trigger? .....	17
Working with Compound Triggers .....	18
Why Compound Triggers? .....	19
Compound Trigger Structure .....	20
Compound Trigger Structure for Views .....	21
Compound Trigger Restrictions .....	22
Lesson Agenda .....	23
Mutating Tables .....	24
Mutating Table: Example .....	25
Using a Compound Trigger to Resolve the Mutating Table Error .....	27
Lesson Agenda .....	30
Creating Triggers on DDL Statements .....	31
Creating Triggers on DDL Statements -Example .....	32
Lesson Agenda .....	33
Creating Database Triggers .....	34
Creating Triggers on System Events .....	35
LOGON and LOGOFF Triggers: Example .....	36
Lesson Agenda .....	37
Guidelines for Designing Triggers .....	38
Quiz .....	39
Summary .....	40
Practice 10 Overview: Creating Compound, DDL, and Event Database Triggers .....	41
<b>Chapter 2: Lesson 13 - Managing Dependencies .....</b>	<b>43</b>
Objectives .....	44
Course Road Map .....	45
Lesson Agenda .....	46
What are Dependencies in a Schema? .....	47
How Dependencies Work? .....	48
Dependent and Referenced Objects .....	50
Querying Object Dependencies: Using the USER_DEPENDENCIES View .....	52
Querying an Object's Status .....	53
Categorizing Dependencies .....	54
Lesson Agenda .....	55
Direct Dependencies .....	56
Indirect Dependencies .....	57
Displaying Direct and Indirect Dependencies .....	58
Lesson Agenda .....	60
Fine-Grained Dependency Management .....	61
Fine-Grained Dependency Management: Example 1 .....	63

Fine-Grained Dependency Management: Example 2 .....	65
Guidelines for Reducing Invalidation .....	66
Object Revalidation .....	67
Lesson Agenda .....	68
Remote Dependencies .....	69
Managing Remote Procedure Dependencies .....	70
Setting the REMOTE_DEPENDENCIES_MODE Parameter .....	71
Timestamp Checking .....	72
Signature Checking .....	77
Lesson Agenda .....	78
Revalidating PL/SQL Program Units .....	79
Unsuccessful Recompilation .....	80
Successful Recompilation .....	81
Recompiling Procedures .....	82
Lesson Agenda .....	83
Packages and Dependencies: Subprogram References the Package .....	84
Packages and Dependencies: Package Subprogram References Procedure .....	85
Quiz .....	86
Summary .....	87
Practice 13 Overview: Managing Dependencies in Your Schema .....	88
<b>Chapter 3: Lesson 1 - Introduction .....</b>	<b>89</b>
Course Agenda .....	90
Lesson Agenda .....	91
Assumptions .....	92
Course Objectives .....	93
Course Agenda .....	94
Appendices Used in This Course .....	95
Lesson Agenda .....	96
PL/SQL Development Environments .....	97
Oracle SQL Developer .....	98
Specifications of SQL Developer .....	99
SQL Developer 4.1.5 Interface .....	100
Coding PL/SQL in SQL*Plus .....	101
Lesson Agenda .....	102
Tables Used in This Course .....	103
Order Entry Schema .....	104
Human Resources Schema .....	106
Lesson Agenda .....	107
Oracle Database 12c: Focus Areas .....	108
Oracle Database 12c .....	109
Lesson Agenda .....	111
Introduction to Oracle Cloud .....	112
Oracle Cloud Services .....	114
Cloud Deployment Models .....	116
Lesson Agenda .....	118
Oracle SQL and PL/SQL Documentation .....	119

Summary .....	120
Practice 1 Overview: Getting Started .....	121
<b>Chapter 4: Lesson 2 - Oracle Database Exadata Express Cloud Service .....</b>	<b>123</b>
Course Agenda .....	124
Lesson Objectives .....	125
Lesson Agenda .....	126
Evolving from On-premises to Exadata Express .....	127
Exadata Express for Users .....	128
Exadata Express for Developers .....	129
Getting Started with Exadata Express .....	130
Managing Exadata .....	132
Lesson Agenda .....	133
Service Console .....	134
Web Access Through Service Console .....	135
Client Access Configuration Through Service Console .....	136
Database Administration Through Service Console .....	137
Lesson Agenda .....	138
SQL Workshop .....	139
Lesson Agenda .....	142
Connecting Through Database Clients .....	143
Enabling SQL*Net Access for Client Applications .....	144
Downloading Client Credentials .....	145
Connecting Oracle SQL Developer .....	146
Connecting Oracle SQLcl .....	147
Summary .....	148
Practice 2: Overview .....	149
<b>Chapter 5: Lesson 3 - Overview of Collections .....</b>	<b>151</b>
Course Agenda .....	152
Objectives .....	153
Lesson Agenda .....	154
Collections .....	155
Why Collections? .....	156
Collection Types .....	157
Collection Types (Notes Only) .....	158
Lesson Agenda .....	159
Using Associative Arrays .....	160
Creating an Associative Array .....	161
Traversing an Associative Array .....	162
Collection Methods .....	164
Lesson Agenda .....	165
Nested Tables .....	166
Creating Nested Table Types .....	167
Nested Tables - Example .....	168
Collection Constructors .....	169
Declaring Collections: Nested Table .....	170
Using Nested Tables .....	171

Referencing Collection Elements .....	173
Using Nested Tables in PL/SQL .....	174
Lesson Agenda .....	176
Varrays .....	177
Declaring Collections: Varray .....	179
Using Varrays .....	180
Quiz .....	182
Summary .....	184
<b>Chapter 6: Lesson 4 - Using Collections .....</b>	<b>185</b>
Course Agenda .....	186
Objectives .....	187
Lesson Agenda .....	188
Usage of Collections in Applications .....	189
Working with Collections in PL/SQL .....	190
Assigning Values to Collection Variables .....	193
Accessing Values in the Collection .....	196
Working with Collection Methods .....	197
Using Collection Methods .....	198
Manipulating Individual Elements .....	200
Querying a Collection Using the TABLE Operator .....	202
Querying a Collection with the TABLE Operator .....	203
Lesson Agenda .....	205
Collection Exceptions .....	206
Avoiding Collection Exceptions: Example .....	207
Lesson Agenda .....	208
Listing Characteristics for Collections .....	209
Lesson Agenda .....	210
PL/SQL Bind Types .....	211
Subprogram with a BOOLEAN Parameter .....	212
Subprogram with a BOOLEAN parameter .....	213
Subprogram with a Record Parameter (Notes Only) .....	214
Quiz .....	215
Summary .....	217
Practice 4: Overview .....	218
<b>Chapter 7: Lesson 5 - Handling Large Objects .....</b>	<b>219</b>
Course Agenda .....	220
Objectives .....	221
Lesson Agenda .....	222
What Is a LOB? .....	223
Types of LOBs .....	224
LOB Locators and LOB Values .....	225
Lesson Agenda .....	226
DBMS_LOB Package .....	227
Security Model of DBMS_LOB Package .....	228
What Is a DIRECTORY Object? .....	229
Managing BFILEs: Role of a DBA .....	230

Managing BFILEs: Role of a Developer .....	231
Lesson Agenda .....	232
Working on BFILEs .....	233
Preparing to Use BFILEs .....	234
Creating BFILE Columns in the Table .....	235
Populating a BFILE Column with PL/SQL .....	236
Using data in the BFILE Column .....	237
Lesson Agenda .....	238
Working on CLOBs .....	239
Initializing LOB Columns Added to a Table .....	240
Populating LOB Columns .....	242
Loading Data to a LOB Column .....	243
Writing Data to a LOB .....	244
Important Instructions .....	246
Lesson Agenda .....	247
Reading LOBs from the Table .....	248
Updating LOB by Using DBMS_LOB in PL/SQL .....	249
Selecting CLOB Values by Using SQL .....	250
Selecting CLOB Values by Using DBMS_LOB .....	251
Selecting CLOB Values in PL/SQL .....	252
Removing LOBs .....	253
Quiz .....	254
Lesson Agenda .....	257
Temporary LOBs .....	258
Creating a Temporary LOB .....	259
Lesson Agenda .....	260
SecureFile LOBs .....	261
Storage of SecureFile LOBs .....	262
Creating a SecureFile LOB .....	263
Quiz .....	264
Summary .....	265
Practice 5: Overview .....	266
<b>Chapter 8: Lesson 6 - Working with JSON Data .....</b>	<b>267</b>
Course Agenda .....	268
Objectives .....	269
Lesson Agenda .....	270
What Is JSON? .....	271
Structure of JSON Data .....	272
JSON Data: Example .....	273
JSON Data: Example (Notes Only) .....	274
Why JSON? .....	275
Lesson Agenda .....	276
JSON Data in Oracle Database - Scenario .....	277
JSON Data in Oracle Database .....	278
Creating a Table with JSON Column .....	280
JSON or Not? .....	282

Inserting Data into JSON Columns .....	283
Lesson Agenda .....	284
SQL/JSON Generation Functions .....	285
JSON_OBJECT Function .....	286
JSON_ARRAY Function .....	287
JSON_OBJECTAGG Function .....	288
JSON_ARRAYAGG Function .....	289
SQL/JSON Functions .....	290
Lesson Agenda .....	291
Retrieving SQL Data from JSON Object .....	292
Accessing JSON Data .....	293
JSON_VALUE Function .....	295
Using SQL/JSON Functions .....	296
Lesson Agenda .....	297
PL/SQL Objects for JSON .....	298
JSON Object Types in PL/SQL .....	299
JSON Object Methods .....	300
Getter and Setter Methods .....	301
JSON Methods in PL/SQL Example .....	302
Summary .....	304
Practice 6: Overview .....	305

## **Chapter 9: Lesson 7 - Using Advanced Interface Methods**

Course Agenda .....	308
Objectives .....	309
Lesson Agenda .....	310
PL/SQL External Procedures .....	311
Oracle Database with Different Languages .....	312
Scenario .....	313
Lesson Agenda .....	314
External Procedure Execution Architecture .....	315
Components for External C Procedure Execution .....	316
Defining an External C Procedure .....	317
Define a C Function .....	318
Creating an Alias Library .....	319
Publishing External C Procedures .....	321
Call Specification Syntax .....	322
Call Specification .....	323
Publishing an External C Routine .....	324
Executing an External C Procedure .....	325
Lesson Agenda .....	326
Executing Java Programs from PL/SQL .....	327
External Procedure Execution Architecture .....	328
Development Steps for Java Class Methods .....	329
Loading Java Class Methods .....	330
Publishing a Java Class Method .....	331
Publishing a Java Class Method (Notes Only) .....	332

Executing the Java Routine .....	333
Creating Call Specifications in Packages .....	334
Quiz .....	335
Summary .....	338
Practice 7: Overview .....	339
<b>Chapter 10: Lesson 8 - Performance and Tuning .....</b>	<b>341</b>
Course Agenda .....	342
Objectives .....	343
Lesson Agenda .....	344
Compiling a PL/SQL Unit .....	345
Deciding on a Compilation Method .....	346
Configuring the Compiler .....	347
Viewing the Compilation Settings .....	349
Setting Up a Database for Native Compilation .....	351
Modifying Compilation Mode of a Program Unit .....	352
Lesson Agenda .....	353
PL/SQL Optimizer .....	354
Subprogram Inlining: Introduction .....	355
Using Inlining .....	356
Inlining Concepts .....	357
Inlining: How to Enable It ? .....	360
PRAGMA INLINE Example .....	361
Inlining: Summary .....	362
Lesson Agenda .....	363
Why PL/SQL Tuning .....	364
Tuning PL/SQL Code .....	365
Avoid Implicit Data Type Conversion .....	366
NOT NULL Constraint .....	367
PLS_INTEGER Data Type for Integers .....	368
Using the SIMPLE_INTEGER Data Type .....	369
Modularizing Your Code .....	370
Bulk Binding .....	371
FORALL Instead of FOR .....	372
BULK COLLECT .....	374
Exception While Bulk Collecting .....	377
Handling FORALL Exceptions .....	378
Tuning Conditional Control Statements .....	379
Passing Data Between PL/SQL Programs .....	381
Quiz .....	384
Summary .....	387
Practice 8: Overview .....	388
<b>Chapter 11: Lesson 9 - Improving Performance with Caching .....</b>	<b>389</b>
Course Agenda .....	390
Objectives .....	391
Lesson Agenda .....	392
What Is Caching? .....	393

Memory Architecture .....	394
Caching in the Database Instance .....	395
What Is Result Caching? .....	396
Lesson Agenda .....	397
Configuring the Server Result Cache .....	398
Configuring the Server Result Cache (Notes Only) .....	399
Setting Result_Cache_Max_Size .....	400
Setting the Result Cache Mode .....	401
Using the DBMS_RESULT_CACHE Package .....	402
Lesson Agenda .....	403
SQL Query Result Cache .....	404
Examining the Memory Cache .....	406
Examining the Execution Plan for a Query .....	407
Examining Another Execution Plan .....	408
Executing Both Queries .....	409
Viewing Cache Results Created .....	410
Viewing Cache Results Found .....	411
Lesson Agenda .....	412
PL/SQL Function Result Cache .....	413
Marking PL/SQL Function Results to Be Cached .....	414
Clearing the Shared Pool and Result Cache .....	415
Creating a PL/SQL Function by Using the RESULT_CACHE Clause .....	416
Calling the PL/SQL Function Inside a Query .....	417
Viewing Cache Results Created .....	418
Calling the PL/SQL Function Again .....	419
Viewing Cache Results Found .....	420
Confirming That the Cached Result Was Used .....	421
Lesson Agenda .....	422
Oracle Database In-Memory .....	423
Quiz .....	424
Summary .....	429
Practice 9: Overview .....	430
<b>Chapter 12: Lesson 10 - Analyzing PL/SQL Code .....</b>	<b>431</b>
Course Agenda .....	432
Objectives .....	433
Lesson Agenda .....	434
PL/SQL Code Analysis .....	435
Data Dictionary Views .....	437
Analyzing PL/SQL Code .....	438
Using SQL Developer for Code Analysis .....	443
Using ALL_ARGUMENTS .....	445
Using ALL_ARGUMENTS (Notes Only) .....	446
ALL_ARGUMENTS .....	448
Using SQL Developer to Report on Arguments .....	449
Lesson Agenda .....	451
PL/Scope .....	452

Using PL/Scope .....	454
USER_IDENTIFIERS View .....	455
Sample Data for PL/Scope .....	456
Collecting Information on Identifiers .....	457
Viewing Identifier Information .....	458
Performing a Basic Identifier Search .....	460
Using USER_IDENTIFIERS to Find All Local Variables .....	461
Finding Identifier Actions .....	462
Lesson Agenda .....	464
Oracle Supplied Packages for Code Analysis .....	465
Using DBMS_DESCRIBE .....	466
DBMS.Utility Package .....	469
Using DBMS.Utility FORMAT_CALL_STACK .....	470
Using DBMS.Utility .....	472
Finding Error Information .....	474
DBMS_Metadata Package .....	477
DBMS_Metadata Subprograms .....	478
Subprograms in DBMS_Metadata (Notes Only) .....	479
FETCH_xxx Subprograms .....	480
Filters on Metadata .....	481
SET_FILTER Procedure .....	482
Examples of Setting Filters .....	483
Programmatic Use: Example 1 .....	484
Programmatic Use: Example 2 .....	486
Browsing APIs .....	488
Using the UTL_CALL_STACK Package .....	489
DEPRECATE Pragma .....	490
Quiz .....	491
Summary .....	494
Practice 10: Overview .....	495
<b>Chapter 13: Lesson 11 - Profiling and Tracing PL/SQL Code .....</b>	<b>497</b>
Course Agenda .....	498
Objectives .....	499
Lesson Agenda .....	500
Tracing PL/SQL Execution .....	501
Tracing PL/SQL Execution (Notes Only) .....	504
Tracing PL/SQL: Steps .....	505
Step 1: Enable Specific Subprograms .....	506
Steps 2 and 3: Identify a Trace Level and Start Tracing .....	507
Step 4 and Step 5: Turn Off and Examine the Trace Data .....	508
plsql_trace_runs and plsql_trace_events .....	509
Lesson Agenda .....	511
Profiling PL/SQL Code .....	512
Hierarchical Profiling .....	513
Hierarchical Profiling Concepts .....	514
Using the PL/SQL Profiler .....	515

Understanding Raw Profiler Data .....	520
Using the Hierarchical Profiler Tables .....	521
Using DBMS_HPROF.ANALYZE .....	522
Using DBMS_HPROF.ANALYZE to Write to Hierarchical Profiler Tables .....	523
Analyzer Output from the DBMSHP_RUNS Table .....	524
Analyzer Output from the DBMSHP_FUNCTION_INFO Table .....	525
plshprof: A Simple HTML Report Generator .....	526
Using plshprof .....	527
Using the HTML Reports .....	530
Quiz .....	533
Summary .....	536
Practice 11: Overview .....	537
<b>Chapter 14: Lesson 12 - Securing Applications through PL/SQL .....</b>	<b>539</b>
Course Agenda .....	540
Objectives .....	541
Lesson Agenda .....	542
Invoker's Rights and Definer's Rights .....	543
Why Invoker's Rights? .....	544
AUTHID clause .....	545
Lesson Agenda .....	546
White Lists .....	547
ACCESSIBLE BY Clause .....	548
Using ACCESSIBLE BY Clause in Packages .....	549
Lesson Agenda .....	550
What Is an Application Security Policy? .....	551
Implementing Application Security Policy .....	552
DBMS_RLS package .....	553
Defining a Policy .....	554
Defining a Policy Function .....	555
Defining a Policy .....	556
Lesson Agenda .....	557
Application Context - Concept .....	558
Application Context - Implementation .....	559
USERENV Application Context .....	560
Creating an Application Context .....	561
Setting a Context .....	562
Lesson Agenda .....	564
Virtual Private Database .....	565
Implementing a Virtual Private Database .....	567
Setting Up a Context .....	568
Creating the Package .....	569
Creating the Package (Notes Only) .....	570
Define the Security Policy .....	571
Setting Up the Logon Trigger .....	572
Policy in Action .....	573
Data Dictionary Views .....	574

Using the ALL_CONTEXT Dictionary View .....	575
Policy Groups .....	577
Quiz .....	578
Summary .....	581
Practice 12: Overview .....	582
<b>Chapter 15: Lesson 13 - Safeguarding Your Code Against SQL Injection Attacks .....</b>	<b>583</b>
Course Agenda .....	584
Objectives .....	585
Lesson Agenda .....	586
SQL injection .....	587
SQL Injection: Example .....	588
Scenario .....	589
Types of SQL Injection .....	590
Avoidance Strategies Against SQL Injection .....	591
Protecting Against SQL Injection: Example .....	592
Lesson Agenda .....	593
Reducing the Attack Surface .....	594
Reducing the Attack Surface (Notes Only) .....	595
Expose the Database Only Via PL/SQL API .....	596
Using Invoker's Rights .....	597
Strengthen Database Security .....	599
Lesson Agenda .....	600
Using Static SQL .....	601
Using Dynamic SQL .....	604
Lesson Agenda .....	605
Using Bind Arguments with Dynamic SQL .....	606
Using Bind Arguments with Dynamic PL/SQL .....	607
What if You Cannot Use Bind Arguments? .....	608
Lesson Agenda .....	609
DBMS_ASSERT Package .....	610
Understanding DBMS_ASSERT .....	611
Oracle Identifiers .....	612
Working with Identifiers in Dynamic SQL .....	614
Choosing a Verification Route .....	615
Validate Input Using DBMS_ASSERT .....	616
Avoiding Injection by Using DBMS_ASSERT.SIMPLE_SQL_NAME .....	617
DBMS_ASSERT Guidelines .....	619
Quiz .....	623
Summary .....	627
Practice 13: Overview .....	628
<b>Chapter 16: Lesson 14 - Advanced Security Mechanisms .....</b>	<b>629</b>
Course Agenda .....	630
Objectives .....	631
Lesson Agenda .....	632
Real Application Security .....	633
How It Works Without RAS? .....	634

How It Works with RAS? .....	635
Real Application Security - Components .....	636
Implementing a RAS Data Security Policy .....	637
Application Sessions in RAS .....	638
RAS Sessions .....	639
Lesson Agenda .....	640
Transparent Data Encryption .....	641
Encrypting a Table Column Using TDE .....	642
Encrypting a Tablespace Using TDE .....	643
Keystores in TDE .....	644
Lesson Agenda .....	645
Oracle Data Redaction .....	646
Data Redaction Methods .....	647
Benefits of Data Redaction .....	648
Summary .....	649



10

# Creating Compound, DDL, and Event Database Triggers

ORACLE

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

## Objectives

After completing this lesson, you should be able to:

- Describe compound triggers
- Describe mutating tables
- Create triggers on DDL statements
- Create triggers on system events
- Display information about triggers



# Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

**Unit 2: Working with Triggers**

Unit 3: Working with the PL/SQL Code

▶ Lesson 9: Creating Triggers

▶ Lesson 10: Creating Compound, DDL, and Event Database Triggers

ORACLE

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

## Lesson Agenda

- Create compound DML triggers
- Resolve mutating table errors
- Schema triggers
- Database triggers
- Guidelines for designing triggers



## What is a Compound Trigger?

- A single trigger on a table that allows you to specify actions for each of the following four timing points:
  - BEFORE
  - BEFORE EACH ROW
  - AFTER EACH ROW
  - AFTER
- Each timing instance has an executable part and an optional exception handling section.
- All the timing instances access a common PL/SQL state.

ORACLE

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

A compound trigger is a single trigger on a table that allows you to specify actions for each of the four triggering timing points:

- Before the triggering statement
- Before each row that the triggering statement affects
- After each row that the triggering statement affects
- After the triggering statement

You use a compound trigger when you want all the timing instances to access a common PL/SQL state. The common PL/SQL state is established when the triggering statement starts and is destroyed when the triggering statement completes.

However, in some situations where you may have to write DML statements in the trigger body, you can use compound triggers.

## Working with Compound Triggers

- The compound trigger body supports a common PL/SQL state that the code for each timing point can access.
- The compound trigger common state is:
  - Established when the triggering statement starts
  - Destroyed when the triggering statement completes
- A compound trigger has a declaration section and a section for each of its timing points.
- The section of each timing point may have an optional exception handling section



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

## Why Compound Triggers?

You can use compound triggers to:

- Program an approach where you want the actions you implement for the various timing points to share common data
- Accumulate rows destined for a second table so that you can periodically bulk-insert them
- Avoid the mutating-table error (ORA-04091)



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

The current PL/SQL state is stored before starting the execution of the trigger and destroyed after completing the execution of the trigger. All the trigger timing points can access the same PL/SQL state in such scenario.

If you are inserting values into a second table while executing the trigger body, compound triggers allow you to accumulate all the newly inserted rows and bulk insert them into the table at periodic intervals. Thus improving the performance of the PL/SQL block.

What is ORA-4091?

Consider a trigger, you have defined for event BEFORE INSERT on a table T. In the trigger body definition, you execute a DML statement on the same table T, such action would result in ORA-4091 : table T mutating, trigger/function may not see it. The trigger execution here is initiated to modify the table. When you try to modify it again in the trigger body, it might lead to an inconsistent state, thereby giving the ORA-4091 error.

## Compound Trigger Structure

```
CREATE OR REPLACE TRIGGER schema.trigger
FOR dml_event_clause ON schema.table
COMPOUND TRIGGER
[ declare_section ]
timing_point_section [ timing_point_section ]... END [ trigger ] ;
```

The slide illustrates the structure of a compound trigger. The difference between a simple trigger and a compound trigger syntax is the `COMPOUND TRIGGER` clause.

The optional declarative part of a compound trigger declares variables and subprograms that all of its timing-point sections can use. When the trigger fires, the declarative part runs before any timing-point sections run, thus creating the PL/SQL state of the trigger. The variables and subprograms exist for the duration of the triggering statement.

The compound triggers further have the timing point sections for the timing points – BEFORE , BEFORE FOR EACH ROW, AFTER, AFTER FOR EACH ROW.

## Compound Trigger Structure for Views

```
CREATE trigger FOR dml_event_clause ON view
COMPOUND TRIGGER
INSTEAD OF EACH ROW IS
BEGIN
statement;
END INSTEAD OF EACH ROW;
```

A compound trigger defined on a view has only an `INSTEAD OF` timing point section defined in the trigger body.

## Compound Trigger Restrictions

- A compound trigger must be a DML trigger defined on either a table or a view.
- An exception that occurs in one section must be handled in that section. It cannot transfer control to another section.
- A timing-point section cannot handle exceptions raised in another timing-point section.
- `:OLD` and `:NEW` cannot appear in the declaration, BEFORE STATEMENT, or the AFTER STATEMENT sections.
- Only the BEFORE EACH ROW section can change the value of `:NEW`.



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

## Lesson Agenda

- Create compound DML triggers
- Resolve mutating table errors
- Schema triggers
- Database triggers
- Guidelines for designing triggers



# Mutating Tables

- A mutating table is:
  - A table that is being modified by an UPDATE, DELETE, or INSERT statement or
  - A table that might be updated by the effects of a DELETE CASCADE constraint
- The session that issued the triggering statement cannot query or modify a mutating table.
- This restriction prevents a trigger from seeing inconsistent data.
- This restriction applies to all triggers that use the FOR EACH ROW clause.
- Views being modified in the INSTEAD OF triggers are not considered mutating.

ORACLE

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

## Mutating Table: Example

```
CREATE OR REPLACE TRIGGER check_salary
BEFORE INSERT OR UPDATE OF salary, job_id
ON employees
FOR EACH ROW
WHEN (NEW.job_id <> 'AD_PRES')
DECLARE
    v_minsalary employees.salary%TYPE;
    v_maxsalary employees.salary%TYPE;
BEGIN
    SELECT MIN(salary), MAX(salary)
    INTO v_minsalary, v_maxsalary
    FROM employees
    WHERE job_id = :NEW.job_id;
    IF :NEW.salary < v_minsalary OR :NEW.salary > v_maxsalary THEN
        RAISE_APPLICATION_ERROR(-20505,'Out of range');
    END IF;
END;
/
```



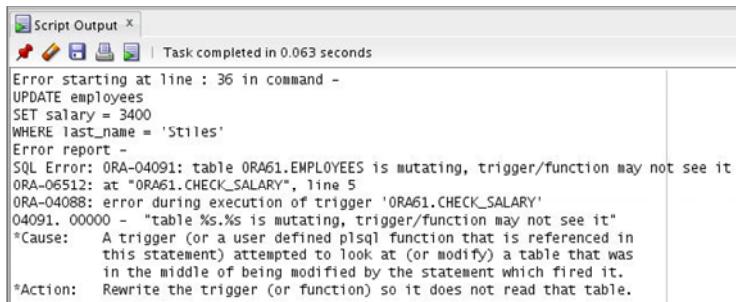
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The CHECK\_SALARY trigger in the slide example attempts to guarantee that whenever a new employee is added to the EMPLOYEES table or whenever an existing employee's salary or job ID is changed, the employee's salary falls within the established salary range for the employee's job.

When an employee record is updated, the CHECK\_SALARY trigger is fired for each row that is updated. The trigger code queries the same table that is being updated. Therefore, it is said that the EMPLOYEES table is a mutating table.

## Mutating Table: Example

```
UPDATE employees  
SET salary = 3400  
WHERE last_name = 'Stiles';
```



The screenshot shows a 'Script Output' window from Oracle SQL Developer. The command entered was:

```
UPDATE employees  
SET salary = 3400  
WHERE last_name = 'Stiles';
```

The output shows the following error message:

```
Error starting at line : 36 in command -  
UPDATE employees  
SET salary = 3400  
WHERE last_name = 'Stiles'  
Error report -  
SQL Error: ORA-04091: table ORA61.EMPLOYEES is mutating, trigger/function may not see it  
ORA-06512: at "ORA61.CHECK_SALARY", line 5  
ORA-04088: error during execution of trigger 'ORA61.CHECK_SALARY'  
04091. 00000 - "table %s.%s is mutating, trigger/function may not see it"  
*Cause: A trigger (or a user defined plsql function that is referenced in  
this statement) attempted to look at (or modify) a table that was  
in the middle of being modified by the statement which fired it.  
*Action: Rewrite the trigger (or function) so it does not read that table.
```

ORACLE

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

In the example in the slide, the trigger code tries to read or select data from a mutating table.

If you restrict the salary within a range between the minimum existing value and the maximum existing value, then you get a runtime error. The EMPLOYEES table is mutating or is in a state of change; therefore, the trigger cannot read from it.

Remember that functions can also cause a mutating table error when they are invoked in a DML statement.

### Possible Solutions

Possible solutions to this mutating table problem include the following:

- Use a compound trigger as described earlier in this lesson.
- Store the summary data (the minimum salaries and the maximum salaries) in another summary table, which is kept up-to-date with other DML triggers.
- Store the summary data in a PL/SQL package, and access the data from the package. This can be done in a BEFORE statement trigger.

# Using a Compound Trigger to Resolve the Mutating Table Error

```
CREATE OR REPLACE TRIGGER check_salary
FOR INSERT OR UPDATE OF salary, job_id
ON employees
WHEN (NEW.job_id <> 'AD_PRES')
COMPOUND TRIGGER

TYPE salaries_t          IS TABLE OF employees.salary%TYPE;
min_salaries            salaries_t;
max_salaries            salaries_t;

TYPE department_ids_t   IS TABLE OF employees.department_id%TYPE;
department_ids          department_ids_t;

TYPE department_salaries_t IS TABLE OF employees.salary%TYPE
                           INDEX BY VARCHAR2(80);
department_min_salaries department_salaries_t;
department_max_salaries department_salaries_t;

-- example continues on next slide
```



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

# Using a Compound Trigger to Resolve the Mutating Table Error

```
BEFORE STATEMENT IS
BEGIN
  SELECT MIN(salary), MAX(salary), NVL(department_id, -1)
  BULK COLLECT INTO min_Salaries, max_salaries, department_ids
  FROM employees
  GROUP BY department_id;
  FOR j IN 1..department_ids.COUNT() LOOP
    department_min_salaries(department_ids(j)) := min_salaries(j);
    department_max_salaries(department_ids(j)) := max_salaries(j);
  END LOOP;
END BEFORE STATEMENT;
AFTER EACH ROW IS
BEGIN
  IF :NEW.salary < department_min_salaries(:NEW.department_id)
    OR :NEW.salary > department_max_salaries(:NEW.department_id) THEN
    RAISE_APPLICATION_ERROR(-20505,'New Salary is out of acceptable
                                range');
  END IF;
END AFTER EACH ROW;
END check_salary;
```



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

department\_ids is used to hold the department IDs. If the employee who earns the minimum or maximum salary does not have an assigned department, you use the NVL function to store –1 for the department id, instead of NULL.

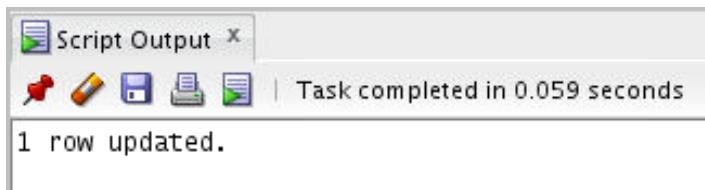
Next, you collect the minimum salary, maximum salary, and the department ID using a bulk insert into the min\_salaries, max\_salaries, and department\_ids respectively grouped by department ID. The select statement returns 13 rows. The values of the department\_ids are used as an index for the department\_min\_salaries and department\_max\_salaries tables.

Therefore, the index for those two tables (VARCHAR2) represents the actual department\_ids.

After each row is added, if the new salary is less than the minimum salary for that department or greater than the department's maximum salary, then an error message is displayed.

To test the newly created compound trigger, issue the following statement:

```
UPDATE employees
SET salary = 3400
WHERE last_name = 'Stiles';
```



To ensure that the salary for employee Stiles was updated, issue the following query by using the F9 key in SQL Developer:

```
SELECT employee_id, first_name,  
last_name, job_id, department_id,salary  
FROM employees  
WHERE last_name = 'Stiles';
```

The screenshot shows the 'Query Result' window in SQL Developer. The title bar says 'Query Result'. Below it, there are icons for Refresh, Print, Copy, and Paste, followed by 'SQL' and the message 'All Rows Fetched: 1 in 0.001 seconds'. The main area is a grid with the following data:

	EMPLOYEE_ID	FIRST_NAME	LAST_NAME	JOB_ID	DEPARTMENT_ID	SALARY
1	138	Stephen	Stiles	ST_CLERK	50	3400

## Lesson Agenda

- Create compound DML triggers
- Resolve mutating table errors
- Schema triggers
- Database triggers
- Guidelines for designing triggers



# Creating Triggers on DDL Statements

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE | AFTER -- Timing
[ddl_event1 [OR ddl_event2 OR ...]]
ON {DATABASE | SCHEMA}
trigger_body
```

Sample DDL Events	Fires When
CREATE	Any database object is created using the CREATE command.
ALTER	Any database object is altered using the ALTER command.
DROP	Any database object is dropped using the DROP command.



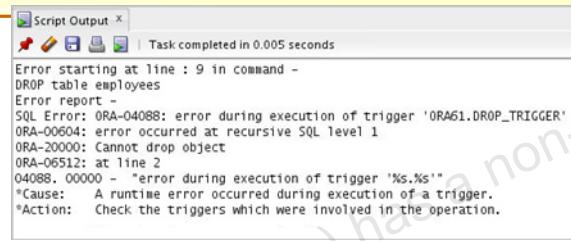
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Creating Triggers on DDL Statements -Example

```
CREATE OR REPLACE TRIGGER drop_trigger
BEFORE DROP ON ora61.SCHEMA
BEGIN
RAISE_APPLICATION_ERROR ( num => -20000, msg => 'Cannot drop object');
END;
/
```

To check the execution of this trigger

```
DROP TABLE employees;
```



ORACLE

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

## Lesson Agenda

- Create compound DML triggers
- Resolve mutating table errors
- Schema triggers
- Database triggers
- Guidelines for designing triggers



# Creating Database Triggers

- Triggering user event:
  - Logging on or off
- Triggering a database event:
  - Shutting down or starting up the database
  - A specific error (or any error) being raised



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

Before coding the trigger body, decide on the components of the trigger.

Triggers on system events can be defined at the database or schema level. For example, a database shutdown trigger is defined at the database level. Triggers on DDL statements, or a user logging on or off, can also be defined at either the database level or the schema level. Triggers on DML statements are defined on a specific table or a view.

A trigger defined at the database level fires for all users, whereas a trigger defined at the schema or table level fires only when the triggering event involves that schema or table.

# Creating Triggers on System Events

```
CREATE [OR REPLACE] TRIGGER trigger_name
BEFORE | AFTER -- timing
[database_event1 [OR database_event2 OR ...]]
ON {DATABASE | SCHEMA}
trigger_body
```

Database Event	Triggers Fires When
AFTER SERVERERROR	An Oracle error is raised
AFTER LOGON	A user logs on to the database
BEFORE LOGOFF	A user logs off the database
AFTER STARTUP	The database is opened
BEFORE SHUTDOWN	The database is shut down normally



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

## LOGON and LOGOFF Triggers: Example

```
-- Create the log_trig_table shown in the notes page
-- first
CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON ON SCHEMA
BEGIN
    INSERT INTO log_trig_table(user_id,log_date,action)
    VALUES (USER, SYSDATE, 'Logging on');
END;
/
```

```
CREATE OR REPLACE TRIGGER logoff_trig
BEFORE LOGOFF ON SCHEMA
BEGIN
    INSERT INTO log_trig_table(user_id,log_date,action)
    VALUES (USER, SYSDATE, 'Logging off');
END;
/
```



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

You can create these triggers to monitor how often you log on and off, or you may want to write a report that monitors the length of time for which you are logged on. When you specify ON SCHEMA, the trigger fires for the specific user. If you specify ON DATABASE, the trigger fires for all users.

The definition of the log\_trig\_table used in the slide examples is as follows:

```
CREATE TABLE log_trig_table(
    user_id  VARCHAR2(30),
    log_date TIMESTAMP,
    action   VARCHAR2(40))
/
```

To see the execution of these triggers you can disconnect from the database and connect with the database. Then execute

```
select * from log_trig_table;
```

You can see the output as:

USER_ID	LOG_DATE	ACTION
1 ORA61	06-SEP-16 11.57.39.000000000 PM	Logging off
2 ORA61	06-SEP-16 11.57.43.000000000 PM	Logging on

## Lesson Agenda

- Create compound DML triggers
- Resolve mutating table errors
- Schema triggers
- Database triggers
- Guidelines for designing triggers



ORACLE

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

## Guidelines for Designing Triggers

- You can design triggers to:
  - ensure that whenever a specific event occurs, any necessary actions are done.
- Don't create triggers:
  - which duplicate the function of the database
  - which are recursive
- You can create stored procedures and invoke them in a trigger, if the PL/SQL code is very lengthy.



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

- Use triggers to ensure that whenever a specific event occurs, any necessary actions are done.  
For example, use a trigger to ensure that whenever anyone updates a table, its log file is updated.
- Don't create triggers that duplicate database features.  
For example, don't create a trigger to reject invalid data if you can do the same with constraints.
- Don't create recursive triggers.  
For example, don't create an AFTER UPDATE trigger that issues an UPDATE statement on the table on which the trigger is defined. The trigger fires recursively until it runs out of memory.

## Quiz



Which of the following statements are true for a trigger?

- a. A trigger is defined with a CREATE TRIGGER statement.
- b. A trigger's source code is contained in the USER\_TRIGGERS data dictionary.
- c. A trigger is explicitly invoked.
- d. A trigger is implicitly invoked by DML.
- e. COMMIT, SAVEPOINT, and ROLLBACK are not allowed when working with a trigger.



ORACLE

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

## Summary

In this lesson, you should have learned how to:

- Describe compound triggers
- Describe mutating tables
- Create triggers on DDL statements
- Create triggers on system events
- Display information about triggers



ORACLE

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

## Practice 10 Overview: Creating Compound, DDL, and Event Database Triggers

This practice covers the following topics:

- Creating advanced triggers to manage data integrity rules
- Creating triggers that cause a mutating table exception
- Creating triggers that use package state to solve the mutating table problem



ORACLE

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

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

PELISSIER LOIC (loic.pelissier@gmail.com) has a non-transferable  
license to use this Student Guide.

13

# Managing Dependencies

ORACLE

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

## Objectives

After completing this lesson, you should be able to:

- Track procedural dependencies
- Predict the effect of changing a database object on procedures and functions
- Manage procedural dependencies
- Manage local and remote dependencies



ORACLE

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

This lesson introduces you to object dependencies, and implicit and explicit recompilation of invalid objects.

# Course Road Map

Lesson 1: Course Overview

Unit 1: Working with Subprograms

Unit 2: Working with Triggers

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

▶ Lesson 11: Design Considerations for the PL/SQL Code

▶ Lesson 12: Tuning the PL/SQL Compiler

▶ Lesson 13: Managing Dependencies

ORACLE

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

## Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



## What are Dependencies in a Schema?

- Dependency refers to a situation where one schema object depends on another object for its definition.

### Examples

- There is a dependency between a view and the set of tables on which it is defined.
- There is a dependency between the procedure and the tables on which the procedure is defined.

## How Dependencies Work?

Let us create the following views in the HR schema:

```
CREATE OR REPLACE VIEW commissioned AS SELECT first_name, last_name,  
commission_pct FROM employees WHERE commission_pct > 0.00;  
  
CREATE OR REPLACE VIEW emp_mails AS SELECT first_name, last_name, email FROM  
employees;
```

```
SELECT object_name, status  
FROM user_objects  
WHERE object_type = 'VIEW'  
ORDER BY object_name;
```

Query Result	
OBJECT_NAME	STATUS
1 COMMISSIONED	VALID
2 EMP_MAILS	VALID

ORACLE

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

In the slide, we create two views – **commissioned** and **emp\_mails** – on the table **employees**. Both the views are dependent on the schema object **employees** table.

There is a dependency between the table **employees**, and views **commissioned** and **emp\_mails**. The views are dependent on the table **employees**.

The view **commissioned** uses columns **first\_name**, **last\_name**, and **commission\_pct** columns of the **employees** table.

The view **emp\_mails** uses columns **first\_name**, **last\_name**, and **email** columns of the **employees** table.

## How Dependencies Work?

Let us modify the email column of the employees table:

```
ALTER TABLE employees MODIFY email VARCHAR2(100);
```

And check the state of the views:

```
SELECT object_name, status  
FROM user_objects  
WHERE object_type = 'VIEW'  
ORDER BY object_name;
```

OBJECT_NAME	STATUS
1 COMMISSIONED	VALID
2 EMP_MAILS	INVALID

You can see that after the email column in the employees table is modified, the status emp\_mails view has become invalid.



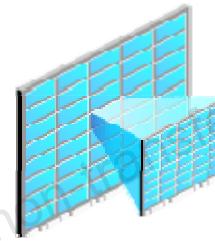
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Dependent and Referenced Objects

Some types of schema objects can reference other objects in their definitions.

For example:

- A view is defined by a query that references tables.
  - View is a dependent object.
  - Tables are referenced objects.



ORACLE

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

Some types of schema objects can reference other objects in their definitions. For example, a view is defined by a query that references tables or other views, and the body of a subprogram can include SQL statements that reference other objects. If the definition of object A references object B, then A is a dependent object (with respect to B) and B is a referenced object (with respect to A).

## Dependency Issues

- If you alter the definition of a referenced object, dependent objects may or may not continue to work properly. For example, if the table definition is changed, the procedure may or may not continue to work without error.
- The Oracle server automatically records dependencies among objects. To manage dependencies, all schema objects have a status (valid or invalid) that is recorded in the data dictionary and you can view the status in the `USER_OBJECTS` data dictionary view.
- If the status of a schema object is `VALID`, then the object has been compiled and can be immediately used when referenced.
- If the status of a schema object is `INVALID`, then the schema object must be compiled before it can be used.

The following table shows schema objects can be dependent, referenced, or both

Object Type	Can Be Dependent or Referenced
Package body	Dependent only
Package specification	Both
Sequence	Referenced only
Subprogram	Both
Synonym	Both
Table	Both
Trigger	Both
User-defined object	Both
User-defined collection	Both
View	Both

## Querying Object Dependencies: Using the USER\_DEPENDENCIES View

```
desc user_dependencies;
```

Name	Null	Type
NAME	NOT NULL	VARCHAR2(128)
TYPE		VARCHAR2(19)
REFERENCED_OWNER		VARCHAR2(128)
REFERENCED_NAME		VARCHAR2(128)
REFERENCED_TYPE		VARCHAR2(19)
REFERENCED_LINK_NAME		VARCHAR2(128)
SCHEMAD		NUMBER
DEPENDENCY_TYPE		VARCHAR2(4)

```
SELECT name, type, referenced_name, referenced_type  
FROM user_dependencies  
WHERE referenced_name IN ('EMPLOYEES', 'EMP_VW' );
```

NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE
1 SECURE_EMPLOYEES	TRIGGER	EMPLOYEES	TABLE
2 DML_CALL_SQL	FUNCTION	EMPLOYEES	TABLE
3 GET_EMP	FUNCTION	EMPLOYEES	TABLE
4 SECURE_EMP	TRIGGER	EMPLOYEES	TABLE
5 EMPLOYEE_SAL	PROCEDURE	EMPLOYEES	TABLE
6 EMP_MAILS	VIEW	EMPLOYEES	TABLE
7 UPDATE_JOB_HISTORY	TRIGGER	EMPLOYEES	TABLE
8 RAISE_SAL	PROCEDURE	EMPLOYEES	TABLE

ORACLE

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

You can determine dependencies in the schema from the USER\_DEPENDENCIES data dictionary view.

The ALL\_DEPENDENCIES and DBA\_DEPENDENCIES views contain the additional OWNER column, which references the owner of the object.

### The USER\_DEPENDENCIES Data Dictionary View Columns

The columns of the USER\_DEPENDENCIES data dictionary view are as follows:

- NAME: The name of the dependent object
- TYPE: The type of the dependent object (PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER, or VIEW)
- REFERENCED\_OWNER: The schema of the referenced object
- REFERENCED\_NAME: The name of the referenced object
- REFERENCED\_TYPE: The type of the referenced object
- REFERENCED\_LINK\_NAME: The database link used to access the referenced object
- SCHEMA\_ID: Opaque schema identifier (16 bytes)
- DEPENDENCY\_TYPE: Indicates whether the dependency is a REF dependency

## Querying an Object's Status

Every database object has one of the following status values:

Status	Description
VALID	The object was successfully compiled by using the current definition in the data dictionary.
COMPILED WITH ERRORS	The most recent attempt to compile the object produced errors.
INVALID	The object is marked invalid because an object that it references has changed. (Only a dependent object can be invalid.)
UNAUTHORIZED	An access privilege on a referenced object was revoked. (Only a dependent object can be unauthorized.)



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

Every database object has one of the status values shown in the table in the slide.

**Note:** The `USER_OBJECTS`, `ALL_OBJECTS`, and `DBA_OBJECTS` static data dictionary views do not distinguish between Compiled with errors, Invalid, and Unauthorized; instead, they describe all these as `INVALID`.

## Categorizing Dependencies

- Local dependencies – Both the referenced and dependent objects are on the same database. Oracle Database manages them implicitly. There are two types of dependencies in local dependencies using the internal tables.
  - Direct dependencies
  - Indirect dependencies
- Remote dependencies – The referenced and dependent objects are on different nodes across the network. Oracle Database uses different mechanisms to manage remote dependencies, depending on the objects involved.

ORACLE

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

## Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



ORACLE

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

## Direct Dependencies

- When a view V is defined on a table T, there is a direct dependency of the view V on the table T.
- Direct dependents are invalidated only by changes to the referenced objects that affect them.

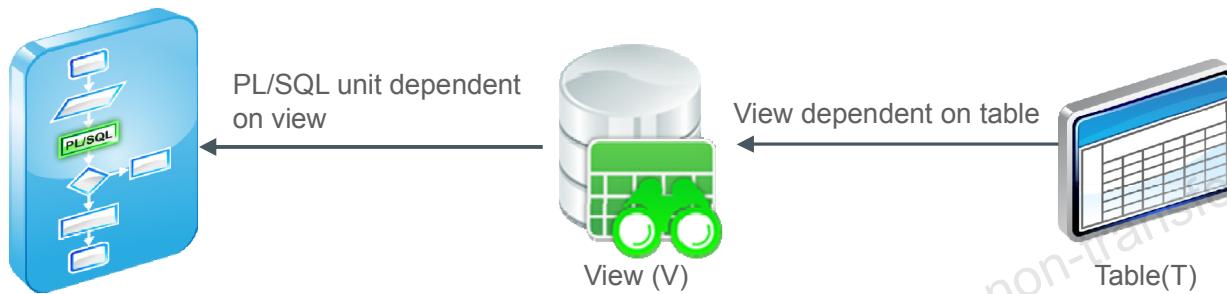


ORACLE

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

## Indirect Dependencies

- Consider a PL/SQL program unit dependent on the view V, which in turn is dependent on the table T. The PL/SQL program unit is indirectly dependent on the table T.
- Indirect dependents can be invalidated by changes to the referenced object that do not affect them.
- The PL/SQL unit may be invalidated if there is a modification to the table.



ORACLE

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

The slide shows an indirect dependency between the PL/SQL unit and the table. So, the PL/SQL unit is an indirect dependent on the table.

Indirect dependents can be invalidated by changes to the reference object that do not affect them. A modification to the table may invalidate the PL/SQL unit. This is called cascading invalidation.

## Displaying Direct and Indirect Dependencies

- Run the `utldtree.sql` script that creates the objects that enable you to display the direct and indirect dependencies.

```
@/home/oracle/labs/plpu/labs/utldtree.sql
```

- This script creates four objects, as given below:
  - A table `deptree temptab` to hold dependency data
  - A procedure `deptree_fill` to populate the table
  - Two views – `deptree` and `ideptree` – to select and format dependency data from the populated table



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

## Displaying Direct and Indirect Dependencies by Using Views Provided by Oracle

You can display direct and indirect dependencies from additional user views, called `DEPTREE` and `IDEPTREE`; these views are provided by Oracle.

### Example

- Make sure that the `utldtree.sql` script has been executed. This script is located in the `$ORACLE_HOME/labs/plpu/labs` folder. You can run the script as follows:

```
@?/labs/plpu/labs/utldtree.sql
```

**Note:** In this class, this script is supplied in the `labs` folder of your class files. The code example above uses the student account `ORA61`. (This applies to a Linux environment. If the file is not found, locate the file in your `labs` subdirectory.)

- Populate the `DEPTREE_TEMP TAB` table with information for a particular referenced object by invoking the `DEPTREE_FILL` procedure. There are three parameters for this procedure:

<code>object_type</code>	Type of the referenced object
<code>object_owner</code>	Schema of the referenced object
<code>object_name</code>	Name of the referenced object

## Displaying Direct and Indirect Dependencies

- Execute the DEPTREE\_FILL procedure.

```
EXECUTE deptree_fill('TABLE', 'ORA61', 'EMPLOYEES')
```

- Display the dependencies

```
SELECT nested_level, type, name  
FROM deptree  
ORDER BY seq#;
```

NESTED_LEVEL	TYPE	NAME
1	0 TABLE	EMPLOYEES
2	1 TRIGGER	SECURE_EMPLOYEES
3	1 TRIGGER	UPDATE_JOB_HISTORY
4	1 PROCEDURE	RAISE_SAL
5	1 PROCEDURE	DISPLAY_NEW_SAL
6	1 FUNCTION	GET_SAL
7	1 FUNCTION	TAX
8	1 FUNCTION	EMP_TAX

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



You can display a tabular representation of all dependent objects by querying the DEPTREE view. You can display an indented representation of the same information by querying the IDEPTREE view, which consists of a single column named DEPENDENCIES as follows:

```
SELECT *  
FROM ideptree;
```

DEPENDENCIES
1 PROCEDURE ORA61.RAISE_SAL
2 FUNCTION ORA61.DML_CALL_SQL
3 PROCEDURE ORA61.SAL_STATUS
4 PROCEDURE ORA61.RAISE_SALARY
5 VIEW ORA61.COMMISSIONED
6 FUNCTION ORA61.TAX
7 FUNCTION ORA61.EMP_TAX
8 VIEW ORA61.EMP_MAILS

...

## Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



# Fine-Grained Dependency Management

```
CREATE OR REPLACE VIEW commissioned AS SELECT first_name, last_name,
commission_pct FROM employees WHERE commission_pct > 0.00;
CREATE OR REPLACE VIEW emp_mails AS SELECT first_name, last_name, email FROM
employees;
```

```
SELECT object_name, status
FROM user_objects
WHERE object_type = 'VIEW'
ORDER BY object_name;
```

Query Result	
OBJECT_NAME	STATUS
1 COMMISSIONED	VALID
2 EMP_MAILS	VALID

```
ALTER TABLE employees MODIFY email VARCHAR2(100);
```

```
SELECT object_name, status
FROM user_objects
WHERE object_type = 'VIEW'
ORDER BY object_name;
```

Query Result	
OBJECT_NAME	STATUS
1 COMMISSIONED	VALID
2 EMP_MAILS	INVALID

ORACLE

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

Let us reconsider the scenario that we discussed earlier. We created two views, `commissioned` and `emp_mails`, on the `employees` table.

The `commissioned` view has the `first_name`, `last_name`, and `commission_pct` columns of the `employees` table.

The `emp_mails` view has the `first_name`, `last_name`, and `email` columns of the `employees` table.

When we altered the definition of the `email` column of the `employees` table, only the `emp_mails` view is invalidated. The `commissioned` view is still valid.

This process of invalidating specific objects based on their definition is fine-grained dependency management. Whenever a referenced object is modified, only those dependent objects are invalidated that are effected by the modification, and not all the dependent objects.

## Fine-Grained Dependency Management

- Starting with Oracle Database 11g, dependencies are now tracked at the level of element within unit.
- Element-based dependency tracking covers the following:
  - Dependency of a single-table view on its base table
  - Dependency of a PL/SQL program unit (package specification, package body, or subprogram) on the following:
    - Other PL/SQL program units
    - Tables
    - Views



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

# Fine-Grained Dependency Management: Example 1

```
CREATE TABLE t2 (col_a NUMBER, col_b NUMBER, col_c NUMBER);
CREATE VIEW v AS SELECT col_a, col_b FROM t2;
```

```
SELECT ud.name, ud.type, ud.referenced_name,
       ud.referenced_type, uo.status
  FROM user_dependencies ud, user_objects uo
 WHERE ud.name = uo.object_name AND ud.name = 'V';
```

Results:					
#	NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE	STATUS
1	V	VIEW	T2	TABLE	VALID

```
ALTER TABLE t2 ADD (col_d VARCHAR2(20));
```

```
SELECT ud.name, ud.type, ud.referenced_name,
       ud.referenced_type, uo.status
  FROM user_dependencies ud, user_objects uo
 WHERE ud.name = uo.object_name AND ud.name = 'V';
```

Results:					
#	NAME	TYPE	REFERENCED_NAME	REFERENCED_TYPE	STATUS
1	V	VIEW	T2	TABLE	VALID



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

## Example of Dependency of a Single-Table View on Its Base Table

In the first example in the slide, table T2 is created with three columns: COL\_A, COL\_B, and COL\_C. A view named V is created based on columns COL\_A and COL\_B of table T2. The dictionary views are queried and view V is dependent on table T and its status is valid.

In the third example, table T2 is altered. A new column named COL\_D is added. The dictionary views still report that view V is dependent because element-based dependency tracking grasps that columns COL\_A and COL\_B are not modified and, therefore, the view does not need to be invalidated.

## Fine-Grained Dependency Management: Example 1

```
ALTER TABLE t2 MODIFY (col_a VARCHAR2(20));
SELECT ud.name, ud.referenced_name, ud.referenced_type, uo.status
FROM user_dependencies ud, user_objects uo
WHERE ud.name = uo.object_name AND ud.name = 'V';
```

Results:

NAME	REFERENCED_NAME	REFERENCED_TYPE	STATUS
1 V	T2	TABLE	INVALID

ORACLE

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

## Fine-Grained Dependency Management: Example 2

```
CREATE OR REPLACE PACKAGE pkg IS
  PROCEDURE proc_1;
END pkg;
/
CREATE OR REPLACE PROCEDURE p IS
BEGIN
  pkg.proc_1();
END p;
/
CREATE OR REPLACE PACKAGE pkg
IS
  PROCEDURE proc_1;
  PROCEDURE unheard_of;
END pkg;
/
```

```
PACKAGE PKG compiled
PROCEDURE P compiled
PACKAGE PKG compiled
```



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

In the example in the slide, you create a package named `PKG` that has procedure `PROC_1` declared. A procedure named `P` invokes `PKG.PROC_1`.

The definition of the `PKG` package is modified and another subroutine is added to the package declaration.

When you query the `USER_OBJECTS` dictionary view for the status of the `P` procedure, it is still valid as shown because the element you added to the definition of `PKG` is not referenced through procedure `P`.

```
SELECT status FROM user_objects
WHERE object_name = 'P';
```

Results:	
	STATUS
	1 VALID

# Guidelines for Reducing Invalidation

To reduce invalidation of dependent objects:

- Add new items to the end of the package
- Reference each table through a view



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

## Add New Items to the End of the Package

When adding new items to a package, add them to the end of the package. This preserves the slot numbers and entry-point numbers of existing top-level package items, thereby preventing their invalidation. For example, consider the following package:

```
CREATE OR REPLACE PACKAGE pkg1 IS
FUNCTION get_var RETURN VARCHAR2;
PROCEDURE set_var (v VARCHAR2);
END;
```

Adding an item to the end of `pkg1` does not invalidate dependents that reference `get_var`. Inserting an item between the `get_var` function and the `set_var` procedure invalidates dependents that reference the `set_var` function.

## Reference Each Table Through a View

Reference tables indirectly by using views. This allows you to do the following:

- Add columns to the table without invalidating dependent views or dependent PL/SQL objects
- Modify or delete columns not referenced by the view without invalidating dependent objects

## Object Revalidation

- An object that is not valid when it is referenced must be validated before it can be used.
- Validation occurs automatically when an object is referenced; it does not require explicit user action.
- Revalidation may not happen automatically if:
  - The object is compiled with errors
  - The object is an unauthorized object
  - The object is an invalid object



ORACLE

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

The compiler cannot automatically revalidate an object that compiled with errors. The compiler recompiles the object, and if it recompiles without errors, it is revalidated; otherwise, it remains invalid.

The compiler checks whether the unauthorized object has access privileges to all of its referenced objects. If so, the compiler revalidates the unauthorized object without recompiling it. If not, the compiler issues appropriate error messages.

The SQL compiler recompiles the invalid object. If the object recompiles without errors, it is revalidated; otherwise, it remains invalid.

For an invalid PL/SQL program unit (procedure, function, or package), the PL/SQL compiler checks whether any referenced object is an invalid object.

- If so, the compiler recompiles the invalid object. If the object recompiles without errors, it is revalidated; otherwise, it remains invalid.
- If not, the compiler revalidates the invalid object without recompiling it. Fast revalidation is usually performed on objects that were invalidated due to cascading invalidation.

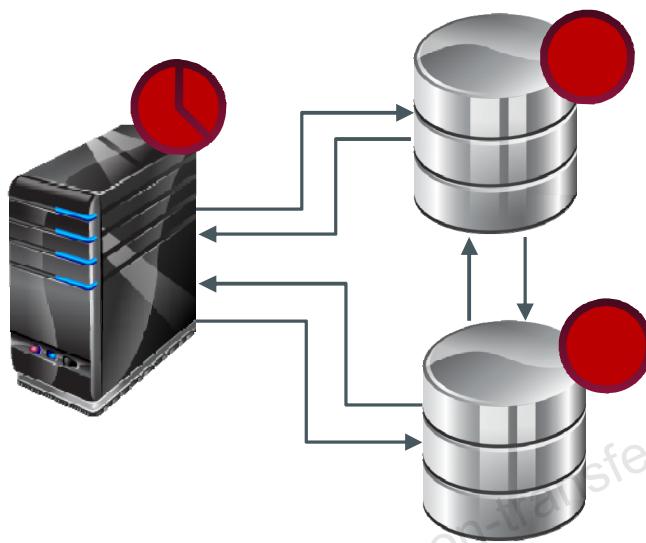
## Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



## Remote Dependencies

- The referenced and dependent objects are on different nodes across the network.
- Oracle Database does not manage dependencies among remote schema objects other than local-procedure-to-remote-procedure dependencies.



ORACLE

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

In the case of remote dependencies, the objects are on separate nodes. The local stored procedure and all its dependent objects are invalidated, but are automatically recompile when called for the first time. Oracle Database does not manage dependencies among remote schema objects other than local-procedure-to-remote-procedure dependencies.

For example, assume that a local view is created and defined by a query that references a remote table. Also assume that a local procedure includes a SQL statement that references the same remote table. Later, the definition of the table is altered.

As a result, the local view and procedure are never invalidated, even if the view or procedure is used after the table is altered and even if the view or procedure now returns errors when used. In this case, the view or procedure must be altered manually so that errors are not returned. In such cases, lack of dependency management is preferable to unnecessary recompilations of dependent objects.

### Recompilation of Dependent Objects: Local and Remote

- Verify successful explicit recompilation of the dependent remote procedures and implicit recompilation of the dependent local procedures by checking the status of these procedures within the `USER_OBJECTS` view.
- If an automatic implicit recompilation of the dependent local procedures fails, the status remains invalid and the Oracle server issues a runtime error. Therefore, to avoid disrupting production, it is strongly recommended that you recompile local dependent objects manually, rather than rely on an automatic mechanism.

# Managing Remote Procedure Dependencies

Mechanisms used for managing remote procedure dependencies are:

- Timestamp checking



- Signature checking



ORACLE

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

## **TIMESTAMP** Checking

Each PL/SQL program unit carries a time stamp that is set when it is created or recompiled. Whenever you alter a PL/SQL program unit or a relevant schema object, all its dependent program units are marked as invalid and must be recompiled before they can execute. The actual time stamp comparison occurs when a statement in the body of a local procedure calls a remote procedure.

## **SIGNATURE** Checking

For each PL/SQL program unit, both the time stamp and the signature are recorded. The signature of a PL/SQL construct contains information about the following:

- The name of the construct (procedure, function, or package)
- The base types of the parameters of the construct
- The modes of the parameters (IN, OUT, or IN OUT)
- The number of the parameters

The recorded time stamp in the calling program unit is compared with the current time stamp in the called remote program unit. If the time stamps match, the call proceeds. If they do not match, the remote procedure call layer performs a simple comparison of the signature to determine whether the call is safe or not. If the signature has not been changed in an incompatible manner, execution continues; otherwise, an error is returned.

## Setting the REMOTE\_DEPENDENCIES\_MODE Parameter

- You can define whether a remote procedure call should undergo Timestamp checking or Signature checking by setting the REMOTE\_DEPENDENCIES\_MODE parameter.
  - At the system level:  
ALTER SYSTEM SET REMOTE\_DEPENDENCIES\_MODE = TIMESTAMP | SIGNATURE
  - At the session level:  
ALTER SESSION SET REMOTE\_DEPENDENCIES\_MODE = TIMESTAMP | SIGNATURE
  - As an init.ora parameter:  
REMOTE\_DEPENDENCIES\_MODE = TIMESTAMP /  
SIGNATURE
- If the REMOTE\_DEPENDENCIES\_MODE parameter is not specified, either in the init.ora parameter file or by using the ALTER SESSION or ALTER SYSTEM DDL statements, TIMESTAMP is the default value.



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

## Timestamp Checking



Remote procedure B:  
compiles and is VALID  
at 8:00 AM

ORACLE

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

### Local Procedures Referencing Remote Procedures

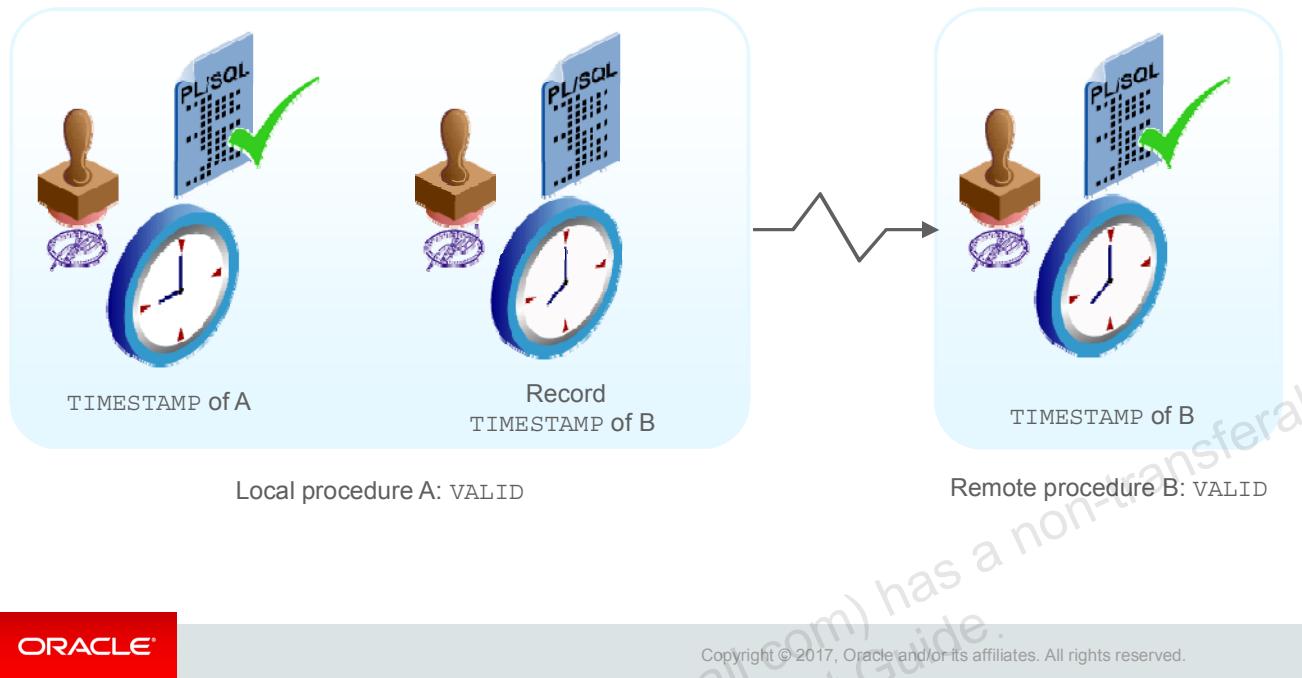
A local procedure that references a remote procedure is invalidated by the Oracle server if the remote procedure is recompiled after the local procedure is compiled.

#### Automatic Remote Dependency Mechanism

When a procedure compiles, the Oracle server records the time stamp of that compilation within the P code of the procedure.

In the slide, when the remote procedure B is successfully compiled at 8:00 AM, this time is recorded as its time stamp.

## Timestamp Checking



ORACLE

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

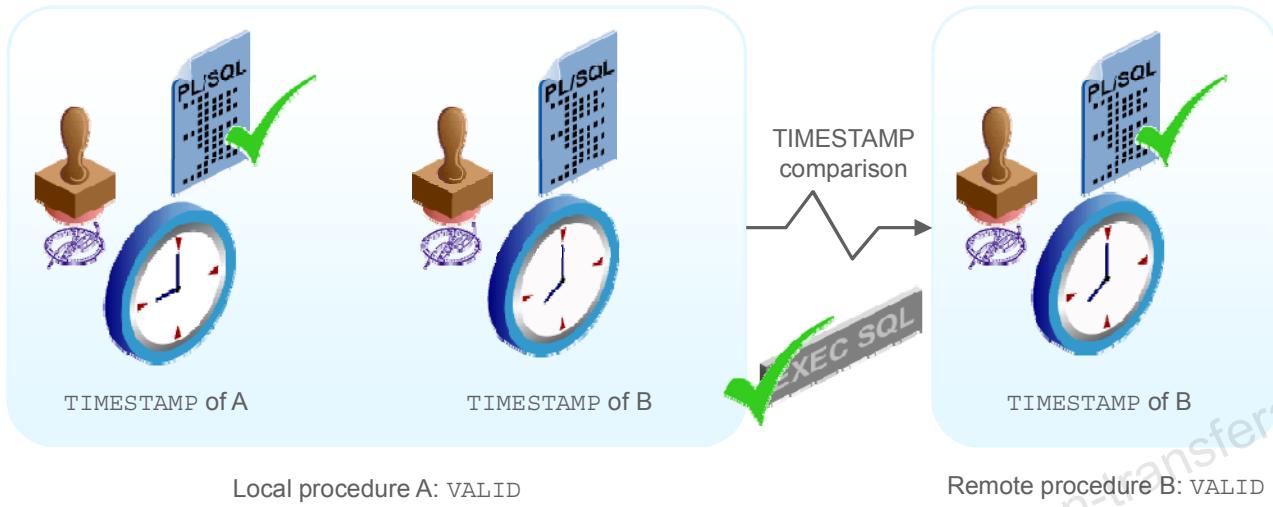
## Local Procedures Referencing Remote Procedures

### Automatic Remote Dependency Mechanism

When a local procedure referencing a remote procedure compiles, the Oracle server also records the time stamp of the remote procedure in the P code of the local procedure.

In the slide, local procedure A (which is dependent on remote procedure B) is compiled at 9:00 AM. The time stamps of both procedure A and remote procedure B are recorded in the P code of procedure A.

## Timestamp Checking



ORACLE

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

## Timestamp Checking



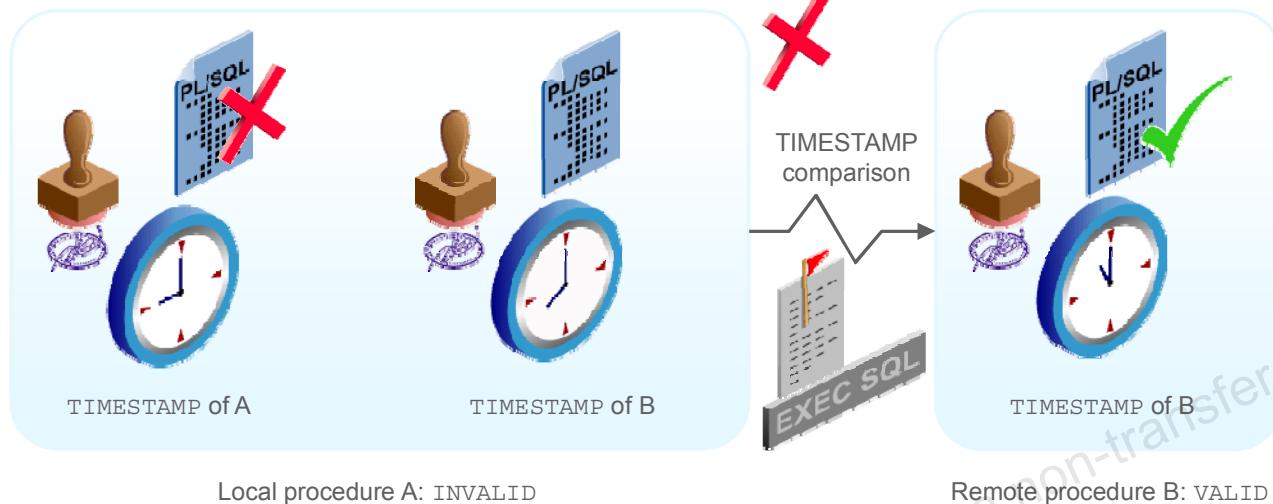
Remote procedure B:  
Recompiles and is VALID  
at 11:00 AM

ORACLE

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

# Timestamp Checking

Saved TIMESTAMP of B != COMPILE TIME of B



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

## Automatic Remote Dependency

If the time stamps are not equal (indicating that the remote procedure has recompiled), then the Oracle server invalidates the local procedure and returns a runtime error. If the local procedure (which is now tagged as invalid) is invoked a second time, then the Oracle server recompiles it before executing, in accordance with the automatic local dependency mechanism.

**Note:** If a local procedure returns a runtime error the first time it is invoked (indicating that the remote procedure's time stamp has changed), then you should develop a strategy to reinvoke the local procedure.

In the example in the slide, the remote procedure is recompiled at 11:00 AM and this time is recorded as its time stamp in the P code. The P code of local procedure A still has 8:00 AM as the time stamp for remote procedure B. Because the time stamp recorded with the P code of local procedure A is different from that recorded with the remote procedure B, the local procedure is marked invalid. When the local procedure is invoked for the second time, it can be successfully compiled and marked valid.

A disadvantage of the time stamp mode is that it is unnecessarily restrictive. Recompilation of dependent objects across the network is often performed when not strictly necessary, leading to performance degradation.

## Signature Checking

- The signature of a procedure is:
  - The name of the procedure
  - The data types of the parameters
  - The modes of the parameters
- The signature of the remote procedure is saved in the local procedure.
- When executing a dependent procedure, the signature of the referenced remote procedure is compared.



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

To alleviate some of the problems with the time stamp-only dependency model, you can use the signature model. This allows the remote procedure to be recompiled without affecting the local procedures. This is important if the database is distributed.

The signature of a subprogram contains the following information:

- The name of the subprogram
- The data types of the parameters
- The modes of the parameters
- The number of parameters
- The data type of the return value for a function

If a remote program is changed and recompiled but the signature does not change, then the local procedure can execute the remote procedure. With the time stamp method, an error would have been raised because the time stamps would not have matched.

## Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



## Revalidating PL/SQL Program Units

You revalidate a PL/SQL program unit by recompiling it. There are two ways of recompiling:

- Implicit runtime recompilation
  - Local dependencies are resolved implicitly
- Explicit recompilation with the `ALTER` statement
  - Here is the syntax for using the `ALTER` statement.

```
ALTER PROCEDURE | FUNCTION | TRIGGER | PACKAGE  
[SCHEMA.] object_name COMPILE;
```



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

## Unsuccessful Recompilation

Recompiling-dependent procedures and functions are unsuccessful when:

- The referenced objects are compiled with errors
- The referenced objects are dropped or renamed
- The data type of the referenced column is changed
- The referenced column is dropped
- A referenced view is replaced by a view with different columns
- The parameter list of a referenced procedure is modified



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

## Successful Recompilation

Recompiling-dependent procedures and functions are successful if:

- The referenced table has new columns
- The data type of referenced columns has not changed
- A private table is dropped, but a public table that has the same name and structure exists
- The PL/SQL body of a referenced procedure has been modified and recompiled successfully



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

The recompilation of dependent objects is successful if:

- New columns are added to a referenced table
- All `INSERT` statements include a column list
- No new column is defined as `NOT NULL`

When a private table is referenced by a dependent procedure and the private table is dropped, the status of the dependent procedure would become invalid. When the procedure is recompiled (either explicitly or implicitly) and a public table exists, the procedure can recompile successfully but is now dependent on the public table. The recompilation is successful only if the public table contains the columns that the procedure requires; otherwise, the status of the procedure remains invalid.

## Recompiling Procedures

Minimize dependency failures by:

- Declaring records with the `%ROWTYPE` attribute
- Declaring variables with the `%TYPE` attribute
- Querying with the `SELECT *` notation
- Including a column list with `INSERT` statements



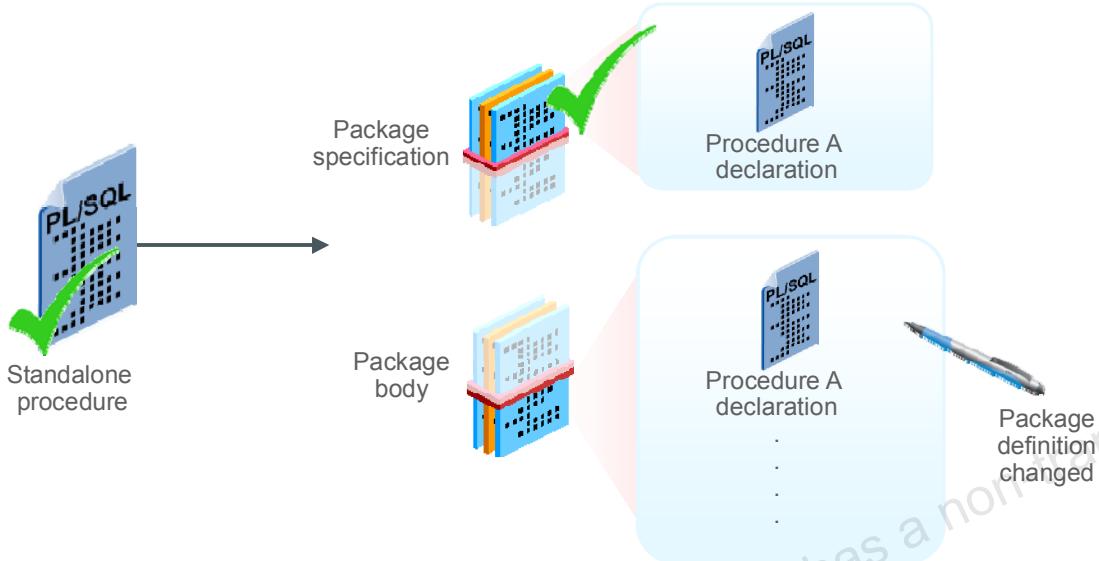
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Lesson Agenda

- Schema object dependencies
- Direct and indirect dependencies
- Fine-grained dependency management
- Managing remote dependencies
- Revalidating PL/SQL program units
- Package dependency management



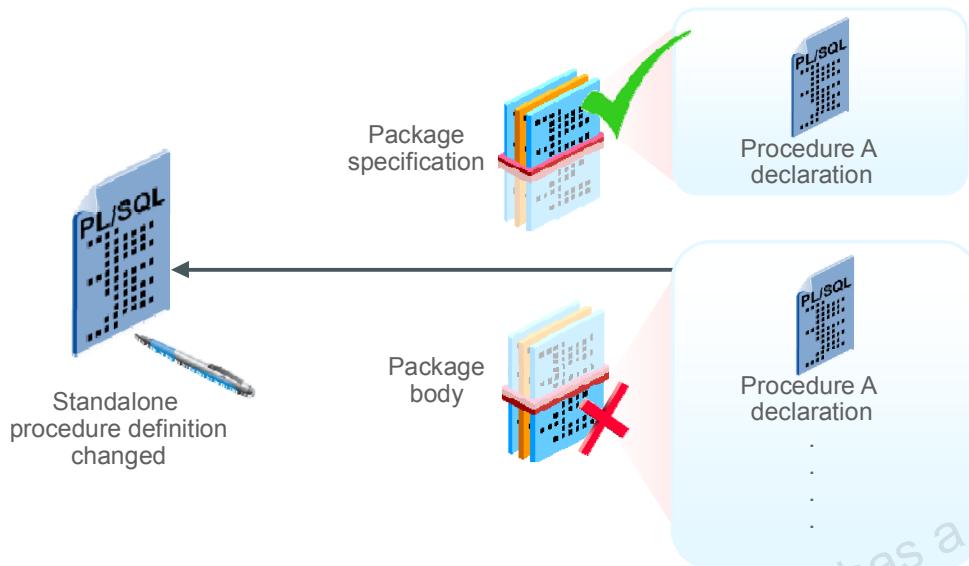
## Packages and Dependencies: Subprogram References the Package



ORACLE

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

## Packages and Dependencies: Package Subprogram References Procedure



ORACLE

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

# Quiz



You can display direct and indirect dependencies by running the `utldtree.sql` script, populating the `DEPTREE_TEMP TAB` table with information for a particular referenced object, and querying the `DEPTREE` or `IDEPTREE` views.

- a. True
- b. False



ORACLE

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

## Summary

In this lesson, you should have learned how to:

- Track procedural dependencies
- Predict the effect of changing a database object on procedures and functions
- Manage procedural dependencies
- Manage local and remote dependencies



ORACLE

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

## Practice 13 Overview: Managing Dependencies in Your Schema

This practice covers the following topics:

- Using `DEPTREE_FILL` and `IDEPTREE` to view dependencies
- Recompiling procedures, functions, and packages



ORACLE

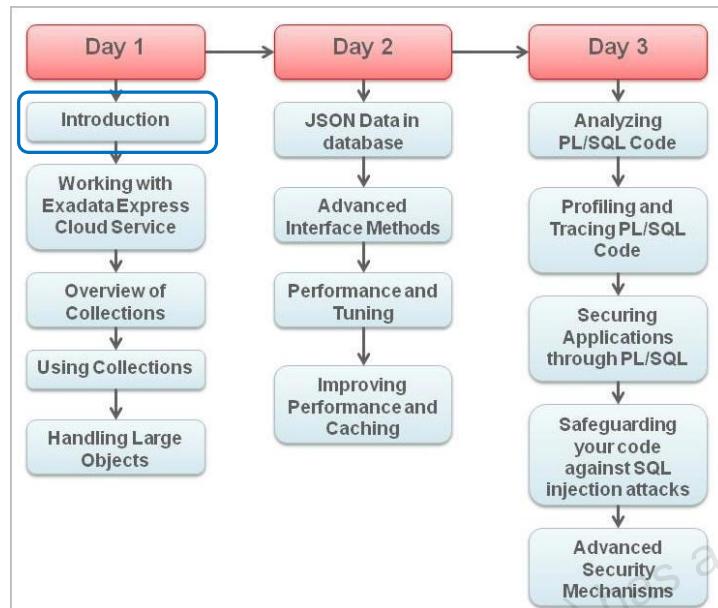
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

# Introduction

ORACLE

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

## Course Agenda



ORACLE®

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



## Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle database 12c and related products
- Oracle Cloud
- Oracle documentation and additional resources



ORACLE®

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

## Assumptions

- You have good knowledge of SQL and can write complex SQL queries.
- You have good knowledge of writing PL/SQL blocks of code.
- You understand terms like joins and nested queries.
- You have clear understanding of PL/SQL program constructs – loops, conditional statements and so on.
- You understand when there are references to terms such as cursors, procedures, functions and triggers.



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

This is an advanced course which assumes that you have good knowledge of SQL and PL/SQL concepts.

In this course you can see references to terms such as cursors, procedures, joins, indexes. It is assumed that you can write complex SQL queries using joins and nested queries. You should know how to write PL/SQL blocks both anonymous and named. It is assumed that you can write PL/SQL blocks using various PL/SQL constructs such as cursors, loops and so on.

## Course Objectives

After completing this course, you will be able to:

- Efficiently design PL/SQL packages and program units
- Write code to interface with external applications
- Create PL/SQL applications that can manipulate complex data
- Write and tune PL/SQL code effectively to maximize performance
- Understand the security requirements of application
- Manage PL/SQL code effectively



ORACLE®

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

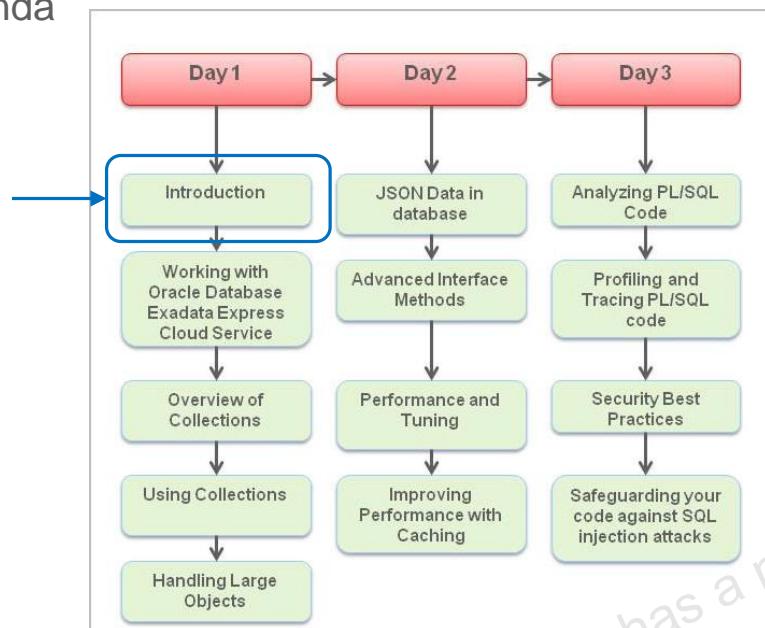
In this course, you will learn to handle complex data. You will learn about collections which enable you to work with a logical group of homogenous data. You will also learn to manage large objects and JSON data in the database.

Apart from handling complex data, you will learn how to interface database with other programming languages such as C and Java.

You will be introduced to utilities such as PL/Scope and Hierarchical profiler to analyze, trace and profile PL/SQL code. These functions help you in improving the performance of the application.

You will be introduced to various security mechanisms that can be used to secure applications.

## Course Agenda



ORACLE®

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

This course is completed over a duration of 3 days.

On day1, you will be introduced to the Oracle Database 12c new features, Exadata Express Cloud Service . You will be introduced to the concept of Collections and you will learn how to use collections and large objects in a PL/SQL block.

On day2, you will dive deeper into the course. You will learn to handle JSON data in Oracle database. You will also learn interfacing PL/SQL with Java and C languages. Towards the end of the day you will be learn about Performance tuning and caching techniques to improve application performance.

On day3, you will learn about analyzing, tracing and profiling PL/SQL code. These functions enable you to understand and tune application performance. You will be introduced to various security mechanisms available with Oracle database and discuss Virtual Private Database in detail. You will learn what is SQL injection and how to safeguard applications against SQL injection attacks.

## Appendices Used in This Course

- Appendix A: Table Descriptions and Data
- Appendix B: Using SQL Developer
- Appendix C: Using SQL\*Plus



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

There are three appendices in the course:

Appendix A discusses about the data in the tables of the schemas used in the course.

Appendix B discusses about SQL Developer. All practices in the course are demonstrated through SQL Developer. You execute the practices through SQL Developer.

Appendix C introduces you to the command line tool SQL Plus.

## Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle database 12c and related products
- Oracle Cloud
- Oracle documentation and additional resources



ORACLE®

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

## PL/SQL Development Environments

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

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



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

There are many tools that provide an environment for developing PL/SQL code. Some of the development tools that are available for use in this course are:

- **Oracle SQL Developer:** A graphical tool
- **Oracle SQL\*Plus:** A window or command-line application

**Note:** The code and screen examples presented in the course notes were generated from 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 your productivity and simplify the development 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.

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

## Specifications of SQL Developer

- Is developed in Java
- Supports 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

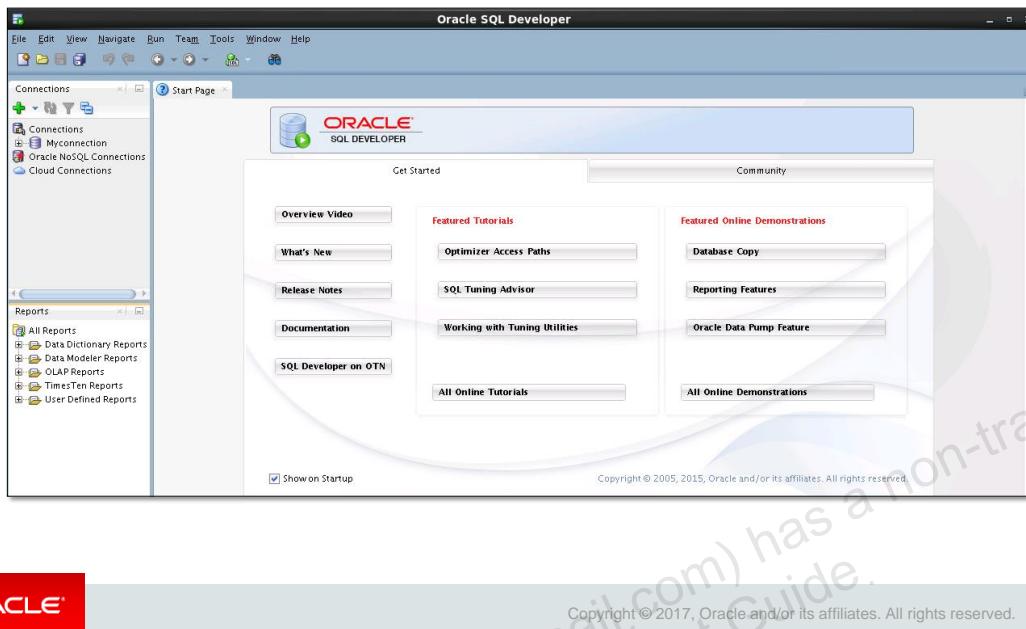


Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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, 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. You can connect to Oracle Database Service on cloud also through SQL Developer.

## SQL Developer 4.1.5 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.
- **Start page:** The Start page gives you some links which are helpful while using SQL Developer while creating 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 '+' in the Connections tab

## Coding PL/SQL in SQL\*Plus



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

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

Connected to:
Oracle Database 12c Enterprise Edition Release 12.1.0.0.2 - 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

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle database 12c and related products
- Oracle Cloud
- Oracle documentation and additional resources



ORACLE®

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

## Tables Used in This Course

- The sample schemas used are:
  - Order Entry (OE) schema
  - Human Resources (HR) schema
- Primarily, the OE schema is used.
- The OE schema user can read data in the HR schema tables.
- Appendix A contains more information about the sample schemas.



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

The sample company portrayed by Oracle Database Sample Schemas operates worldwide to fulfill orders for several different products. The company has several divisions:

- The Human Resources division tracks information about the employees and the facilities of the company.
- The Order Entry division tracks product inventories and sales of the company's products through various channels.
- The Sales History division tracks business statistics to facilitate business decisions. Although not used in this course, the SH schema is part of the "Example" sample schemas shipped with the database.

Each of these divisions is represented by a schema.

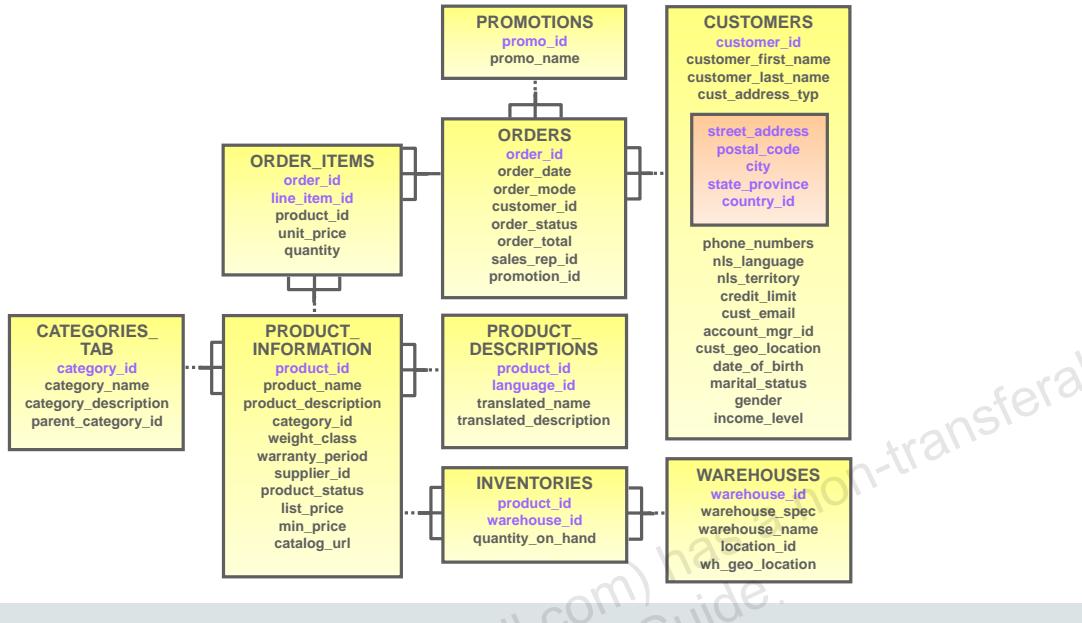
This course primarily uses the Order Entry (OE) sample schema.

**Note:** More details about the sample schema are found in Appendix A.

You have the HR schema installed by default. You can install OE schema and SH schema from Github. You can follow the instructions in database documentation.

<http://docs.oracle.com/database/122/COMSC/toc.htm>

## Order Entry Schema



ORACLE

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

The company sells several categories of products, including computer hardware and software, music, clothing, and tools. The company maintains product information that includes product identification numbers, the category into which the product falls, the weight group (for shipping purposes), the warranty period if applicable, the supplier, the status of the product, a list price, a minimum price at which a product will be sold, and a URL address for manufacturer information.

Inventory information is also recorded for all products, including the warehouse where the product is available and the quantity on hand. Because products are sold worldwide, the company maintains the names of the products and their descriptions in different languages.

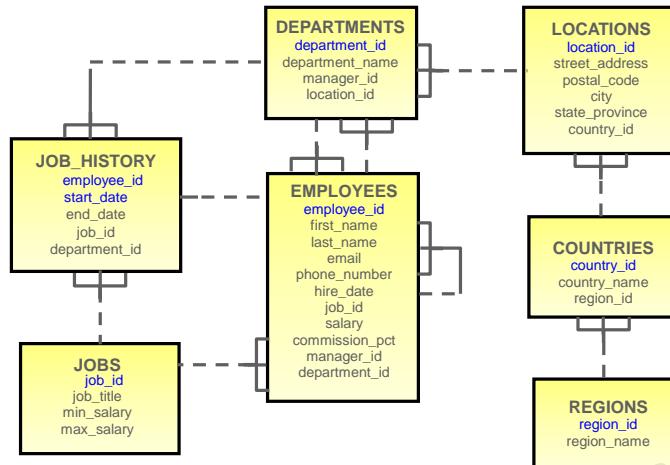
The company maintains warehouses in several locations to facilitate filling customer orders. Each warehouse has a warehouse identification number, name, and location identification number.

Customer information is tracked in some detail. Each customer is assigned an identification number. Customer records include name, street address, city or province, country, phone numbers (up to five phone numbers for each customer), and postal code. Some customers order through the Internet, so email addresses are also recorded. Because of language differences among customers, the company records the NLS language and territory of each customer. The company places a credit limit on its customers to limit the amount for which they can purchase at one time. Some customers have account managers, whom the company monitors. It keeps track of a customer's phone number. At present, you do not know how many phone numbers a customer might have, but you try to keep track of all of them. Because of the language differences among customers, you also identify the language and territory of each customer.

When a customer places an order, the company tracks the date of the order, the mode of the order, status, shipping mode, total amount of the order, and the sales representative who helped place the order. This may be the same individual as the account manager for a customer, it may be someone else, or, in the case of an order over the Internet, the sales representative is not recorded. In addition to the order information, the company also tracks the number of items ordered, the unit price, and the products ordered.

For each country in which it does business, the company records the country name, currency symbol, currency name, and the region where the country resides geographically. This data is useful to interact with customers who are living in different geographic regions of the world.

## Human Resources Schema



ORACLE

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

In the human resources records, each employee has an identification number, email address, job identification code, salary, and manager. Some employees earn a commission in addition to their salary.

The company also tracks information about the jobs within the organization. Each job has an identification code, job title, and a minimum and maximum salary range for the job. Some employees have been with the company for a long time and have held different positions within the company. When an employee switches jobs, the company records the start date and end date of the former job, the job identification number, and the department.

The sample company is regionally diverse, so it tracks the locations of not only its warehouses but also its departments. Each company employee is assigned to a department. Each department is identified by a unique department number and a short name. Each department is associated with one location. Each location has a full address that includes the street address, postal code, city, state or province, and country code.

For each location where it has facilities, the company records the country name, currency symbol, currency name, and the region where the country resides geographically.

**Note:** For more information about the “Example” sample schemas, refer to Appendix A.

## Lesson Agenda

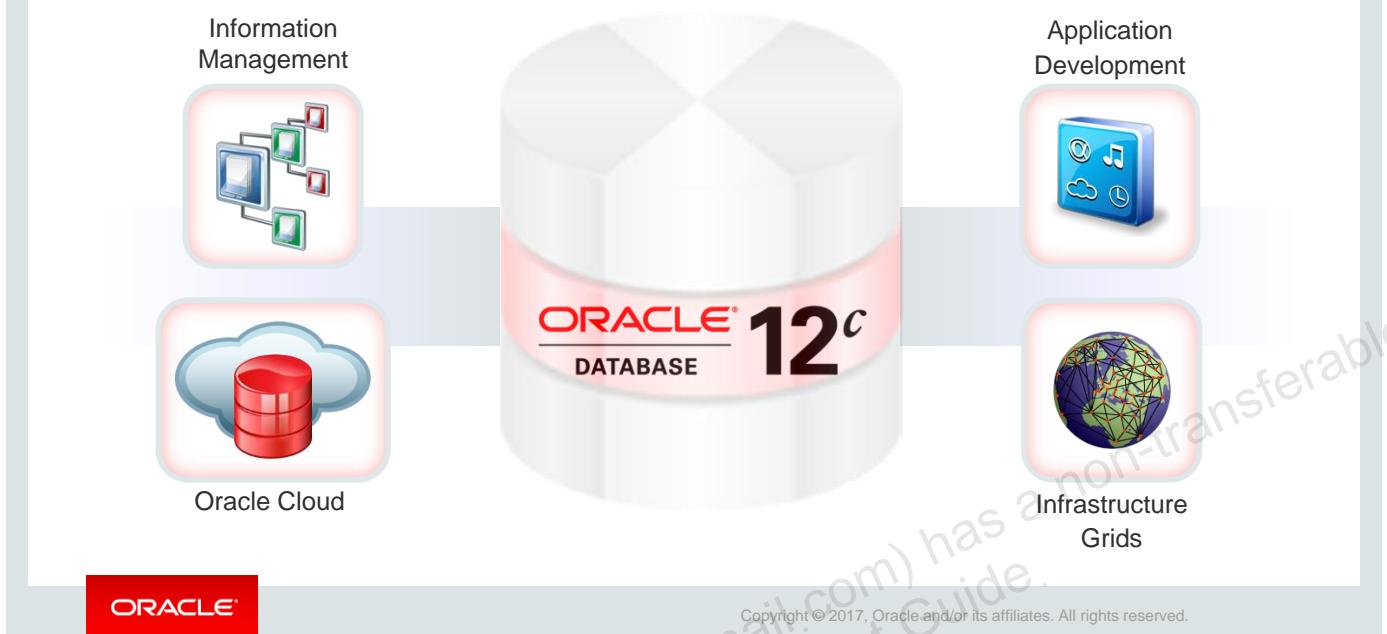
- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle database 12c and related products
- Oracle Cloud
- Oracle documentation and additional resources



ORACLE®

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

## Oracle Database 12c: Focus Areas



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



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

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle database 12c and related products
- Oracle Cloud
- Oracle documentation and additional resources



ORACLE®

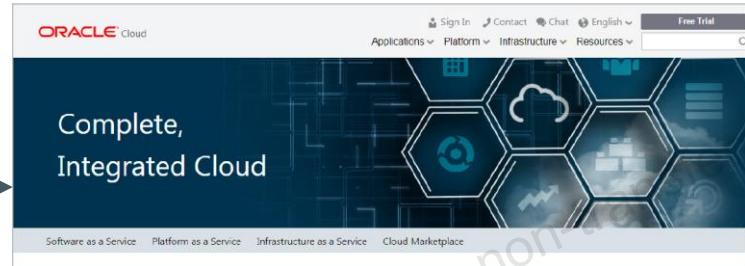
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## Introduction to Oracle Cloud

Oracle Cloud is an enterprise cloud for business. Oracle Public Cloud consists of many different services that share some common characteristics:

- On-demand self-service
- Resource pooling
- Rapid elasticity
- Measured service
- Broad network access

[cloud.oracle.com](http://cloud.oracle.com)



Oracle Cloud is an enterprise cloud for business. It provides an integrated collection of application and platform cloud services that are based on best-in-class products and open Java and SQL standards.

### Why Oracle Cloud?

The applications and databases that are deployed in Oracle Cloud are portable and can easily migrate from a private cloud or an on-premise (local machine) environment.

You can use the self-service interface to provision the cloud services and resources. These Cloud Services can be delivered onto Integrated Development and Deployment that enables the users to quickly extend and create new services.

Oracle Cloud services are built on the Oracle Exalogic Elastic Cloud and Oracle Exadata Database Machine, which together offer a platform that delivers extreme performance, redundancy, and scalability. The top two benefits of cloud computing are speed and cost.

The five essential characteristics are:

- **On-demand self-service:** Provisioning, monitoring, management control
- **Resource pooling:** Sharing and a level of abstraction between consumers and services
- **Rapid elasticity:** The ability to quickly scale up or down as needed
- **Measured service:** Metering utilization for either internal chargeback (private cloud) or external billing (public cloud)
- **Broad network access:** Typically, access through a browser on any networked device

## Oracle Cloud Services

Oracle Cloud provides the following 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.

- **What Is Software as Service (SaaS)?**

**SaaS** 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.

- **What Is Platform as a Service (PaaS)?**

**PaaS** 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.

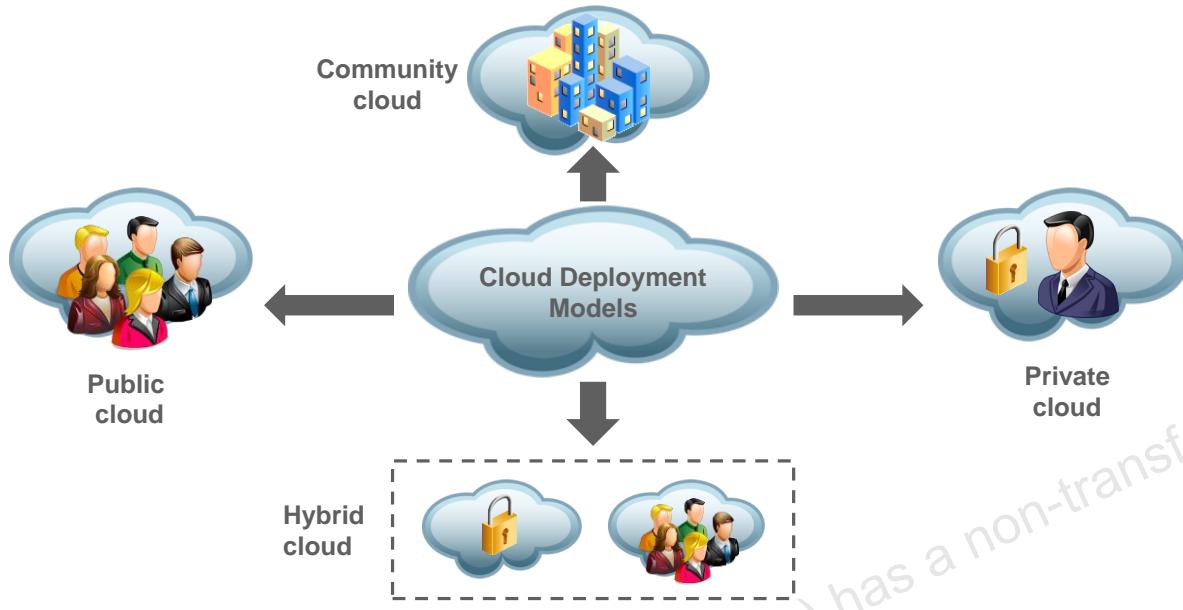
- **What Is Infrastructure as a Service (IaaS)?**

**IaaS** refers to computing hardware (servers, storage, and network) delivered as a service. This 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 database cloud is built within an enterprise's private cloud environment, as a PaaS model. It provides on-demand access to database services in a self-service, elastically scalable, and metered manner. Thus offering compelling advantages in cost, quality of service, and agility. A database can also be deployed within a virtual machine on an IaaS platform.

We can quickly deploy database clouds Oracle Exadata, a pre-integrated and optimized hardware platform that supports both online transaction processing (OLTP) and data warehouse (DW) workloads.

## Cloud Deployment Models



ORACLE®

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

- **What Is a Private Cloud?**  
A private cloud is generally used in private data centers , where a single organization hosts, manages, and controls private data centers or other resources. At times, the hosting is outsourced to a third-party service provider such as Amazon's Virtual Private Cloud.
- **What Is a Public Cloud?**  
Public cloud is accessed over a public network such as Internet. The resources provided by the cloud service provider are accessed by multiple users on shared basis. For example: Amazon Elastic Compute Cloud (EC2), IBM's Blue Cloud, Sun Cloud, Google AppEngine, and so on.
- **What is a 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.

- **What Is a 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

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle database 12c and related products
- Oracle Cloud
- Oracle documentation and additional resources



ORACLE®

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

## Oracle SQL and PL/SQL Documentation

- [Oracle Database New Features Guide](#)
- [Oracle Database Text Application Developer's Guide](#)
- [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](#)



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

## Summary

In this lesson, you should have learned how to:

- Describe the goals of the course
- Identify the environments that can be used in this course
- Describe the database schema and tables that are used in the course
- List the available documentation and resources



ORACLE

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

In this lesson, you were introduced to the goals of the course, the SQL Developer and SQL\*Plus environments used in the course, and the database schema and tables used in the lectures and practices.

## Practice 1 Overview: Getting Started

This practice covers the following topics:

- Reviewing the available SQL Developer resources
- Starting SQL Developer, and creating new database connections and browsing the HR, OE, and SH tables
- Executing SQL statements and an anonymous PL/SQL block by using SQL Worksheet
- Accessing and bookmarking the Oracle Database documentation and other useful websites



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

In this practice, you use SQL Developer to execute SQL statements for examining the data in the “Example” sample schemas: HR, OE, and SH. You also create a simple anonymous block. Optionally, you can experiment by creating and executing the PL/SQL code in SQL\*Plus.

**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.

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

PELISSIER LOIC (loic.pelissier@gmail.com) has a non-transferable  
license to use this Student Guide.

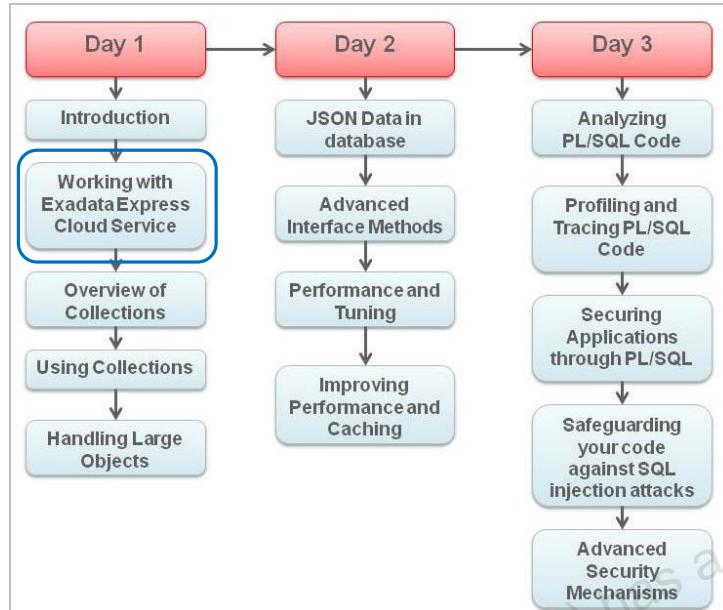
2

# Oracle Database Exadata Express Cloud Service



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

## Course Agenda



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:

- Define Oracle Exadata Express Cloud Service
- List the features of Oracle Exadata Express Cloud Service
- Discuss the service console and its components of Oracle Exadata Express Cloud Service
- Identify the different database clients that can be used for connecting to Oracle Exadata Express Cloud Service



ORACLE®

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

In this lesson, 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.

## Lesson Agenda

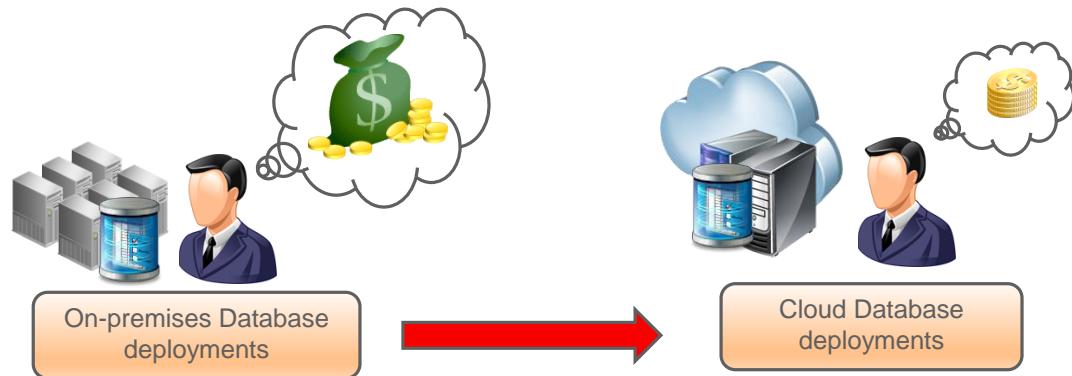
- Overview of Oracle Database Exadata Express Cloud Service
- Understanding the Service Console
- Accessing Cloud Database using SQL Workshop
- Connecting to Exadata Express using Database Clients
  - Connecting Oracle SQL Developer
  - Connecting Oracle SQLcl



ORACLE®

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

## 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.

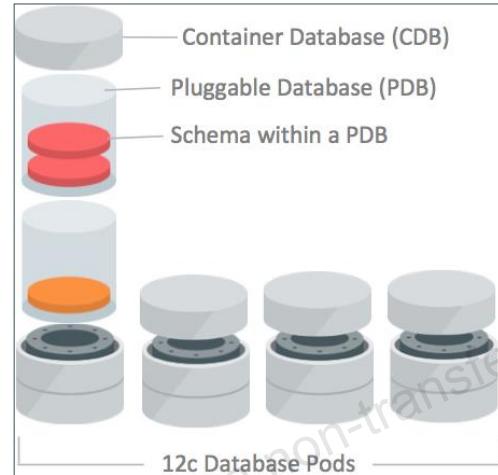
Exadata Express is a powerful database machine, extended as a cloud service. End users can use it for Oracle 12c database deployments. It delivers a complete database experience for developers and enterprises.

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.

## 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.



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).

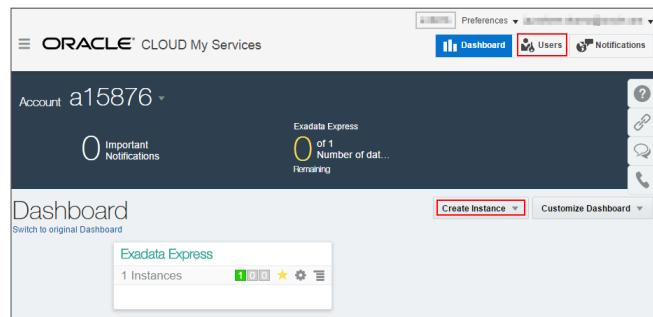
Note: You can refer to [Using Oracle Database Exadata Express Cloud Service](#) for details on the subscription process.



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

## 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 screenshot shows the Oracle Service Console interface. On the left, the 'Service Instances' dashboard lists an instance named 'exa4'. The instance details are as follows:

- Service Type: Exadata Express
- Instance Id: 500033811
- Status: Active
- Size: BASIC
- Service SFTP User Name: u\$148271
- Service SFTP Host & Port: den00vrc.us.oracle.com

An 'Open Service Console' button is highlighted with a red box. An arrow points from this button to the right-hand 'Service Console: exa4' window.

The 'Service Console: exa4' window contains several sections:

- Web Access**: Develop database and web apps using Oracle Application Express (APEX). Includes 'Learn More' and 'Watch Video' links.
- Define REST Data Services**: Create and manage RESTful web service interfaces to your database. Includes 'Learn More' and 'Watch Video' links.
- Client Access**: Download Client Credentials (a zip file containing security credentials and network configuration files). Includes 'Learn More' and 'Watch Video' links.
- Disable Client Access**: Disable SQL\*Net access and invalidate all existing client credential files. Includes 'Learn More' and 'Watch Video' links.
- Administration**: Manage your cloud database. Includes 'Learn More' and 'Watch Video' links.
- Create Database Schema**: Create a new schema for database objects. Includes 'Set Administrator Password' link.
- Download Drivers**: Get database drivers for Java, .NET, Node.js, Python, PHP, Ruby, C, C++, Instant Client and more.
- Download Tools**: Get SQL\*Plus command-line and developer tools including SQL Developer and iDeveloper.
- 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.

At the bottom of the interface, there is an 'ORACLE' logo and a copyright notice: '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.

## Lesson Agenda

- Overview of Oracle Database Exadata Express Cloud Service
- Understanding Service Console
- Accessing Cloud Database using SQL Workshop
- Connecting to Exadata Express using Database Clients
  - Connecting Oracle SQL Developer
  - Connecting Oracle SQLcl

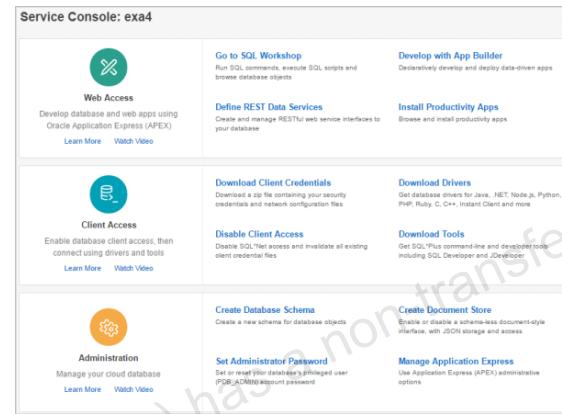


ORACLE®

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

## 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



**ORACLE®**

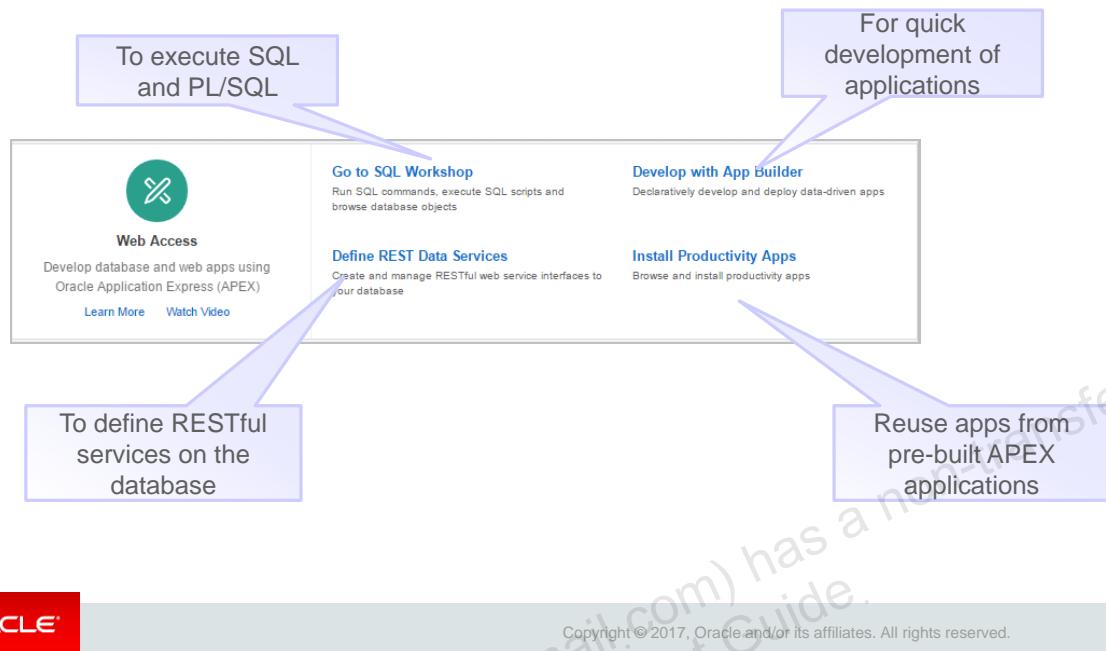
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

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.

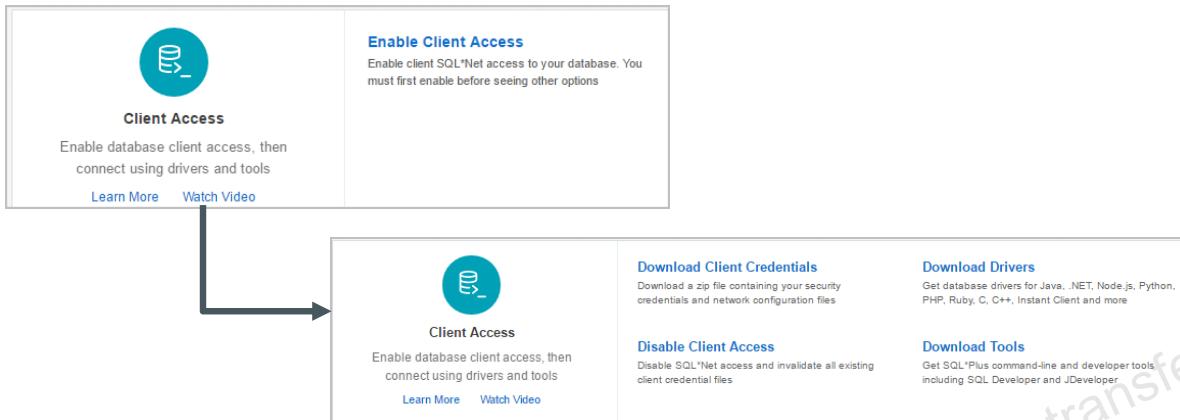
Administration in the Service Console provides for performing administration tasks such as create new database schemas for database objects, set or reset administration password, create a schema-less documents and collections interface, and use administrative options to manage Oracle Application Express.

## Web Access Through Service Console



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



ORACLE®

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 Oracle Service Console interface under the 'Administration' section. It includes the following components:

- 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 each option from the left and bottom right:

- An arrow points from the top-left callout 'Create a new schema for database objects' to the 'Create Database Schema' button.
- An arrow points from the bottom-left callout 'Set or reset password for admin' to the 'Set Administrator Password' button.
- An arrow points from the top-right callout 'To create a document store using a schema' to the 'Create Document Store' button.
- An arrow points from the bottom-right callout 'To manage tasks such as archiving APEX schemas and association among APEX schema' to the 'Manage Application Express' button.

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.

## Lesson Agenda

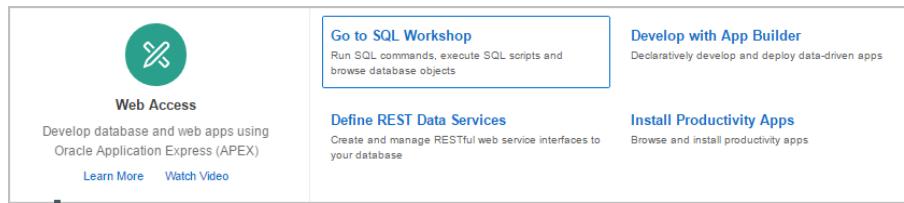
- Overview of Oracle Database Exadata Express Cloud Service
- Understanding the Service Console
- Accessing Cloud Database using SQL Workshop
- Connecting to Exadata Express using Database Clients
  - Connecting Oracle SQL Developer
  - Connecting Oracle SQLcl



ORACLE®

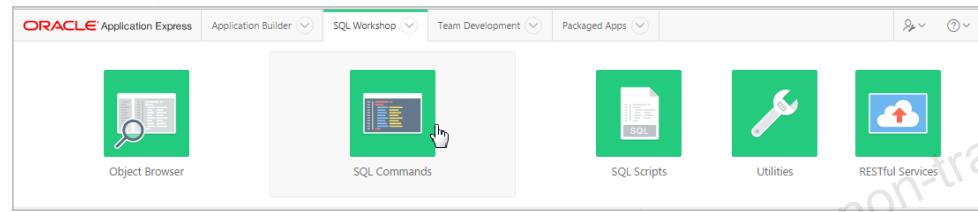
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

## SQL Workshop



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. ( To be used in SQL courses)

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, the schema is set to BZNMKSSA. The SQL Commands editor contains the query `SELECT * FROM emp;`. The Results tab is selected, displaying the following data:

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

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.

## SQL Workshop

You can run PL/SQL statements in the editor. ( To be used in PL/SQL courses)

The screenshot shows the Oracle Application Express SQL Workshop interface. The top navigation bar includes tabs for Application Express, Application Builder, SQL Workshop (which is selected), Team Development, Packaged Apps, and various help and search icons. The schema dropdown is set to BZJNMKSSA. The main workspace contains a SQL command editor with the following code:

```
BEGIN  
DBMS_OUTPUT.PUT_LINE('Hello World');  
END;]
```

Below the editor, there are tabs for Results, Explain, Describe, Saved SQL, and History. The Results tab displays the output of the executed code:

```
Hello World  
Statement processed.  
0.34 seconds
```



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

## Lesson Agenda

- Overview of Oracle Database Exadata Express Cloud Service
- Understanding the Service Console
- Accessing Cloud Database using SQL Workshop
- Connecting to Exadata Express using Database Clients
  - Connecting Oracle SQL Developer
  - Connecting Oracle SQLcl



ORACLE®

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

## 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.

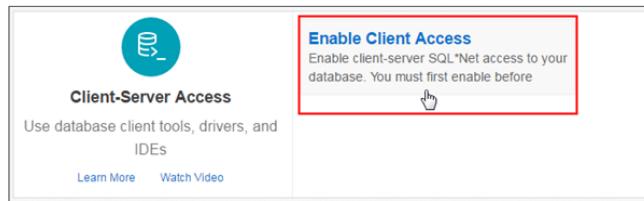
Prerequisite tasks for database client connectivity require you to:

- 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.

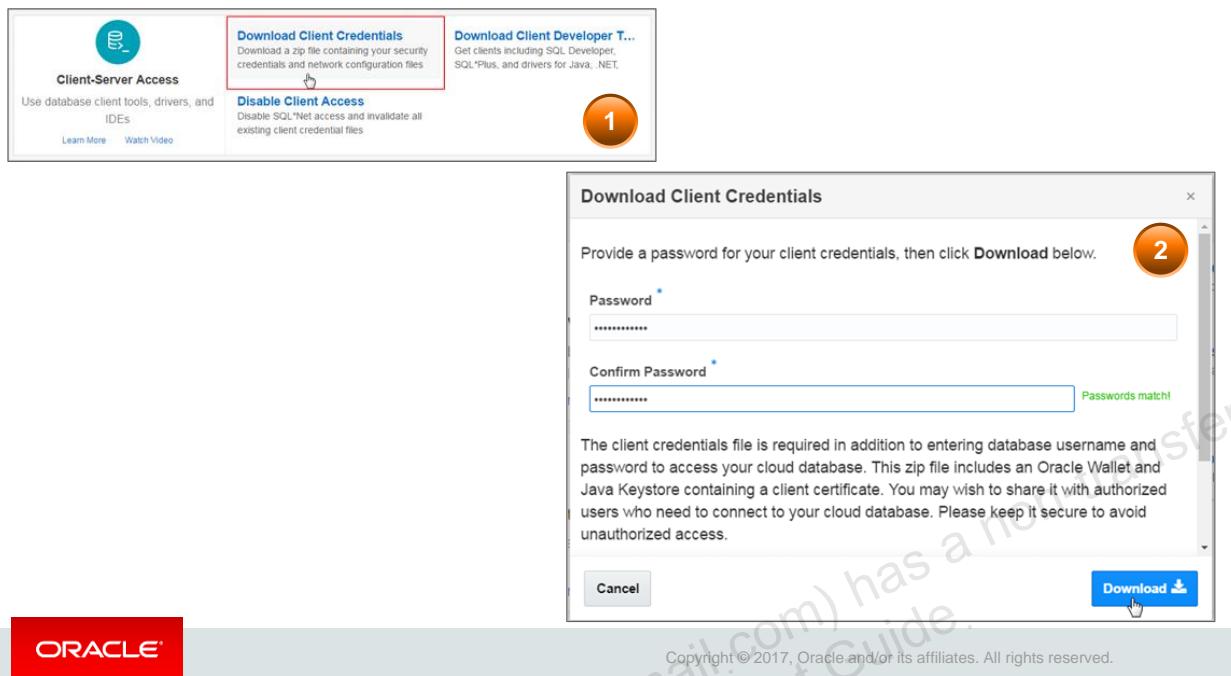
You can connect to your Exadata Express cloud service from diverse database clients over SQL\*Net also called as Oracle Database Net Services. Some examples of supported clients include SQL\*Plus, SQLcl, SQL Developer, JDBC Thin, ODP.NET, OCI, and Instant Client.

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



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.

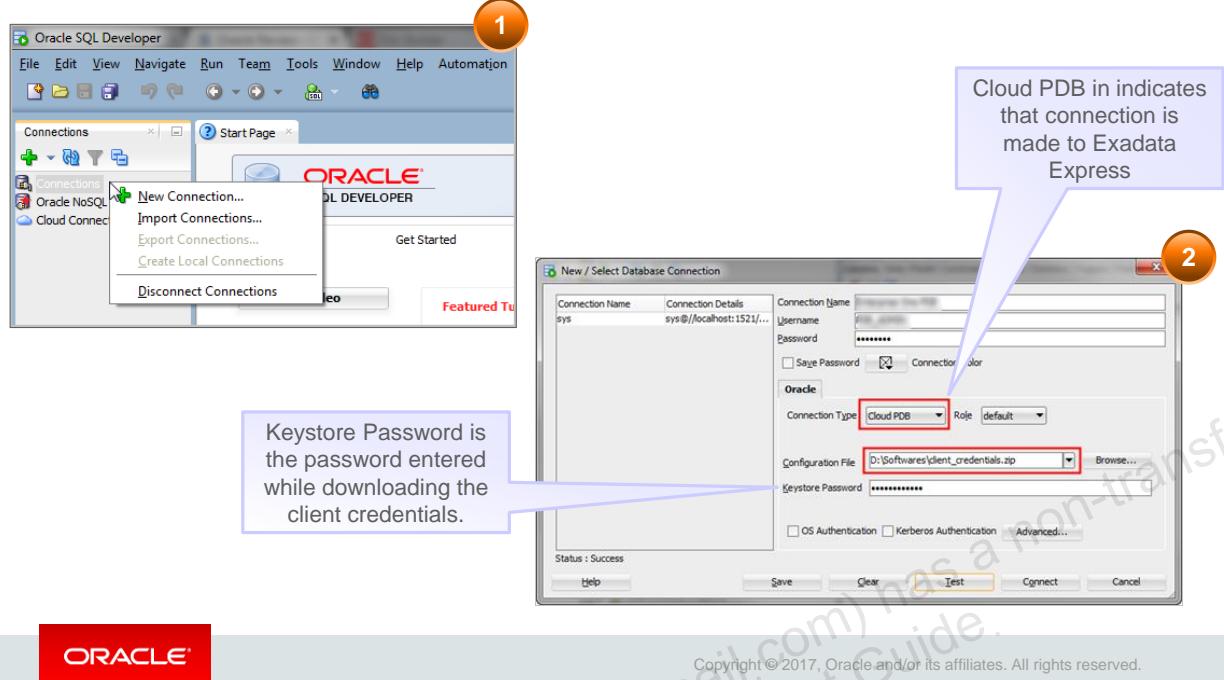
Note that client credential zip files should be carefully managed. Please remember to keep the file secure to avoid unauthorized database access. If you believe the security of this file has been compromised, then immediately disable client access using the cloud service console.

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

ORACLE

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:

- Oracle Database Exadata Express Cloud Service
- Features of Exadata Express Cloud Service
- The process to connect to Database Cloud using SQL Workshop
- The different database clients used to connect to Exadata Express Cloud Service



ORACLE

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

This lesson provided an overview of Oracle Database Exadata Express Cloud Service. If you want to learn more about Oracle Database Exadata Express Cloud Service, you can visit <http://docs.oracle.com/cloud/latest/exadataexpress-cloud/index.html>.

If you want to learn more about Oracle Database Cloud service and get a free trial of Oracle Database Cloud Service, you can visit <https://cloud.oracle.com/database>.

## Practice 2: Overview

This practice covers the following topics:

- Logging in and exploring Oracle Database Exadata Express Cloud Service.
- Running a simple SQL statement from SQL Workshop, Oracle SQL Developer and Oracle SQLcl.



ORACLE®

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

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

PELISSIER LOIC (loic.pelissier@gmail.com) has a non-transferable  
license to use this Student Guide.

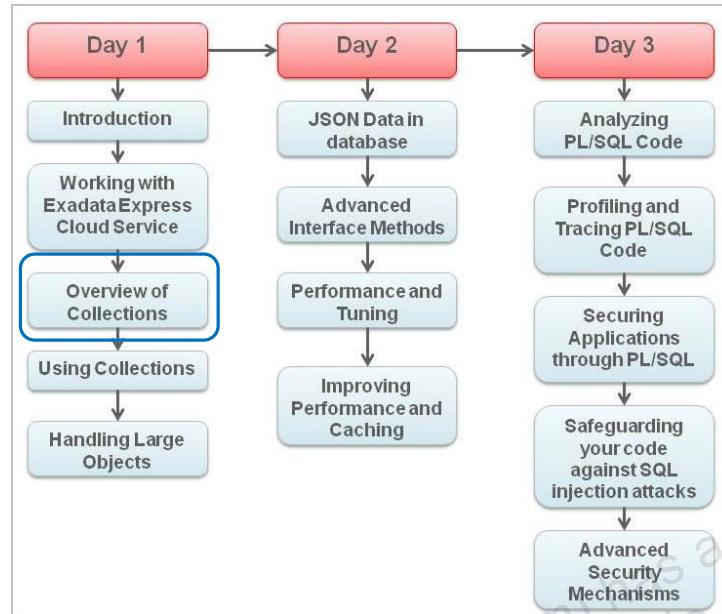
3

# Overview of Collections

ORACLE

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

## Course Agenda



ORACLE

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



In this lesson, you are introduced to Collections. Collections are composite data types, you can store values of data which can in turn have various internal components. In earlier course ( Oracle database 12c: PL/SQL Program Units) you must've learnt about records. Records are also composite data types.

A collection stores a set of values of same data type, whereas a record stores a set of values of different data types. In this lesson we will discuss different variants of collections.

## Objectives

After completing this lesson, you should be able to:

- Create and traverse Associative arrays
- Create and traverse Varrays
- Create and traverse Nested tables



ORACLE®

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

On completing this lesson, you'll understand what are collections and different types of collections. You can create a collection and traverse through the elements of the collection after completing this lesson.

## Lesson Agenda

- Understanding Collections
- Creating Associative Arrays
- Creating Nested Tables
- Creating Varrays



ORACLE®

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

## Collections

- A Collection is a group of data values.
- All the values are of same type
- Each value in the collection is accessed through an index.
- Types of collections in PL/SQL:
  - Associative arrays
    - String-indexed collections
    - INDEX BY PLS\_INTEGER or BINARY\_INTEGER
  - Nested tables
  - Varrays



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

A collection is a group of values, all of the same type. Each value is accessed through an unique subscript that determines its position in the collection. You can group a set of objects( document objects like reports) as a collection. You can store information of a subset of employees(all managers in the organization) in a collection for various purposes in your application. Collections work like arrays found in most third-generation programming languages.

Collections are declared and used within a PL/SQL block. You can retrieve data from database tables into collections and write the data back into tables from PL/SQL blocks

Collections can also be passed as parameters to PL/SQL blocks. You can use them to move columns of data into and out of database tables, or between client-side applications and stored subprograms.

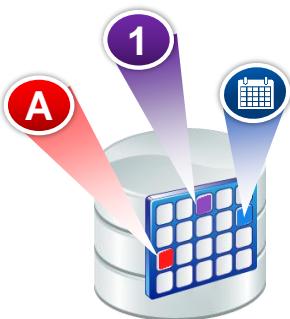
There are three categories of collections:

- Associative arrays (known as “index by tables” in previous Oracle releases) are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be an integer or a string.
- Nested tables can have any number of elements.
- A varray is an ordered collection of elements with an upper limit on the number of elements.

**Note:** Associative arrays indexed by PLS\_INTEGER are covered in the prerequisite courses—*Oracle Database 12c: Program with PL/SQL* and *Oracle Database 12c: Develop PL/SQL Program Units*—and are not emphasized in this course.

## Why Collections?

We have variables.



We have cursors



Then why collections?

ORACLE®

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

You use variables in PL/SQL blocks to store and operate on data. Variables are used to store single values of certain data type. Whenever you want to retrieve a set of values from the database and operate upon it in PL/SQL block, you can use cursors.

A cursor retrieves data from the database based on a SQL query. The data retrieved by the cursor is known as an active set. You can traverse through each row of data retrieved by cursor and perform operations on it. However data retrieved by the cursor is not stored anywhere. Every time you open a cursor the data is fetched into memory from the database and operated upon.

When you want to store and operate on a group of data values in PL/SQL block then using Collections would be the right option. Instead of retrieving the data from the database each time you need it using a cursor, you can have the data in the collection. Using a collection will reduce the I/O and improve the application performance.

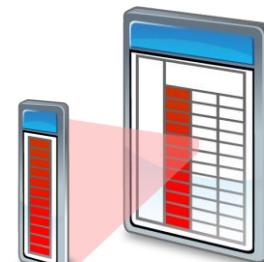
## Collection Types



Associative Arrays



Varrays



Nested Tables



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

### Associative Arrays

Associative arrays are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be either integer (PLS\_INTEGER or BINARY\_INTEGER) or a character string (VARCHAR2). Associative arrays may be sparse.

When you assign a value by using a key for the first time, it adds that key to the associative array. Subsequent assignments using the same key update the same entry. However, it is important to choose a key that is unique. For example, the key values may come from the primary key of a database table, from a numeric hash function, or from concatenating strings to form a unique string value.

Associative arrays are generally used for storing temporary data, therefore you can't use them with SQL statements, such as `INSERT` and `SELECT INTO`. You can make them persistent for the life of a database session by declaring the type in a package and assigning the values in a package body. They are typically populated with a `SELECT BULK COLLECT` statement unless they are `VARCHAR2` indexed. `BULK COLLECT` prevents context switching between the SQL and PL/SQL engines, and is much more efficient on large data sets.

## Nested Tables

A nested table holds a set of values, you can see it as a single column table. It is a column type and can be stored into the database. Each value in such column can hold a set of rows which can be seen as equivalent to a single column table. You can access each row in the nested table through a subscript value starting with 1.

Nested tables are unbounded; that is, the single column table can have any number of rows. In a PL/SQL block, nested tables are like one-dimensional arrays whose size can increase dynamically. In the database, nested tables are column types where each value of this column will in turn hold a single column table. The Oracle database stores the rows of a nested table in no particular order. When you retrieve a nested table from the database into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. This gives you an array-like access to individual rows. Nested tables are initially dense, but they can become sparse through deletions and, therefore, have nonconsecutive subscripts.

## Varrays

A **varray (variable-size array)** is an array whose number of elements can vary from zero (empty) to the declared maximum size. You can access each element of the varray through an index. The lower bound of *index* is 1; the upper bound is the current number of elements. The upper bound changes as you add or delete elements, but it cannot exceed the maximum size. When you store and retrieve a varray from the database, its indexes and element order remain stable.

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.

## Lesson Agenda

- Understanding Collections
- Creating Associative Arrays
- Creating Nested Tables
- Creating Varrays



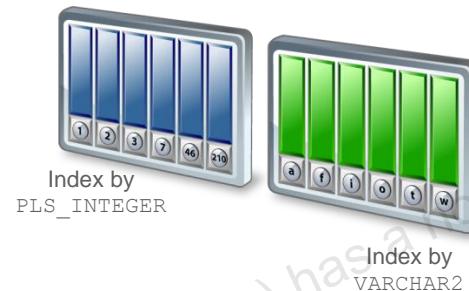
ORACLE®

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

## Using Associative Arrays

Associative arrays:

- That are indexed by strings or integers can improve performance
- Are pure memory structures that are much faster than schema-level tables
- Provide significant additional flexibility



ORACLE

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

An associative array (formerly called PL/SQL table or index-by table) is a set of key-value pairs. Each key is a unique index, you can access the corresponding value through the index.

The data type of index can be either a string type (VARCHAR2, VARCHAR, STRING, or LONG) or PLS\_INTEGER. Indexes are stored in sort order, not creation order.

An associative array can hold unspecified number of elements.

## Creating an Associative Array

**Syntax:**

```
TYPE type_name IS TABLE OF element_type
INDEX BY VARCHAR2(size)
```

**Example:**

```
CREATE OR REPLACE PROCEDURE report_credit
  (p_last_name    customers.cust_last_name%TYPE,
   p_credit_limit customers.credit_limit%TYPE)
IS
  TYPE typ_name IS TABLE OF customers%ROWTYPE
    INDEX BY customers.cust_email%TYPE;
  v_by_cust_email  typ_name;
  i VARCHAR2(50);

  PROCEDURE load_arrays IS
  BEGIN
    FOR rec IN (SELECT * FROM customers WHERE cust_email IS NOT NULL)
    LOOP
      -- Load up the array in single pass to database table.
      v_by_cust_email (rec.cust_email) := rec;
    END LOOP;
  END;
  ...

```

← Create the string-indexed associative array type.

← Create the string-indexed associative array variable.

← Populate the string-indexed associative array variable.



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

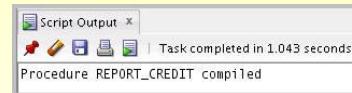
You can see the syntax of creating an associative array or index by tables. You can index the values of array either through an integer (PLS\_INTEGER) or through a character string (VARCHAR2, VARCHAR, STRING, or LONG).

The REPORT\_CREDIT procedure shown in the slide accepts two parameters – the customer last name and a credit limit value. The procedure is expected to display the email ids and credit limit of the customers, whose last name is same as the parameter passed to the procedure and credit limit is greater than the value passed as parameter.

The string-indexed collection is loaded with the customer information in the `load_arrays` procedure. In the main body of the program, the collection is traversed to find the credit information. The associative array is indexed through the email id because it is unique. The last name of the customer can repeat but not the email id.

## Traversing an Associative Array

```
...
BEGIN
    load_arrays;
    i := v_by_cust_email.FIRST;
    dbms_output.put_line ('For credit amount of: ' || p_credit_limit);
    WHILE i IS NOT NULL LOOP
        IF v_by_cust_email(i).cust_last_name = p_last_name
        AND v_by_cust_email(i).credit_limit > p_credit_limit
        THEN dbms_output.put_line ('Customer'|| 
            v_by_cust_email(i).cust_last_name || ':' || 
            v_by_cust_email(i).cust_email || ' has credit limit of: ' || 
            v_by_cust_email(i).credit_limit);
        END IF;
        i := v_by_cust_email.NEXT(i);
    END LOOP;
END report_credit;
/
```



```
EXECUTE report_credit('Walken', 1200)
```

```
PL/SQL procedure successfully completed.
For credit amount of: 1200
Customer Walken: Emmet.Walken@LIMPKIN.EXAMPLE.COM has credit limit of: 3600
Customer Walken: Prem.Walken@BRANT.EXAMPLE.COM has credit limit of: 3700
```

ORACLE

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

You can see that in the given code, you are traversing the collection through a loop. The loop exits when there is a null index while traversing the collection. You can see the usage of `v_by_email.FIRST` and `v_by_cust_email.NEXT(i)`. These are predefined Collection methods.

Here is the modified code:

```

CREATE OR REPLACE PROCEDURE report_credit
    (p_email      customers.cust_email%TYPE,
     p_credit_limit customers.credit_limit%TYPE)
IS
    TYPE typ_name IS TABLE OF customers%ROWTYPE
        INDEX BY customers.cust_email%TYPE;
    v_by_cust_email typ_name;
    i VARCHAR2(50);

    PROCEDURE load_arrays IS
    BEGIN
        FOR rec IN (SELECT * FROM customers
                    WHERE cust_email IS NOT NULL) LOOP
            v_by_cust_email (rec.cust_email) := rec;
        END LOOP;
    END;

    BEGIN
        load_arrays;
        dbms_output.put_line
            ('For credit amount of: ' || p_credit_limit);
        IF v_by_cust_email(p_email).credit_limit > p_credit_limit
            THEN dbms_output.put_line ( 'Customer ' ||
                v_by_cust_email(p_email).cust_last_name ||
                ': ' || v_by_cust_email(p_email).cust_email ||
                ' has credit limit of: ' || v_by_cust_email(p_email).credit_limit);
        END IF;
    END report_credit;
/
EXECUTE report_credit('Prem.Walken@BRANT.EXAMPLE.COM', 100)

```

PL/SQL procedure successfully completed.

For credit amount of: 100  
 Customer Walken: Prem.Walken@BRANT.EXAMPLE.COM has credit limit of: 3700

## Collection Methods

- A collection method is a PL/SQL sub program.
- It can be a function or a procedure.
- Collection methods simplify the usage of Collections in PL/SQL block.
- You can use the collection methods along with the Collection variable.

Usage:

```
collection_variable.method
```



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

A collection method is a PL/SQL subprogram – procedure or function. You can use the collection method in the PL/SQL block just as you use any other procedure or function.

You invoke the collection methods along with the collection variables. Following is a list of commonly used collection methods:

Method	Type	Description
DELETE	Procedure	Deletes elements from collection.
TRIM	Procedure	Deletes elements from end of varray or nested table.
EXTEND	Procedure	Adds elements to end of varray or nested table.
EXISTS	Function	Returns TRUE if and only if specified element of varray or nested table exists.
FIRST	Function	Returns first index in collection.
LAST	Function	Returns last index in collection.
COUNT	Function	Returns number of elements in collection.
LIMIT	Function	Returns maximum number of elements that collection can have.
PRIOR	Function	Returns index that precedes specified index.
NEXT	Function	Returns index that succeeds specified index.

## Lesson Agenda

- Understanding Collections
- Creating Associative Arrays
- Creating Nested Tables
- Creating Varrays



ORACLE®

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

## Nested Tables

- A collection which can hold a set of values or rows.
- All the values are of the same type.
- There is no upper limit on the number of values in the collection.
- Can be stored as a column type in a table.
- Each value in a nested table is accessed through an index starting from 1.



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

A nested table holds a set of values. There is no upper limit on the number of values that can be stored in a nested table. The size of the nested table can vary dynamically.

When used in a PL/SQL block each value in the collection is referenced through a numeric index starting from 1.

Within the database, nested tables are column types that hold set of values. The Oracle database stores the rows of a nested table in no particular order. Each row in the nested table can be a set of scalar values or a set of records.

Nested tables are initially dense, but they can become sparse through deletions and, therefore, have nonconsecutive subscripts.

## Creating Nested Table Types

- To create a nested table type in the database:

```
CREATE [OR REPLACE] TYPE type_name AS TABLE OF  
Element_datatype [NOT NULL];
```

- To create a nested table type in PL/SQL:

```
TYPE type_name IS TABLE OF element_datatype  
[NOT NULL];
```



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

To create a collection, you first define a collection type, and then declare collections of that type. The slide shows the syntax for defining the nested table collection type in both the database (persistent) and in PL/SQL (transient).

### Creating Collections in the Database

You can create a nested table data type in the database, which makes the data type available to use in places such as columns in database tables, variables in PL/SQL programs, and attributes of object types.

Before you can define a database table containing a nested table, you must first create the data type for the collection in the database.

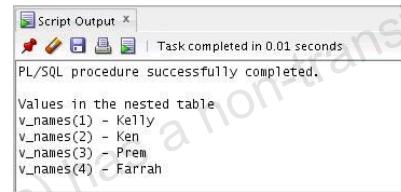
Use the syntax shown in the slide to create collection types in the database.

### Creating Collections in PL/SQL

You can also create a nested table in PL/SQL. Use the syntax shown in the slide to create collection types in PL/SQL.

## Nested Tables - Example

```
DECLARE
    TYPE names IS TABLE OF VARCHAR2(15);
    v_names names := names('Kelly','Ken', 'Prem', 'Farrah');
BEGIN
    DBMS_OUTPUT.PUT_LINE('Values in the nested table');
    FOR i IN v_names.FIRST .. v_names.LAST LOOP
        DBMS_OUTPUT.PUT_LINE('v_names'||i||') -'||v_names(i));
    END LOOP;
END;
```



ORACLE

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

The slide shows a simple nested table created in a PL/SQL block.

You create a data type `names` which is the nested table. You can store nested tables as column types in the database. Therefore before attempting to store the nested table in the database, you have to create a type of the nested table. In the slide we create a data type `names` and create a variable `v_names` of type `names`.

An uninitialized nested table is a null collection. You are initializing the nested table in the code through a collection constructor. You will learn about collection constructors in the next slide.

You traverse the nested table using a for loop, the first value of the collection `v_names` is referred through `v_names.first` and the last value is referred through `v_names.last`.

## Collection Constructors

- A **collection constructor (constructor)** is a system-defined function with the same name as a collection type, which returns a collection of that type.

Example:

```
v_names names := names('Kelly', 'Ken', 'Prem', 'Farrah');
```



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

A collection constructor is used to initialize the collection. In the example the collection `v_names` of type `names` is initialized with a list of four values.

You can also create null collections by having an empty initialization list.

## Declaring Collections: Nested Table

- First, define an object type:

```

CREATE TYPE typ_item AS OBJECT --create object
  (prodid NUMBER(5),
   price   NUMBER(7,2) )
/
CREATE TYPE typ_item_nst -- define nested table type
  AS TABLE OF typ_item
/

```

1

2

- Then, declare a column of that collection type:

```

CREATE TABLE pOrder ( -- create database table
  ordid   NUMBER(5),
  supplier NUMBER(5),
  requester  NUMBER(4),
  ordered   DATE,
  items    typ_item_nst)
/
  NESTED TABLE items STORE AS item_stor_tab
/

```

3



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

You have seen creating a collection of scalar data values in the earlier example. You can also create a collection of objects. The slide shows creation of the object type and then creating a collection of the objects defined.

To create a table based on a nested table, perform the following steps:

- Create the `typ_item` type, which holds the information for a single line item.
- Create the `typ_item_nst` type, which is created as a table of the `typ_item` type.  
**Note:** You must create the `typ_item_nst` nested table type based on the previously declared type.
- Create the `pOrder` table and use the nested table type in a column declaration, which includes an arbitrary number of items based on the `typ_item_nst` type. Thus, each row of `pOrder` may contain a table of items.

The `NESTED TABLE STORE AS` clause is required to indicate the name of the storage table in which the rows of all values of the nested table reside. The storage table is created in the same schema and the same tablespace as the parent table.

**Note:** The `USER_COLL_TYPES` dictionary view holds information about collections.

## Using Nested Tables

- Add data to the nested table:

```
INSERT INTO pOrder
VALUES (500, 50, 5000, sysdate, typ_item_nst(
    typ_item(55, 555),
    typ_item(56, 566),
    typ_item(57, 577)));

```

```
INSERT INTO pOrder
VALUES (800, 80, 8000, sysdate,
typ_item_nst (typ_item (88, 888)));
```

pOrder nested table

ORDID	SUPPLIER	REQUESTER	ORDERED	ITEMS
500	50	5000	30-OCT-07	
800	80	8000	31-OCT-07	

PRODID	PRICE
55	555
56	566
57	577

PRODID	PRICE
88	888



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

To insert data into the nested table, you use the `INSERT` statement. You are using a collection constructor to add values to the nested table.

In the example in the slide, the constructors are `TYP_ITEM_NST()` and `TYP_ITEM()`. You pass two elements to the `TYP_ITEM()` constructor, and then pass the results to the `TYP_ITEM_NST()` constructor to build the nested table structure.

The first `INSERT` statement builds the nested table with three sub element rows.

The second `INSERT` statement builds the nested table with one sub element row.

## Using Nested Tables

- Querying the results:

```
select * from porder;
```

	ORDID	SUPPLIER	REQUESTER	ORDERED	ITEMS
1	500	50	5000	09-MAR-17	OE.TYP_ITEM_NST([OE.TYP_ITEM],[OE.TYP_ITEM],[OE.TYP_ITEM])
2	800	80	8000	09-MAR-17	OE.TYP_ITEM_NST([OE.TYP_ITEM])

- Querying the results with the TABLE function:

```
select * from p2.ordid,p1.*  
from porder p2, TABLE(p2.items) p1;
```

	ORDID	PRODID	PRICE
1	500	55	555
2	500	56	566
3	500	57	577
4	800	88	888

ORACLE

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

In order to see the data in the nested table you should use the TABLE function. When you execute a simple select statement to see the data in the table then the data in the nested table column would not appear.

To display the output you should unnest the collection such that each collection element appears on a row by itself. You can use the TABLE expression in the FROM clause to unnest a collection.

A TABLE expression enables you to query a collection in the FROM clause like a table. In effect, you join the nested table with the row that contains the nested table without writing a JOIN statement.

The collection column in the TABLE expression uses a table alias to identify the containing table.

## Referencing Collection Elements

Use the collection name and a subscript to reference a collection element:

- Syntax:

```
collection_name(subscript)
```

- Example:

```
v_with_discount(i)
```

- To reference a field in a collection:

```
p_new_items(i).prodid
```



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

Every element reference includes a collection name and a subscript enclosed in parentheses. The subscript determines which element is processed. To reference an element, you can specify its subscript by using the following syntax:

```
collection_name(subscript)
```

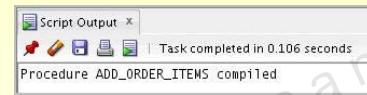
In the preceding syntax, `subscript` is an expression that yields a positive integer. For nested tables, the integer must lie in the range 1 through 2147483647. For varrays, the integer must lie in the range 1 through `maximum_size`.

## Using Nested Tables in PL/SQL

```

CREATE OR REPLACE PROCEDURE add_order_items
(p_ordid NUMBER, p_new_items typ_item_nst)
IS
    v_num_items      NUMBER;
    v_with_discount typ_item_nst;
BEGIN
    v_num_items := p_new_items.COUNT;
    v_with_discount := p_new_items;
    IF v_num_items > 2 THEN
        --ordering more than 2 items gives a 5% discount
        FOR i IN 1..v_num_items LOOP
            v_with_discount(i) :=
                typ_item(p_new_items(i).prodid,
                          p_new_items(i).price*.95);
        END LOOP;
    END IF;
    UPDATE pOrder
        SET items = v_with_discount
        WHERE ordid = p_ordid;
END;

```



ORACLE

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

When you define a variable of a collection type in a PL/SQL block, it is transient and available only for the scope of the PL/SQL block.

In the example shown in the slide:

- The nested table `P_NEW_ITEMS` parameter is passed into the block.
- A local variable `V_WITH_DISCOUNT` is defined with the nested table data type `TYP_ITEM_NST`.
- A collection method, called `COUNT`, is used to determine the number of items in the nested table.
- If more than two items are counted in the collection, the local nested table variable `V_WITH_DISCOUNT` is updated with the product ID and a 5% discount on the price.
- To reference an element in the collection, the subscript `i`, representing an integer from the current loop iteration, is used with the constructor method to identify the row of the nested table.
- Update the `pOrder` table with the discount values.

## Using Nested Tables in PL/SQL

```
-- caller pgm:
DECLARE
    v_form_items  typ_item_nst:= typ_item_nst();
BEGIN
    -- let's say the form holds 4 items
    v_form_items.EXTEND(4);
    v_form_items(1) := typ_item(1804, 65);
    v_form_items(2) := typ_item(3172, 42);
    v_form_items(3) := typ_item(3337, 800);
    v_form_items(4) := typ_item(2144, 14);
    add_order_items(800, v_form_items);
END;
```

```
SELECT p2.ordid, p1.price
FROM porder p2, TABLE(p2.items) p1;
```

v\_form\_items variable

PROID	PRICE
1804	65
3172	42
3337	800
2144	14

Query Result X

SQL | All Rows Fetched: 7 in 0.007 seconds

ORDID	PRICE
1	500
2	500
3	500
4	800
5	800
6	800
7	800



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

In the example code shown in the slide:

- A local PL/SQL variable of nested table type is declared and instantiated with the `TYP_ITEM_NST()` collection method
- The nested table variable is extended to hold four rows of elements with the `EXTEND (4)` method.
- The nested table variable is populated with four rows of elements by constructing a row of the nested table with the `TYP_ITEM` constructor.
- The nested table variable is passed as a parameter to the `ADD_ORDER_ITEMS` procedure shown on the previous page.
- The `ADD_ORDER_ITEMS` procedure updates the `ITEMS` nested table column in the `pOrder` table with the contents of the nested table parameter passed into the routine.

## Lesson Agenda

- Understanding Collections
- Creating Associative Arrays
- Creating Nested Tables
- Creating Varrays



ORACLE®

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

## Varrays

- Varrays are variable sized arrays.
- Varray can hold data values from zero to a declared maximum size.
- Can be stored in the database as a table column.



ORACLE®

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

Varrays are also collections of homogeneous elements that hold a fixed number of elements (although you can change the number of elements using `EXTEND` method). They use sequential numbers as subscripts.

You can define varrays as a SQL type, thereby allowing varrays to be stored in database tables. They can be stored and retrieved through SQL, but with less flexibility than nested tables. You can reference individual elements for array operations, or manipulate the collection as a whole.

You can define varrays in PL/SQL to be used during PL/SQL program execution.

Varrays are always bounded and never sparse. You can specify the maximum size of the varray in its type definition. Its index has a fixed lower bound of 1 and an extensible upper bound. A varray can contain a varying number of elements, from zero (when empty) to the maximum specified in its type definition.

To reference an element, you can use the standard subscripting syntax.

## Varrays

- To create a varray in the database:

```
CREATE [OR REPLACE] TYPE type_name AS VARRAY  
  (max_elements) OF element_datatype [NOT NULL];
```

- To create a varray in PL/SQL:

```
TYPE type_name IS VARRAY (max_elements) OF  
  element_datatype [NOT NULL];
```



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

You can see the syntax of creating a varray in the slide. Varrays like nested tables can be stored as database columns. You can choose to use a varray when you know the maximum number of values that have to be accommodated in the collection.

You can increase the capacity of nested tables or varrays using EXTEND method.

## Declaring Collections: Varray

- First, define a collection type:

```

CREATE TYPE typ_Project AS OBJECT( --create object
    project_no NUMBER(4),
    title      VARCHAR2(35),
    cost       NUMBER(12,2))
/
CREATE TYPE typ_ProjectList AS VARRAY(50) OF typ_Project
    -- define VARRAY type
/

```

1

2

- Then, declare a collection of that type:

```

CREATE TABLE department ( -- create database table
    dept_id  NUMBER(2),
    name     VARCHAR2(25),
    budget   NUMBER(12,2),
    projects typ_ProjectList) -- declare varray as column
/

```

3



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

The example in the slide shows how to create a table based on a varray.

- Create the TYP\_PROJECT type, which holds the information for a project.
- Create the TYP\_PROJECTLIST type, which is created as a varray of the project type. The varray contains a maximum of 50 elements.
- Create the DEPARTMENT table and use the varray type in a column declaration. Each element of the varray will store a project object.

This example demonstrates how to create a varray of phone numbers, and then use it in a CUSTOMERS table: (The OE sample schema uses this definition.)

```

CREATE TYPE phone_list_typ
AS VARRAY(5) OF VARCHAR2(25);
/
CREATE TABLE customers
(customer_id NUMBER(6)
,cust_first_name VARCHAR2(50)
,cust_last_name VARCHAR2(50)
,cust_address cust_address_typ(100)
,phone_numbers phone_list_typ
...
);

```

## Using Varrays

Add data to the table containing a varray column:

```
INSERT INTO department
VALUES (10, 'Executive Administration', 30000000,
       typ_ProjectList(
         typ_Project(1001, 'Travel Monitor', 400000),
         typ_Project(1002, 'Open World', 10000000)));
1

INSERT INTO department
VALUES (20, 'Information Technology', 5000000,
       typ_ProjectList(
         typ_Project(2001, 'DB11gR2', 900000)));
2
```

**DEPARTMENT** table

DEPT_ID	NAME	BUDGET	PROJECTS		
			PROJECT_NO	TITLE	COSTS
10	Executive Administration	30000000	1001	Travel Monitor	400000
			1002	Open World	10000000
20	Information Technology	5000000	2001	DB11gR2	900000



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

To add rows to the DEPARTMENT table that contains the PROJECTS varray column, you use the INSERT statement. The structure of the varray column is identified with the constructor methods.

- `TYP_PROJECTLIST()` constructor constructs the varray data type.
- `TYP_PROJECT()` constructs the elements for the rows of the varray data type.

The first INSERT statement adds three rows to the PROJECTS varray for department 10.

The second INSERT statement adds one row to the PROJECTS varray for department 20.

## Using Varrays

- Querying the results:

```
SELECT * FROM department;
```

Query Result			
SQL   All Rows Fetched: 2 in 0.478 seconds			
DEPT_ID	NAME	BUDGET	PROJECTS
1	10 Executive Administration	3000000	0E.TYP_PROJECTLIST([0E.TYP_PROJECT],[0E.TYP_PROJECT])
2	20 Information Technology	5000000	0E.TYP_PROJECTLIST([0E.TYP_PROJECT])

- Querying the results with the TABLE function:

```
SELECT d2.dept_id, d2.name, d1.*  
FROM department d2, TABLE(d2.projects) d1;
```

Query Result				
SQL   All Rows Fetched: 3 in 0.005 seconds				
DEPT_ID	NAME	PROJECT_NO	TITLE	COST
1	10 Executive Administration	1001	Travel Monitor	400000
2	10 Executive Administration	1002	Open World	1000000
3	20 Information Technology	2001	DB11gR2	900000



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

You query a varray column in the same way that you query a nested table column.

In the first example in the slide, the collections are nested in the result rows that contain them. By including the collection column in the SELECT list, the output shows as a row associated with the other row output in the SELECT list.

In the second example, the output is unnested such that each collection element appears on a row by itself. You can use the TABLE expression in the FROM clause to unnest a collection.

## Quiz



Which of the following collections is a set of key-value pairs, where each key is unique and is used to locate a corresponding value in the collection?

- a. Associative arrays
- b. Nested Table
- c. Varray
- d. Semsegs

ORACLE

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

**Answer: a**

## Quiz



Which of the following collections can be stored in the database?

- a. Associative arrays
- b. Nested table
- c. Varray

ORACLE

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

**Answer: b, c**

## Summary

In this lesson, you should have learned how to:

- Identify types of collections
  - Nested tables
  - Varrays
  - Associative arrays
- Define nested tables and varrays in the database



ORACLE®

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

Collections are a grouping of elements, all of the same type. The types of collections are nested tables, varrays, and associative arrays. You can define nested tables and varrays in the database. Nested tables, varrays, and associative arrays can be used in a PL/SQL program.

There are guidelines for using collections effectively and for determining which collection type is appropriate under specific circumstances.

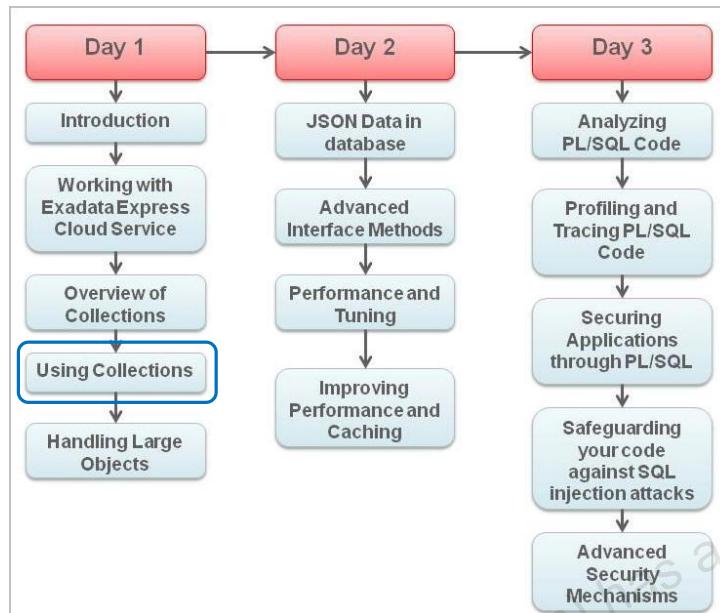
# 4

## Using Collections

ORACLE

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

## Course Agenda



ORACLE

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



## Objectives

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

- Use collection methods
- Manipulate collections
- Distinguish between the different types of collections and when to use them
- Use PL/SQL bind types



ORACLE

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

In this lesson, you learn about using collections.

Collection is a structure used to store a group of values of the same type. You can access each element in the collection through an index. You can use collections to work with a group of data in your PL/SQL block that may or may not stored in the database tables.

There are three types of collections – associative arrays, nested tables and varrays. Associative arrays are used only in the PL/SQL blocks, you cannot run SQL statements on them. You can have Nested tables and Varrays as column types in database tables and therefore run SQL queries on these collections.

## Lesson Agenda

- Working with Collections
- Collection exceptions
- Summarizing Collections
- PL/SQL Bind types



ORACLE®

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

## Usage of Collections in Applications

Consider a scenario where you have to keep track of various projects run by a company

- The company has multiple departments
- Each department runs multiple projects
- There is a budget allocated to each project

The developer has to create a PL/SQL package to manage various projects.



ORACLE

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

Consider a scenario where the developer has to create mechanisms to manage various projects of a department.

You will create a package `manage_dept_proj` to maintain the data of various projects handled by different departments. The package will implement the following functionality.

1. Assign new projects to a department( Procedure `allocate_new_proj_list`).
2. Retrieve data of a project from the database( Function `get_dept_project` ).
3. Update project data in the database (Procedure `update_a_project`).
4. Perform other operations on the project data (`manipulate_project`).
5. Check the costs of the projects incurred by the department( Function `check_costs`).

## Working with Collections in PL/SQL

- You can declare collections as the formal parameters of procedures and functions.
- You can specify a collection type in the RETURN clause of a function specification.
- The code of `manage_dept_proj` uses collection both as parameters and return values.

```
CREATE OR REPLACE PACKAGE manage_dept_proj
AS
    PROCEDURE allocate_new_proj_list
        (p_dept_id NUMBER, p_name VARCHAR2, p_budget NUMBER);
    FUNCTION get_dept_project (p_dept_id NUMBER)
        RETURN typ_projectlist;
    PROCEDURE update_a_project
        (p_deptno NUMBER, p_new_project typ_Project,
         p_position NUMBER);
    FUNCTION manipulate_project (p_dept_id NUMBER)
        RETURN typ_projectlist;
    FUNCTION check_costs (p_project_list typ_projectlist)
        RETURN boolean;
END manage_dept_proj;
```



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

There are several points about collections that you must know when working with them:

- You can declare collections as the formal parameters of functions and procedures.
- A function's RETURN clause can be a collection type.
- Collections follow the usual scoping and instantiation rules. In a block or subprogram, collections are instantiated when you enter the block or subprogram and cease to exist when you exit. In a package, collections are instantiated when you first reference the package and cease to exist when you end the database session.

**Note:** The code in the following page has references `TYP_PROJECT` and `TYP_PROJECTLIST` types and a `department` table. The SQL for the type definition is as follows:

```
CREATE OR REPLACE TYPE typ_project AS OBJECT (
    project_no NUMBER(4),
    title VARCHAR2(35),
    cost NUMBER(12,2));
CREATE OR REPLACE typ_projectlist AS VARRAY (50) OF typ_project;
CREATE TABLE department ( -- create database table
    dept_id NUMBER(2),
    name      VARCHAR2(25),
    budget    NUMBER(12,2),
    projects typ_ProjectList) -- declare varray as column
```

This is the package body for the varray examples shown on the following pages.

```

CREATE OR REPLACE PACKAGE BODY manage_dept_proj
AS
    PROCEDURE allocate_new_proj_list
        (p_dept_id NUMBER, p_name VARCHAR2, p_budget NUMBER)
    IS
        v_accounting_project typ_projectlist;
    BEGIN -- this example uses a constructor
        v_accounting_project :=
            typ_ProjectList
                (typ_Project (1, 'Dsgn New Expense Rpt', 3250),
                 typ_Project (2, 'Outsource Payroll', 12350),
                 typ_Project (3, 'Audit Accounts Payable', 1425));
        INSERT INTO department VALUES
            (p_dept_id, p_name, p_budget, v_accounting_project);
    END allocate_new_proj_list;

    FUNCTION get_dept_project (p_dept_id NUMBER)
        RETURN typ_projectlist
    IS
        v_accounting_project typ_projectlist;
    BEGIN
        -- this example uses a fetch from the database
        SELECT projects
            INTO v_accounting_project
            FROM department
            WHERE dept_id = p_dept_id;
        RETURN v_accounting_project;
    END get_dept_project;

    PROCEDURE update_a_project
        (p_deptno NUMBER, p_new_project typ_Project,
         p_position NUMBER)
    IS
        v_my_projects typ_ProjectList;
    BEGIN
        v_my_projects := get_dept_project (p_deptno);
        v_my_projects.EXTEND;      --make room for new project
        /* Move varray elements forward */
        FOR i IN REVERSE p_position..v_my_projects.LAST - 1 LOOP
            v_my_projects(i + 1) := v_my_projects(i);
        END LOOP;
        v_my_projects(p_position) := p_new_project; -- add new
                                                    -- project
        UPDATE department SET projects = v_my_projects
            WHERE dept_id = p_deptno;
    END update_a_project;

```

```

FUNCTION manipulate_project (p_dept_id NUMBER)
    RETURN typ_projectlist
IS
    v_accounting_project typ_projectlist;
    v_changed_list typ_projectlist;
BEGIN
    SELECT projects
        INTO v_accounting_project
        FROM department
        WHERE dept_id = p_dept_id;
-- this example assigns one collection to another
    v_changed_list := v_accounting_project;
    RETURN v_changed_list;
END manipulate_project;

FUNCTION check_costs (p_project_list typ_projectlist)
    RETURN boolean
IS
    c_max_allowed      NUMBER := 10000000;
    i                  INTEGER;
    v_flag             BOOLEAN := FALSE;
BEGIN
    i := p_project_list.FIRST ;
    WHILE i IS NOT NULL LOOP
        IF p_project_list(i).cost > c_max_allowed then
            v_flag := TRUE;
            dbms_output.put_line (p_project_list(i).title ||
                ' exceeded allowable budget.');
        END IF;
        i := p_project_list.NEXT(i);
    END LOOP;
    RETURN null;
END check_costs;

END manage_dept_proj;

```

## Assigning Values to Collection Variables

You can initialize a constructor using one of the following mechanisms

- Invoke a constructor
- Use an assignment statement
- Pass it to a subprogram as a parameter and assign value in the sub program
- Fetch from the database into the collection



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

Until you initialize it, a collection is null (that is, the collection itself is null, not its elements). To initialize a collection, you can use one of the following methods:

- Use a constructor, which is a system-defined function with the same name as the collection type. A constructor allows the creation of an object from an object type. Invoking a constructor is a way to instantiate (create) an object. This function “constructs” collections from the elements passed to it.
- Assign another collection variable directly. You can copy the entire contents of one collection to another as long as both are built from the same data type.
- Pass collection as a parameter to a sub program and then assign values within the subprogram
- Fetch data from the database through a `SELECT...INTO` statement.

## Assigning Values to Collection Variables

```
CREATE OR REPLACE PROCEDURE allocate_new_proj_list
  (p_dept_id NUMBER, p_name VARCHAR2, p_budget NUMBER)
IS
  v_accounting_project typ_projectlist;
BEGIN
  -- this example uses a constructor
  v_accounting_project :=
    typ_ProjectList
      (typ_Project (1, 'Dsgn New Expense Rpt', 3250),
       typ_Project (2, 'Outsource Payroll', 12350),
       typ_Project (3, 'Audit Accounts Payable',1425));
  INSERT INTO department
    VALUES(p_dept_id, p_name, p_budget, v_accounting_project);
END allocate_new_proj_list;
```



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

In the procedure, the collection is assigned values through a constructor.

## Assigning Values to Collection Variables

```
FUNCTION get_dept_project (p_dept_id NUMBER)
  RETURN typ_projectlist
IS
  v_accounting_project typ_projectlist;
BEGIN -- this example uses a fetch from the database
  SELECT projects INTO v_accounting_project
    FROM department WHERE dept_id = p_dept_id;
  RETURN v_accounting_project;
END get_dept_project;
```

1

```
FUNCTION manipulate_project (p_dept_id NUMBER)
  RETURN typ_projectlist
IS
  v_accounting_project typ_projectlist;
  v_changed_list typ_projectlist;
BEGIN
  SELECT projects INTO v_accounting_project
    FROM department WHERE dept_id = p_dept_id;
-- this example assigns one collection to another
  v_changed_list := v_accounting_project;
  RETURN v_changed_list;
END manipulate_project;
```

2



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

In the first example in the slide, the data from the database is fetched into the local PL/SQL collection variable.

In the second example, one collection variable is assigned to another collection variable.

## Accessing Values in the Collection

```
-- sample caller program to the manipulate_project function
DECLARE
    v_result_list typ_projectlist;
BEGIN
    v_result_list := manage_dept_proj.manipulate_project(10);
    FOR i IN 1..v_result_list.COUNT LOOP
        dbms_output.put_line('Project #: '
            ||v_result_list(i).project_no);
        dbms_output.put_line('Title: '||v_result_list(i).title);
        dbms_output.put_line('Cost: ' ||v_result_list(i).cost);
    END LOOP;
END;
```

The screenshot shows the Oracle SQL Developer interface with a yellow callout box highlighting the 'Script Output' window. The window title is 'Script Output x'. It displays the message 'Task completed in 0.002 seconds.' followed by the text 'PL/SQL procedure successfully completed.' Below this, two sets of project details are listed:

Project #	Title	Cost
1001	Travel Monitor	400000
1002	Open World	10000000



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

The PL/SQL code in the example calls the `manipulate_project` function in the `MANAGE_DEPT_PROJ` package. Department 10 is passed in as the parameter. The output shows the varray element values for the `PROJECTS` column in the `DEPARTMENT` table for department 10.

Although the value of 10 is hard-coded, you can have a form interface to query the user for a department value that can then be passed into the routine.

## Working with Collection Methods

- Collection methods are predefined PL/SQL programs
- Collection methods help user to operate on collections.

`collection_name.method_name [(parameters)]`



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

Function or Procedure	Description
EXISTS	Returns TRUE if and only if specified element of varray or nested table exists
COUNT	Returns the number of elements that a collection contains
LIMIT	Returns the maximum number of elements that a collection can have
FIRST and LAST	Returns the first and last (smallest and largest) indexes in a collection
PRIOR and NEXT	PRIOR (n) returns the index that precedes index n in a collection; NEXT (n) returns the index that follows index n.
EXTEND	Appends one null element. EXTEND (n) appends n elements; EXTEND (n, i) appends n copies of the i th element.
TRIM	Removes one element from the end; TRIM (n) removes n elements from the end of a collection
DELETE	Removes all elements from a nested or associative array table. DELETE (n) removes the nth element; DELETE (m, n) removes a range. Note: This does not work on varrays.

## Using Collection Methods

Traverse collections with the following methods:

```
FUNCTION check_costs (p_project_list typ_projectlist)
    RETURN boolean
IS
    c_max_allowed      NUMBER := 10000000;
    i                  INTEGER;
    v_flag             BOOLEAN := FALSE;
BEGIN
    i := p_project_list.FIRST ;
    WHILE i IS NOT NULL LOOP
        IF p_project_list(i).cost > c_max_allowed then
            v_flag := TRUE;
            dbms_output.put_line (p_project_list(i).title || '
                                exceeded allowable budget.');
        END IF;
        i := p_project_list.NEXT(i);
    END LOOP;
    RETURN null;
END check_costs;
```



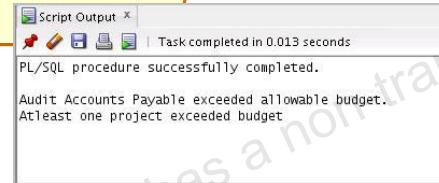
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the example in the slide, the FIRST method finds the smallest index number, and the NEXT method traverses the collection starting at the first index.

The check\_costs function checks the collection values and returns a TRUE if it encounters a project which has exceeded the budget constraint. The function returns a NULL if none of the projects have exceeded the budget constraint. The maximum budget allowed for a project element is defined by the C\_MAX\_ALLOWED constant in the function.

## Using Collection Methods

```
-- sample caller program to check_costs
set serveroutput on
DECLARE
    v_project_list typ_projectlist;
BEGIN
    v_project_list := typ_ProjectList(
        typ_Project (1, 'Dsgn New Expense Rpt', 3250),
        typ_Project (2, 'Outsource Payroll', 120000),
        typ_Project (3, 'Audit Accounts Payable', 14250000));
    IF manage_dept_proj.check_costs(v_project_list) THEN
        dbms_output.put_line('Atleast one project exceeded budget');
    ELSE
        dbms_output.put_line('All Projects accepted, fill out forms.');
    END IF;
END;
```



ORACLE

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

The code shown in the slide calls the `CHECK_COSTS` function (shown on the previous page). The `CHECK_COSTS` function accepts a varray parameter and returns a Boolean value.

A project with three elements is constructed and passed to the `CHECK_COSTS` function. The `CHECK_COSTS` function returns true, because the third element of the varray exceeds the value of the maximum allowed costs.

Although the sample caller program has the varray values hard-coded, you could have some sort of form interface where the user enters the values for projects and the form calls the `CHECK_COSTS` function.

## Manipulating Individual Elements

```
PROCEDURE update_a_project
  (p_deptno NUMBER, p_new_project typ_Project, p_position NUMBER)
IS
  v_my_projects typ_ProjectList;
BEGIN
  v_my_projects := get_dept_project (p_deptno);
  v_my_projects.EXTEND;  --make room for new project
  /* Move varray elements by one position */
  FOR i IN REVERSE p_position..v_my_projects.LAST - 1
  LOOP
    v_my_projects(i + 1) := v_my_projects(i);
  END LOOP;
  v_my_projects(p_position) := p_new_project; -- insert new one
  UPDATE department SET projects = v_my_projects
  WHERE dept_id = p_deptno;
END update_a_project;
```



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

You must use PL/SQL procedural statements to reference the individual elements of a varray in an INSERT, an UPDATE, or a DELETE statement. In the example shown in the slide, the UPDATE\_A\_PROJECT procedure inserts a new project into a department's project list at a given position, and then updates the PROJECTS column with the newly entered value that is placed within the old collection values.

This code essentially shuffles the elements of a project so that you can insert a new element in a particular position.

## Manipulating Individual Elements

```
-- check the table prior to the update:  
SELECT d2.dept_id, d2.name, d1.*  
FROM department d2, TABLE(d2.projects) d1;
```

DEPT_ID	NAME	PROJECT_NO	TITLE	COST
1	10 Executive Administration	1001	Travel Monitor	400000
2	10 Executive Administration	1002	Open World	1000000
3	20 Information Technology	2001	DB11gR2	900000

```
-- caller program to update_a_project  
BEGIN  
    manage_dept_proj.update_a_project(20,  
        typ_Project(2002, 'AQM', 80000), 2);  
END;
```

Script Output X  
Task completed in 0.004 seconds  
PL/SQL procedure successfully completed.

```
-- check the table after the update:  
SELECT d2.dept_id, d2.name, d1.*  
FROM department d2, TABLE(d2.projects) d1;
```

DEPT_ID	NAME	PROJECT_NO	TITLE	COST
1	10 Executive Administration	1001	Travel Monitor	400000
2	10 Executive Administration	1002	Open World	1000000
3	20 Information Technology	2001	DB11gR2	900000
4	20 Information Technology	2002	AQM	80000

ORACLE

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

To execute the procedure, pass the department number to which you want to add a project, the project information, and the position where the project information is to be inserted.

The third code box shown in the slide identifies that a project element should be added to the second position for project 2002 in department 20.

If you execute the following code, the AQM project element is shuffled to position 3 and the CQN project element is inserted at position 2.

```
BEGIN  
    manage_dept_proj.update_a_project(20,  
        typ_Project(2003, 'CQN', 85000), 2);  
END;
```

## Querying a Collection Using the TABLE Operator

A collection can be queried if the following are true:

- The collection type was created at the schema level
- The collection type was declared in a package specification.
- Using a TABLE function
  - Accepts a collection parameter
  - Enables developers to query the collection like a table



ORACLE®

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

A collection can be queried if the following are true:

- The data type of the collection was either created at the schema level or declared in a package specification.
- In the query `FROM` clause, the collection appears in `table_collection_expression` as the argument of the `TABLE` operator. The data type of the collection element is either a scalar data type that SQL supports or a record type in which every field has a data type that SQL supports.
- `TABLE` function – A `TABLE` function accepts a Collection type parameter and returns a collection instance which can be queried like table. You can use the `TABLE` function in the `FROM` clause of the select query.

## Querying a Collection with the TABLE Operator

```
CREATE OR REPLACE function GET_EMPS(P_JOB_ID JOBS.JOB_ID%TYPE)
RETURN EMP_TYPE_LIST
IS
...
CURSOR C_EMP(C_JOB_ID JOBS.JOB_ID%TYPE) IS
SELECT DEPARTMENT_ID, LAST_NAME, SALARY, J.JOB_ID, JOB_TITLE
FROM EMPLOYEES E, JOBS J
WHERE E.JOB_ID=J.JOB_ID AND J.JOB_ID=C_JOB_ID;
BEGIN
OPEN C_EMP(P_JOB_ID);
...
RETURN EMPS;
END;
/
SELECT * FROM TABLE(GET_EMPS('IT_PROG'))
/
SELECT D.DEPARTMENT_NAME,E.*
FROM
TABLE(get_emps('IT_PROG')) E, DEPARTMENTS D
WHERE E.DEPARTMENT_ID=D.DEPARTMENT_ID
/
```



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

The code example in the slide shows the technique of querying a collection returned by a PL/SQL function by using the TABLE operator.

The function GET\_EMPS returns a nested table of type EMP\_TYPE\_LIST, The SQL queries use TABLE operator to read data from the nested table.

The complete example is illustrated on the following page.

Execute the following statements using your HR connection:

```

DROP TYPE EMP_TYPE;
DROP TYPE EMP_TYPE_LIST ;
CREATE OR REPLACE TYPE EMP_TYPE AS OBJECT
(
    DEPARTMENT_ID          NUMBER(6),
    LAST_NAME               VARCHAR2(25),
    SALARY                  NUMBER(8,2),
    JOB_ID                  VARCHAR2(30),
    JOB_TITLE                VARCHAR2(30));
/
CREATE OR REPLACE TYPE EMP_TYPE_LIST AS TABLE OF EMP_TYPE;
/
CREATE OR REPLACE function GET_EMPS(P_JOB_ID  JOBS.JOB_ID%TYPE)
RETURN EMP_TYPE_LIST
IS
EMPS  EMP_TYPE_LIST:=EMP_TYPE_LIST();
R EMP_TYPE:=EMP_TYPE(NULL,NULL,NULL,NULL,NULL);
i pls_integer:=0;
CURSOR C_EMP(C_JOB_ID  JOBS.JOB_ID%TYPE) IS
SELECT DEPARTMENT_ID,LAST_NAME,SALARY,J.JOB_ID,JOB_TITLE
FROM EMPLOYEES E, JOBS J
WHERE E.JOB_ID=J.JOB_ID AND J.JOB_ID=C_JOB_ID;
BEGIN
OPEN C_EMP(P_JOB_ID);
LOOP
FETCH C_EMP INTO
R.DEPARTMENT_ID,R.LAST_NAME,R.SALARY,R.JOB_ID,R.JOB_TITLE;
EXIT WHEN C_EMP%NOTFOUND;
i:=i+1;
emps.extend;
EMPS(I):=R;
END LOOP;
RETURN EMPS;
END;

```

Check the result:

```

SELECT * FROM TABLE(GET_EMPS('IT_PROG'));
SELECT D.DEPARTMENT_NAME,E.*
FROM
TABLE(get_emps('IT_PROG')) E, DEPARTMENTS D
WHERE E.DEPARTMENT_ID=D.DEPARTMENT_ID;

```

	DEPARTMENT_ID	LAST_NAME	SALARY	JOB_ID	JOB_TITLE
1	60 Hunold	9000	IT_PROG	Programmer	
2	60 Ernst	6000	IT_PROG	Programmer	
3	60 Austin	4800	IT_PROG	Programmer	
4	60 Pataballa	4800	IT_PROG	Programmer	
5	60 Lorentz	4200	IT_PROG	Programmer	

## Lesson Agenda

- Working with Collections
- Collection exceptions
- Summarizing Collections
- PL/SQL Bind types



ORACLE®

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

## Collection Exceptions

Common exceptions with collections:

- COLLECTION\_IS\_NULL
- NO\_DATA\_FOUND
- SUBSCRIPT\_BEYOND\_COUNT
- SUBSCRIPT\_OUTSIDE\_LIMIT
- VALUE\_ERROR



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

In most cases, if you reference a nonexistent collection element, PL/SQL raises a predefined exception. Following table has a list of commonly raised exceptions, while using collections:

Exception	Raised when:
COLLECTION_IS_NULL	You try to operate on an atomically null collection.
NO_DATA_FOUND	A subscript designates an element that was deleted.
SUBSCRIPT_BEYOND_COUNT	A subscript exceeds the number of elements in a collection.
SUBSCRIPT_OUTSIDE_LIMIT	A subscript is outside the legal range.
VALUE_ERROR	A subscript is null or not convertible to an integer.

## Avoiding Collection Exceptions: Example

Common exceptions with collections:

```
DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    nums NumList;          -- atomically null
BEGIN
    /* Assume execution continues despite the raised exceptions.
 */
    nums(1) := 1;           -- raises COLLECTION_IS_NULL
    nums := NumList(1,2);   -- initialize table
    nums(NULL) := 3;        -- raises VALUE_ERROR
    nums(0) := 3;           -- raises SUBSCRIPT_OUTSIDE_LIMIT
    nums(3) := 3;           -- raises SUBSCRIPT_BEYOND_COUNT
    nums.DELETE(1);         -- delete element 1
    IF nums(1) = 1 THEN    -- raises NO_DATA_FOUND
    ...

```



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

The partial code in the slide illustrates various situations when an exception can be raised.

- **COLLECTION\_IS\_NULL** is raised you are assigning a value to be stored at a given index. The collection here is **NULL** because in the declare section we have not initialized the collection. Before assigning or updating the value you have to initialize the collection.  
We initialize the collection in the following statement using a constructor
- A **VALUE\_ERROR** exception is raised when you are trying to access a **NULL** index of the collection
- **SUBSCRIPT\_OUTSIDE\_LIMIT** exception is raised when you try to access a subscript that is outside the legal range.
- **SUBSCRIPT\_BEYOND\_COUNT** exception is raised because you are trying to access an index 3, whereas you have initialized the collection with only two values. Based on the initialization you can have an index value of 2.  
You can increase the size of the collection using **EXTEND** method.
- **NO\_DATA\_FOUND** exception is raised when you are trying to access an element that was deleted.

## Lesson Agenda

- Working with Collections
- Collection exceptions
- Summarizing Collections
- PL/SQL Bind types



ORACLE®

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

## Listing Characteristics for Collections

Collection Type	Number of Elements	Index Type	Dense or Sparse	Uninitialized status	Where defined	Can be ADT attribute data type
Associative array (index-by table)	Unspecified	String or PLS_INTEGRER	Either	Empty	In PL/SQL block or package	No
VARRAY(variable-sized array)	Specified	Integer	Always dense	Null	In PL/SQL block or package or schema level	Only if defined at schema level
Nested table	Unspecified	Integer	Starts dense can become sparse	Null	In PL/SQL block or package or schema level	Only if defined at schema level



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

- Use associative arrays when you need:
  - To collect information of unknown volume
  - Flexible subscripts (negative, non sequential, or string based)
  - To pass the collection to and from the database server (Use associative arrays with the bulk constructs.)
- Use nested tables when you need:
  - Persistence
  - To pass the collection as a parameter

### Choosing Between Nested Tables and Varrays

- Use varrays when:
  - The number of elements is known in advance
  - The elements are usually all accessed in sequence
- Use nested tables when:
  - The index values are not consecutive
  - There is no predefined upper bound for the index values
  - You need to delete or update some, not all, elements simultaneously
  - You would usually create a separate lookup table with multiple entries for each row of the main table and access it through join queries

## Lesson Agenda

- Working with Collections
- Collection exceptions
- Summarizing Collections
- PL/SQL Bind types



ORACLE®

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

## PL/SQL Bind Types

- PL/SQL bind types refer to the data types which can be passed as parameters to PL/SQL program units while invoking them from dynamic SQL or clients such as JDBC or OCI.
- In 12c, you can use the following types as parameters while invoking PL/SQL program units
  - Boolean
  - Records
  - Collections



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

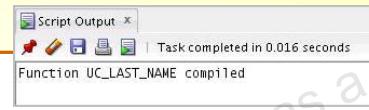
In 12.1, Oracle Database extended its support for PL/SQL-specific data types in SQL statements. Before Oracle Database 12c, values with PL/SQL-only data types (for example, BOOLEAN, associative array, and record) could not be bound from client programs (OCI or JDBC) or from static and native dynamic SQL issued from PL/SQL in the server. Whenever you had to bind a PL/SQL expression by using EXECUTE IMMEDIATE or DBMS\_SQL, the data type of that expression had to be a known SQL data type. Specifically, you could not bind boolean and user-defined types declared in a package specification, including records and collections.

Oracle Database 12c, makes it possible to bind values with PL/SQL-only data types to anonymous blocks (which are SQL statements), PL/SQL function calls in SQL queries and CALL statements, and the TABLE operator in SQL queries.

## Subprogram with a BOOLEAN Parameter

```
CREATE OR REPLACE FUNCTION uc_last_name (
    employee_id_in    IN employees.employee_id%TYPE,
    upper_in          IN BOOLEAN)
RETURN employees.last_name%TYPE
IS
    l_return    employees.last_name%TYPE;
BEGIN
    SELECT last_name
        INTO l_return
        FROM employees
       WHERE employee_id = employee_id_in;

    RETURN CASE WHEN upper_in THEN UPPER (l_return) ELSE
        l_return END;
END;
/
```



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

The slide shows a subprogram which accepts a boolean parameter along with a number parameter.

The function returns the last name of the employee whose `employee_id` is passed as parameter to the function `uc_last_name`.

Based on the boolean value passed as parameter the `last_name` value is returned either in upper case or lower case.

## Subprogram with a BOOLEAN parameter

```

DECLARE
    b BOOLEAN := TRUE;
BEGIN
    FOR rec IN (SELECT uc_last_name(employee_id, b) lname
                 FROM employees
                WHERE department_id = 10)
    LOOP
        DBMS_OUTPUT.PUT_LINE (rec.lname);
    END LOOP;
END;
/

```

PL/SQL procedure successfully completed.  
WHALEN



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

The code shown in the slide invokes the `uc_last_name` function in an SQL query. The `boolean` value which is passed as parameter to the function is initialized in the PL/SQL block.

This was not possible in earlier versions of Oracle Database. Oracle Database 12c allows more PL/SQL data types to cross PL/SQL to SQL interface.

Following is the code which allows you to pass records as parameters to the PL/SQL Program units:

```

CREATE OR REPLACE PACKAGE names_pkg AUTHID CURRENT_USER
AS
    TYPE names_t
    IS TABLE OF VARCHAR2 (100)
        INDEX BY PLS_INTEGER;

    PROCEDURE display_names (
        names_in    IN names_t);
END names_pkg;
/

```

## Subprogram with a RECORD parameter (continued)

```

CREATE OR REPLACE PACKAGE BODY names_pkg
AS
    PROCEDURE display_names (
        names_in    IN names_t)
    IS
    BEGIN
        FOR indx IN 1 .. names_in.COUNT
        LOOP
            DBMS_OUTPUT.put_line (
                names_in (indx));
        END LOOP;
    END;
END names_pkg;
/
DECLARE
    l_names    names_pkg.names_t;
BEGIN
    l_names (1) := 'Loey';
    l_names (2) := 'Dylan';
    l_names (3) := 'Indigo';
    l_names (4) := 'Saul';
    l_names (5) := 'Sally';
    EXECUTE IMMEDIATE
        'BEGIN names_pkg.display_names (:names); END; '
        USING l_names;
    FOR rec
        IN (SELECT * FROM TABLE (l_names))
    LOOP
        DBMS_OUTPUT.put_line (
            rec.COLUMN_VALUE);
    END LOOP;
END;

```

## Quiz



Which of the following collection method is used for traversing a collection?

- a. EXISTS
- b. COUNT
- c. LIMIT
- d. FIRST

ORACLE

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

**Answer: d**

## Quiz



A PL/SQL anonymous block, a SQL CALL statement, or a SQL query can invoke a PL/SQL function that has parameters of the type:

- a. Boolean
- b. Record declared in the package specification
- c. Collection declared in the package specification
- d. All of the above

ORACLE®

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



**Answer: d**

## Summary

In this lesson, you should have learned how to:

- Access collection elements
- Use collection methods in PL/SQL
- Identify raised exceptions with collections
- Decide which collection type is appropriate for each scenario



ORACLE®

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

Collections are a grouping of elements, all of the same type. The types of collections are nested tables, varrays, and associative arrays. You can define nested tables and varrays in the database. Nested tables, varrays, and associative arrays can be used in a PL/SQL program.

When using collections in PL/SQL programs, you can access the collection elements, use predefined collection methods, and use the exceptions that are commonly encountered with collections.

There are guidelines for using collections effectively and for determining which collection type is appropriate under specific circumstances.

## Practice 4: Overview

This practice covers using collections.



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

In this practice, you write a PL/SQL package to manipulate the collection.  
Use the OE schema for this practice.

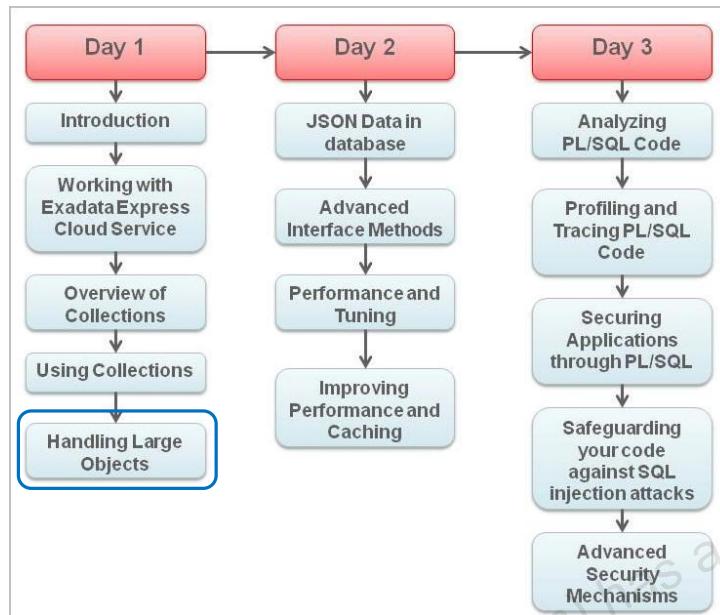
# 5

## Handling Large Objects

ORACLE

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

## Course Agenda



ORACLE

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



## Objectives

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

- Create and maintain LOB data types
- Differentiate between internal and external LOBS
- Use the DBMS\_LOB package
- Describe the use of temporary LOBS
- Describe Secure File LOB



ORACLE

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

This lesson describes the characteristics of the LOB data types. Examples, syntax, and issues regarding the LOB types are also presented.

You will learn about various types of LOB data types and packages which support manipulating LOBs.

## Lesson Agenda

- Introduction to LOBs
- Using DBMS\_LOB package
- Working with BFILEs
- Working with CLOBs
- Reading LOBs from the database
- Using temporary LOBs
- Using SecureFile LOBs

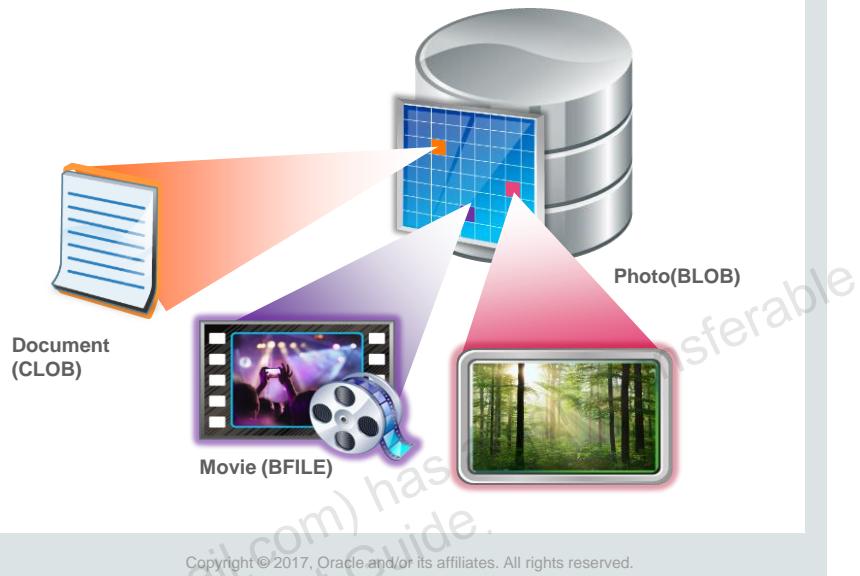


ORACLE®

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

## What Is a LOB?

- Large Objects(LOBs) refer to a set of data types which are designed to hold large amounts of data.
- There are four LOB data types:
  - BLOB
  - CLOB
  - NCLOB
  - BFILE



ORACLE

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

A LOB data type is used to store large volumes of data. You can store structured, semi-structured or unstructured data .

You can use LOB data types to define column types which can store documents, videos and so on.

There are four large object data types:

- **BLOB** represents a binary large object, such as a video clip or a picture.
- **CLOB** represents a character large object.
- **NCLOB** represents a multiple-byte character large object.
- **BFILE** represents a binary file stored in an OS binary file outside the database. The **BFILE** column or attribute stores a file locator that points to the external file.

## Types of LOBs

- Based on the physical location of the large object, there are two types of LOBs
  - Internal LOBs
  - External LOBs



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

You can store LOBs either in the table space of the database or external to the database in the operating system's file system.

Based on the actual location of the large object, there are two types of large objects – Internal large objects and External large objects.

Internal LOBs are stored in table spaces within the database. The SQL data types BLOB, CLOB and NCLOB are used for supporting internal large objects.

External LOBs are stored in the operating system files, outside the database table spaces. BFILE is the data type used to support external large objects.

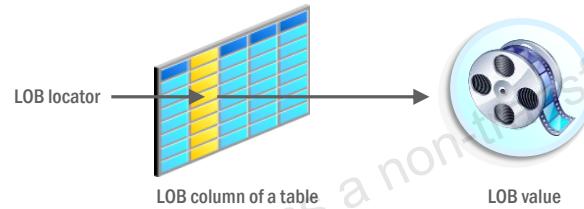
You can define a column type of LOBs in tables for both internal and external LOBs.

Differences between Internal and External LOBs:

Internal LOBs	External LOBs
LOB values are stored in the table space of the database	LOB values are stored as operating system files outside the database
CLOB, BLOB and NCLOB types are used for internal LOBs	BFILE type is used for external LOBs
Internal LOBs can be part of transactions	External LOBs cannot be part of transactions
In case of an internal LOB there is a distinct locator for each value	In case of external LOB multiple LOB locators can refer to the same LOB

## LOB Locators and LOB Values

- There are two components for every LOB instance:
  - LOB locator
  - LOB value
- LOB locator is a reference to the LOB value.
- A LOB type column in a table has LOB locators.
- LOB value is the actual LOB data which can be stored externally or internally to the database.



ORACLE

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

There are two parts to a LOB:

- **LOB value:** The data that constitutes the real object being stored
- **LOB locator:** A pointer to the location of the LOB value that is stored in the database

Irrespective of the type of the LOB, a LOB locator is stored for value in a LOB data type column. You can think of a LOB locator as a pointer to the actual location of the LOB value.

A LOB column does not contain the data; it contains the locator of the LOB value.

When a user creates an internal LOB, the value is stored in the LOB segment and a locator to the out-of-line LOB value is placed in the LOB column of the corresponding row in the table. External LOBs store the data outside the database, so only a locator to the LOB value is stored in the table.

## Lesson Agenda

- Introduction to LOBs
- Using DBMS\_LOB package
- Working with BFILEs
- Working with CLOBs
- Reading LOBs from the database
- Using temporary LOBs
- Using SecureFile LOBs



ORACLE®

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

## DBMS\_LOB Package

- DBMS\_LOB is an Oracle supplied package.
- Has subprograms which can read and modify LOBs.
- DBMS\_LOB subprograms perform operations based on LOB locators.



ORACLE®

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

DBMS\_LOB package has subprograms for working with LOBs. You can read and modify the BLOB, CLOB and NCLOB types of LOBs through the subprograms defined in DBMS\_LOB package. It has read only operations for BFILEs.

The DBMS\_LOB functions and procedures can be broadly classified into two types: mutators and observers:

- The mutators can modify LOB values: APPEND, COPY, ERASE, TRIM, WRITE, FILECLOSE, FILECLOSEALL, and FILEOPEN.
- The observers can read LOB values: COMPARE, FILEGETNAME, INSTR, GETLENGTH, READ, SUBSTR, FILEEXISTS, and FILEISOPEN.

DBMS\_LOB provides routines to access and manipulate internal and external LOBs:

- Modify LOB values: APPEND, COPY, ERASE, TRIM, WRITE, LOADFROMFILE
- Read or examine LOB values: GETLENGTH, INSTR, READ, SUBSTR
- Specific to BFILEs: FILECLOSE, FILECLOSEALL, FILEEXISTS, FILEGETNAME, FILEISOPEN, FILEOPEN

## Security Model of DBMS\_LOB Package

- DBMS\_LOB package is created as a SYS user.
- You can use the subprograms of as a non SYS user with applicable access rights.
- You can modify BLOBS, CLOBS and NCLOBs, but not BFILEs.
- You can access BFILEs through a DIRECTORY object.



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

A DBMS\_LOB subprogram called from an anonymous PL/SQL block is executed with the privileges of the current user.

DBMS\_LOB subprogram invoked from a stored procedure is executed with the privileges of the owner of the stored procedure.

## What Is a DIRECTORY Object?

- A DIRECTORY object enables secure access to the BFILEs .
- Specifies a logical alias name for a physical directory on the operating system's file system.
- Provides the flexibility to manage the locations of the BFILEs without hard coding the absolute path.
- Created by administrators or users with CREATE DIRECTORY privileges.



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

A DIRECTORY is a nonschema database object that enables the administration of access and usage of BFILEs in Oracle Database. To associate an OS file with a BFILE, you should first create a DIRECTORY object that is an alias for the full path name to the OS file.

The DIRECTORY object provides the flexibility to manage the locations of the files, instead of forcing you to hard-code the absolute path names of physical files in your applications.

The DIRECTORY object is created by the DBA (or a user with the CREATE ANY DIRECTORY privilege). The privileges may differ from those defined for the DIRECTORY object and could change after creation of the DIRECTORY object. Create DIRECTORY objects by using the following guidelines:

- Directories should point to paths that do not contain database files, because tampering with these files could corrupt the database.
- The CREATE ANY DIRECTORY and DROP ANY DIRECTORY system privileges should be used carefully and not granted to users indiscriminately.
- DIRECTORY objects are not schema objects; all are owned by SYS.
- Create the directory paths with appropriate permissions on the OS before creating the DIRECTORY object. Oracle does not create the OS path.
- If you migrate the database to a different OS, you may have to change the path value of the DIRECTORY object.

Information about the DIRECTORY object that you create by using the CREATE DIRECTORY command is stored in the DBA\_DIRECTORIES and ALL\_DIRECTORIES data dictionary views.

## Managing BFILEs: Role of a DBA

The DBA or the system administrator:

1. Creates an OS directory and supplies files
2. Creates a DIRECTORY object in the database
3. Grants the READ privilege on the DIRECTORY object to the appropriate database users



ORACLE®

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

Managing BFILEs requires coordination between the database administrator and the system administrator, and then between the developer and the user of the files.

The database or system administrator must perform the following privileged tasks:

1. Create the operating system (OS) directory (as an Oracle user), and set permissions so that the Oracle server can read the contents of the OS directory. Load files into the OS directory.
2. Create a database DIRECTORY object that references the OS directory.
3. Grant the READ privilege on the database DIRECTORY object to the database users that require access to it.

## Managing BFILEs: Role of a Developer

The developer or the user:

1. Creates an Oracle table with a column that is defined as a BFILE data type
2. Inserts rows into the table by using the BFILENAME function to populate the BFILE column
3. Writes a PL/SQL subprogram that declares and initializes a LOB locator, and reads BFILE



ORACLE

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

The designer, application developer, or user must perform the following tasks:

1. Create a database table containing a column that is defined as the BFILE data type.
2. Insert rows into the table by using the BFILENAME function to populate the BFILE column, associating the BFILE field to an OS file in the named DIRECTORY.
3. Write PL/SQL subprograms that:
  - a. Declare and initialize the BFILE LOB locator
  - b. Select the row and column containing the BFILE into the LOB locator
  - c. Read the BFILE with a DBMS\_LOB function by using the locator file reference

## Lesson Agenda

- Introduction to LOBs
- Using DBMS\_LOB package
- Working with BFILEs
- Working with CLOBS
- Reading LOBs from the database
- Using temporary LOBs
- Using SecureFile LOBs



ORACLE®

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

## Working on BFILEs

To use BFILEs in your application, you usually follow this workflow:

1. Create a DIRECTORY object as SYSDBA user.
2. Grant access to the application user to access DIRECTORY object.
3. Create and initialize a BFILE column in the table as the application user(non SYSDBA user).
4. Populate and access the data in BFILE column.



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

In order to use BFILEs in your application, you have to perform certain administrator tasks.

1. You have to create a DIRECTORY object in the database. You create the DIRECTORY object as a SYSDBA user or any other user who has CREATE DIRECTORY privileges.
2. Once you create the DIRECTORY object, you have to grant the application user ( OE in this case) access to the DIRECTORY object.
3. Connect as the application user (OE) and add BFILE column to table.
4. You populate the BFILE column by loading the files from the directory. You can use the subprograms defined in the DBMS\_LOB package for loading data into the BFILE column.

You can now use the BFILE column for various operations in the application. We will look into the implementation of these steps in the following slides.

## Preparing to Use BFILEs

1. Create an OS directory to store the physical data files:

```
mkdir /home/oracle/labs/DATA_FILES/MEDIA_FILES
```

2. Create a DIRECTORY object by using the CREATE DIRECTORY command:

```
CREATE OR REPLACE DIRECTORY data_files AS  
'/home/oracle/labs/DATA_FILES/MEDIA_FILES';
```

3. Grant the READ privilege on the DIRECTORY object to the appropriate users:

```
GRANT READ ON DIRECTORY data_files TO OE;
```



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

To use a BFILE within an Oracle table, you must have a table with a column of the BFILE data type. For the Oracle server to access an external file, the server must know the physical location of the file in the OS directory structure.

The database DIRECTORY object provides the means to specify the location of the BFILEs. Use the CREATE DIRECTORY command to specify the pointer to the location where your BFILEs are stored. You must have the CREATE ANY DIRECTORY privilege.

**Syntax definition:** CREATE DIRECTORY *dir\_name* AS *os\_path*;

In this syntax, *dir\_name* is the name of the directory database object, and *os\_path* specifies the location of the BFILEs.

The slide examples show the commands to set up:

- The physical directory (for example, /temp/data\_files) in the OS
- A named DIRECTORY object, called *data\_files*, that points to the physical directory in the OS
- The READ access right on the directory to be granted to users in the database that provides the privilege to read the BFILEs from the directory

**Note:** The value of the SESSION\_MAX\_OPEN\_FILES database initialization parameter, which is set to 10 by default, limits the number of BFILEs that can be opened in a session.

## Creating BFILE Columns in the Table

- Use the BFILENAME function to initialize a BFILE column. The function syntax is:

```
FUNCTION BFILENAME(directory_alias IN VARCHAR2,
                   filename IN VARCHAR2)
RETURNS BFILE;
```

- Example:

- Add a BFILE column to a table:

```
ALTER TABLE customers ADD video BFILE;
```

- Update the column using the BFILENAME function:

```
UPDATE customers
SET video = BFILENAME('DATA_FILES', 'Winters.avi')
WHERE customer_id = 448;
```



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

The BFILENAME function is a built-in function that you use to initialize a BFILE column, by using the following two parameters:

- *directory\_alias* for the name of the database DIRECTORY object that references the OS directory containing the files
- *filename* for the name of the BFILE to be read

It creates a pointer (or LOB locator) to the external file stored in a physical directory and assigns it to the *directory\_alias*.

Populate the BFILE column by using the BFILENAME function in either of the following:

- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

You can use an UPDATE operation to change the pointer reference target of the BFILE. You can also initialize BFILE column to a NULL value and updated later.

After the BFILE columns are associated with a file, subsequent read operations on the BFILE can be performed by using the PL/SQL DBMS\_LOB package and OCI

## Populating a BFILE Column with PL/SQL

```
CREATE OR REPLACE PROCEDURE set_video(
    dir_alias VARCHAR2, custid NUMBER) IS
    filename VARCHAR2(40);
    file_ptr BFILE;
    CURSOR cust_csr IS
        SELECT cust_first_name FROM customers
        WHERE customer_id = custid FOR UPDATE;
BEGIN
    FOR rec IN cust_csr LOOP
        filename := rec.cust_first_name || '.gif';
        file_ptr := BFILENAME(dir_alias, filename);
        DBMS_LOB.FILEOPEN(file_ptr);
        UPDATE customers SET video = file_ptr
        WHERE CURRENT OF cust_csr;
        DBMS_OUTPUT.PUT_LINE('FILE: ' || filename ||
            ' SIZE: ' || DBMS_LOB.GETLENGTH(file_ptr));
        DBMS_LOB.FILECLOSE(file_ptr);
    END LOOP;
END set_video;
```



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

The slide example demonstrates populating the BFILE column. It is a PL/SQL procedure called `set_video`, which accepts the name of the directory, and a customer ID. The procedure performs the following tasks:

- Uses a cursor FOR loop to obtain each customer record.
- Sets the filename by appending .gif to the customer's first\_name.
- Creates a LOB locator for the BFILE in the file\_ptr variable.
- Calls the DBMS\_LOB.FILEOPEN procedure to verify whether the file exists, and to determine the size of the file by using the DBMS\_LOB.GETLENGTH function.
- Executes an UPDATE statement to write the BFILE locator value to the video BFILE column.
- Displays the file size returned from the DBMS\_LOB.GETLENGTH function.
- Closes the file by using the DBMS\_LOB.FILECLOSE procedure.

Suppose that you execute the following call:

```
EXECUTE set_video ('DATA_FILES', 844)
```

The sample result is:

```
FILE: Alice.gif SIZE: 2619802
```

## Using data in the BFILE Column

The DBMS\_LOB.FILEEXISTS function can check whether the file exists in the OS. The function:

- Returns 0 if the file does not exist
- Returns 1 if the file does exist

```
CREATE OR REPLACE FUNCTION get_filesize(p_file_ptr IN
OUT BFILE)
RETURN NUMBER IS
  v_file_exists BOOLEAN;
  v_length NUMBER:= -1;
BEGIN
  v_file_exists := DBMS_LOB.FILEEXISTS(p_file_ptr) = 1;
  IF v_file_exists THEN
    DBMS_LOB.FILEOPEN(p_file_ptr);
    v_length := DBMS_LOB.GETLENGTH(p_file_ptr);
    DBMS_LOB.FILECLOSE(p_file_ptr);
  END IF;
  RETURN v_length;
END;
/
```



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

The set\_video procedure on the previous page terminates with an exception if a file does not exist. To prevent the loop from prematurely terminating, you could create a function, such as get\_filesize, to determine whether a given BFILE locator references a file that actually exists on the server's file system. The DBMS\_LOB.FILEEXISTS function accepts the BFILE locator as a parameter and returns an INTEGER with:

- A value 0 if the physical file does not exist
- A value 1 if the physical file exists

If the BFILE parameter is invalid, one of the following three exceptions may be raised:

- NOEXIST\_DIRECTORY if the directory does not exist
- NOPRIV\_DIRECTORY if the database processes do not have privileges for the directory
- INVALID\_DIRECTORY if the directory was invalidated after the file was opened

In the get\_filesize function, the output of the DBMS\_LOB.FILEEXISTS function is compared with the value 1 and the result of the condition sets the BOOLEAN variable file\_exists. The DBMS\_LOB.FILEOPEN call is performed only if the file exists, thereby preventing unwanted exceptions from occurring. The get\_filesize function returns a value of -1 if a file does not exist; otherwise, it returns the size of the file in bytes. The caller can take appropriate action with this information.

## Lesson Agenda

- Introduction to LOBs
- Using DBMS\_LOB package
- Working with BFILEs
- Working with CLOBS
- Reading LOBs from the database
- Using temporary LOBs
- Using SecureFile LOBs



ORACLE®

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

## Working on CLOBs

To use CLOBs in your application, you usually follow this workflow :

1. Create a table space as a SYSDBA user.
2. Create a CLOB column in the table as the application user(non SYSDBA) .
3. Initialize the CLOB columns.
4. Populate and access the data in CLOB columns.
5. Write Data to a CLOB column.



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

You can store BLOBs, CLOBs and NCLOBs in the database as internal LOBs. Internal LOBs are stored in the table space within the database.

1. To have an internal LOB column in a table, you have to define the table space first. You can define the table space as SYSDBA user.
2. Once you have the table space defined you can create internal LOB columns in the database table.
3. Initialize or populate the columns.
4. Load data into the LOB columns.
5. Perform operations on the B/CLOB columns.

BFILEs are read only files therefore you cannot update the values in the BFILE columns. You can perform write and update operations on the internal LOB columns. We will look into the implementation of these steps in the following slides.

## Initializing LOB Columns Added to a Table

- Add the LOB columns to an existing table by using ALTER TABLE:

```
ALTER TABLE customers  
ADD (resume CLOB, picture BLOB);
```

- Create a tablespace where you will put a new table with the LOB columns:

```
CREATE TABLESPACE lob_tbs1  
DATAFILE 'lob_tbs1.dbf' SIZE 800M REUSE  
EXTENT MANAGEMENT LOCAL  
UNIFORM SIZE 64M  
SEGMENT SPACE MANAGEMENT AUTO;
```



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

You can create an internal LOB type column in a table either while creating the table or by using ALTER TABLE command. The example in the slide is adding two LOB columns to the table customers. You can also add LOB columns to a table while creating the table using CREATE command.

The internal LOBs are stored in a table space within the database. Therefore while creating columns of internal LOB data types, it is recommended that you create a table space first. You have to create the table space as a SYSDBA user.

## Initializing LOB Columns Added to a Table

Initialize the column LOB locator value with the `DEFAULT` option or the DML statements by using:

- `EMPTY_CLOB()` function for a CLOB column
- `EMPTY_BLOB()` function for a BLOB column

```
CREATE TABLE customer_profiles (
    id NUMBER,
    full_name      VARCHAR2(45),
    resume         CLOB DEFAULT EMPTY_CLOB(),
    picture        BLOB DEFAULT EMPTY_BLOB()
    LOB(picture) STORE AS BASICFILE
    (TABLESPACE lob_tbs1);
```



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

The contents of a LOB column are stored in the LOB segment, whereas the column in the table contains only a reference to that specific storage area called the LOB locator. In PL/SQL, you can define a variable of the LOB type, which contains only the value of the LOB locator. You can initialize the LOB locators by using the following functions:

- `EMPTY_CLOB()` function to a LOB locator for a CLOB column
- `EMPTY_BLOB()` function to a LOB locator for a BLOB column

These functions create the LOB locator pointing to an empty LOB. You can use the `DBMS_LOB` package subroutines to populate the content.

The example in the slide shows that you can use the `EMPTY_CLOB()` and `EMPTY_BLOB()` functions in the `DEFAULT` option in a `CREATE TABLE` statement. Thus, the LOB locator values are populated in their respective columns when a row is inserted into the table and the LOB columns were not specified in the `INSERT` statement.

The `CUSTOMER_PROFILES` table is created. The `PICTURE` column holds the LOB data in the BasicFile format, because the storage clause identifies the format.

## Populating LOB Columns

- Insert a row into a table with LOB columns:

```
INSERT INTO customer_profiles
  (id, full_name, resume, picture)
VALUES (164, 'Charlotte Kazan', EMPTY_CLOB(), NULL);
. . .
```

- Initialize a LOB by using the EMPTY\_BLOB() function:

```
UPDATE customer_profiles
SET resume = 'Date of Birth: 8 February 1951',
picture = EMPTY_BLOB()
WHERE id = 164;
```

- Update a CLOB column:

```
UPDATE customer_profiles
SET resume = 'Date of Birth: 1 June 1956'
WHERE id = 150;
```



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

You can use the special EMPTY\_BLOB() and EMPTY\_CLOB() functions in INSERT or UPDATE statements of SQL DML to initialize a NULL or non-NUL internal LOB to empty. To populate a LOB column, perform the following steps:

- Initialize the LOB column to a non-NUL value—that is, set a LOB locator pointing to an empty or populated LOB value. This is done by using the EMPTY\_BLOB() and EMPTY\_CLOB() functions.
- Populate the LOB contents by using the DBMS\_LOB package routines.

However, as shown in the slide examples, the two UPDATE statements initialize the resume LOB locator value and populate its contents by supplying a literal value. This can also be done in an INSERT statement. A LOB column can be updated to:

- Another LOB value
- A NULL value
- A LOB locator with empty contents by using the EMPTY\_\*LOB() built-in function

You can update the LOB by using a bind variable in embedded SQL. When assigning one LOB to another, a new copy of the LOB value is created. Use a SELECT FOR UPDATE statement to lock the row containing the LOB column before updating a piece of the LOB contents.

## Loading Data to a LOB Column

- Create the procedure to read the MS Word files and load them into the LOB column.

```
CREATE OR REPLACE PROCEDURE loadLOBFromBFILE_proc
  (p_dest_loc IN OUT BLOB, p_file_name IN VARCHAR2,
   p_file_dir IN VARCHAR2)
IS
  v_src_loc  BFILE := BFILENAME(p_file_dir, p_file_name);
  v_amount    INTEGER := 4000;
  offset      INTEGER := 1;
BEGIN
  DBMS_LOB.OPEN(v_src_loc, DBMS_LOB.LOB_READONLY);
  v_amount := DBMS_LOB.GETLENGTH(v_src_loc);
  DBMS_LOB.LOADBLOBFROMFILE(p_dest_loc, v_src_loc,
    v_amount,offset, offset);
  DBMS_LOB CLOSE(v_src_loc);
END loadLOBFromBFILE_proc;
```



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

The procedure shown in the slide is used to load data into the LOB column.

Before running the LOADLOBFROMBFILE\_PROC procedure, you must set a directory object that identifies where the LOB files are stored externally. In this example, the Microsoft Word documents are stored in the DATA\_FILES directory that was created earlier in this lesson.

The LOADBLOBFROMFILE procedure is used to read the LOB data into the PICTURE column in the CUSTOMER\_PROFILE table.

In this example:

- DBMS\_LOB.OPEN is used to open an external LOB in read-only mode.
- DBMS\_LOB.GETLENGTH is used to find the length of the LOB value.
- DBMS\_LOB.LOADBLOBFROMFILE is used to load the data from a file into an internal LOB.
- DBMS\_LOB CLOSE is used to close the external LOB.

## Writing Data to a LOB

Create the procedure to insert LOBS into the table:

```
CREATE OR REPLACE PROCEDURE write_lob
  (p_file IN VARCHAR2, p_dir IN VARCHAR2)
IS
  i      NUMBER;          v_fn VARCHAR2(15);
  v_ln VARCHAR2(40);     v_b   BLOB;
BEGIN
  DBMS_OUTPUT.ENABLE;
  DBMS_OUTPUT.PUT_LINE('Begin inserting rows...');

  FOR i IN 1 .. 30 LOOP
    v_fn:=SUBSTR(p_file,1,INSTR(p_file,'.')-1);
    v_ln:=SUBSTR(p_file,INSTR(p_file,'.')+1,LENGTH(p_file)-
                  INSTR(p_file,'.')-4);
    INSERT INTO customer_profiles
      VALUES (i, v_fn, v_ln, EMPTY_BLOB())
      RETURNING picture INTO v_b;
    loadLOBFromBFILE_proc(v_b,p_file, p_dir);
    DBMS_OUTPUT.PUT_LINE('Row'|| i ||' inserted.');
  END LOOP;
  COMMIT;
END write_lob;
```



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

You can initialize a BLOB column value by using the EMPTY\_BLOB() function as a default predicate.

The code shown in the slide uses the INSERT statement to initialize the locator. The LOADLOBFROMBFILE routine is then called and the LOB column value is inserted.

## Writing Data to a LOB

```
CREATE OR REPLACE DIRECTORY resume_files AS  
'/home/oracle/labs/DATA_FILES/RESUMES';
```

```
set serveroutput on  
set verify on  
set term on  
  
execute write_lob('karl.brimmer.doc',    'RESUME_FILES')  
execute write_lob('monica.petera.doc',    'RESUME_FILES')  
execute write_lob('david.sloan.doc',      'RESUME_FILES')
```



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

1. The Microsoft Word files are stored in the `/home/oracle/labs/DATA_FILES/RESUMES` directory.
2. To read them into the `PICTURE` column in the `CUSTOMER_PROFILES` table, the `WRITE_LOB` procedure is called and the name of the `.doc` files is passed as a parameter.

The output is similar to the following:

```
execute write_lob('karl.brimmer.doc', 'RESUME_FILES');
Begin inserting rows...
Row 1 inserted.

...
PL/SQL procedure successfully completed.

execute write_lob('monica.petera.doc', 'RESUME_FILES');
Begin inserting rows...
Row 1 inserted.

...
PL/SQL procedure successfully completed.

execute write_lob('david.sloan.doc', 'RESUME_FILES');
Begin inserting rows...
Row 1 inserted.

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

## Lesson Agenda

- Introduction to LOBs
- Using DBMS\_LOB package
- Working with BFILEs
- Working with CLOBS
- Reading LOBs from the database
- Using temporary LOBs
- Using SecureFile LOBs



ORACLE®

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

## Reading LOBs from the Table

```
CREATE OR REPLACE PROCEDURE lob_txt(file_name VARCHAR2,p_dir VARCHAR2 DEFAULT 'PLSQL_DIR')
IS
c CLOB:=null;
byte_count pls_integer;
fil BFILE:=BFILENAME(p_DIR,file_name);
v_dest_offset integer:=1;
v_src_offset integer:=1;
v_lang_context integer:=0;
v_warning integer;
BEGIN
c:=TO_CLOB(' ');
IF DBMS_LOB.FILEEXISTS(FIL)=1 then
DBMS_LOB.FILEOPEN(fil,DBMS_LOB.FILE_READONLY);
byte_count:=DBMS_LOB.GETLENGTH(fil);
DBMS_OUTPUT.PUT_LINE('The length of the file:'||byte_count);
DBMS_LOB.LOADCLOBFROMFILE
(dest_lob => c,src_bfile => fil,amount => byte_count,dest_offset => v_dest_offset
,src_offset => v_src_offset,bfile_csid => 0,lang_context => v_lang_context
,warning => v_lang_context);
DBMS_LOB.FILECLOSEALL;
INSERT INTO lob_text VALUES (lob_seq.nextval,c);
COMMIT;
ELSE
DBMS_OUTPUT.PUT_LINE('The file does not exist ');
END IF;
END;
/
```



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

The procedure given in the slide uploads a text file into a CLOB type column in a table.

To retrieve the records that were inserted, you can call the LOB\_TXT procedure, For example to retrieve the record for the file 'java.sql':

```
SET LONG 10000
set serveroutput on
exec lob_txt('java.sql')
select KEY,txt from lob_text;
```

## Updating LOB by Using DBMS\_LOB in PL/SQL

```
DECLARE
    v_lobloc CLOB;      -- serves as the LOB locator
    v_text    VARCHAR2(50) := 'Resigned = 5 June 2000';
    v_amount NUMBER ;   -- amount to be written
    v_offset INTEGER;   -- where to start writing
BEGIN
    SELECT resume INTO v_lobloc FROM customer_profiles
    WHERE id = 164 FOR UPDATE;
    v_offset := DBMS_LOB.GETLENGTH(v_lobloc) + 2;
    v_amount := length(v_text);
    DBMS_LOB.WRITE (v_lobloc, v_amount, v_offset, v_text);
    v_text := ' Resigned = 30 September 2000';
    SELECT resume INTO v_lobloc FROM customer_profiles
    WHERE id = 150 FOR UPDATE;
    v_amount := length(v_text);
    DBMS_LOB.WRITEAPPEND(v_lobloc, v_amount, v_text);
    COMMIT;
END;
```



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

In the example in the slide, the V\_LOBLOC variable serves as the LOB locator, and the AMOUNT variable is set to the length of the text that you want to add. The SELECT FOR UPDATE statement locks the row and returns the LOB locator for the RESUME LOB column. Finally, the PL/SQL WRITE package procedure is called to write the text into the LOB value at the specified offset. WRITEAPPEND procedure in the DBMS\_LOB appends to the existing LOB value.

## Selecting CLOB Values by Using SQL

- Query:

```
SELECT id, full_name , resume -- CLOB
FROM customer_profiles
WHERE id IN (164, 150);
```

- Output in SQL Developer:

ID	FULL_NAME	RESUME
1	164 Charlotte Kazan	Resigned = 5 June 2000
2	150 Harry Dean Fonda	Resigned = 30 September 2000



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

It is possible to see the data in a CLOB column by using a SELECT statement. It is not possible to see the data in a BLOB or BFILE column by using a SELECT statement in SQL\*Plus. You must use a tool that can display the binary information for a BLOB, as well as the relevant software for a BFILE—for example, you can use Oracle Forms.

## Selecting CLOB Values by Using DBMS\_LOB

- DBMS\_LOB.SUBSTR(lob, amount, start\_pos)
- DBMS\_LOB.INSTR(lob, pattern)

```
SELECT DBMS_LOB.SUBSTR(resume, 5, 18),
       DBMS_LOB.INSTR(resume, '=')
  FROM customer_profiles
 WHERE id IN (150, 164);
```

- SQL Developer



DBMS_LOB.SUBSTR(resume,5,18)	DBMS_LOB.INSTR(resume,'=')
1 Febru	41
2 June	37



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

### **DBMS\_LOB.SUBSTR**

Use DBMS\_LOB.SUBSTR to display part of a LOB. It is similar in functionality to the SUBSTR SQL function.

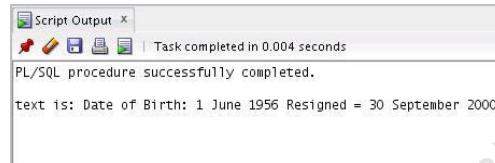
### **DBMS\_LOB.INSTR**

Use DBMS\_LOB.INSTR to search for information within the LOB. This function returns the numerical position of the information.

## Selecting CLOB Values in PL/SQL

```
SET LINESIZE 50 SERVEROUTPUT ON FORMAT WORD_WRAP

DECLARE
    text VARCHAR2(4001);
BEGIN
    SELECT resume INTO text
    FROM customer_profiles
    WHERE id = 150;
    DBMS_OUTPUT.PUT_LINE('text is: '|| text);
END;
/
```



ORACLE

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

The slide shows the code for accessing CLOB values that can be implicitly converted to VARCHAR2. When selected, the RESUME column value is implicitly converted from a CLOB to a VARCHAR2 to be stored in the TEXT variable.

## Removing LOBS

- Delete a row containing LOBS:

```
DELETE
FROM customer_profiles
WHERE id = 164;
```

- Disassociate a LOB value from a row:

```
UPDATE customer_profiles
SET resume = EMPTY_CLOB()
WHERE id = 150;
```



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

A LOB instance can be deleted (destroyed) by using the appropriate SQL DML statements. The `DELETE` SQL statement deletes a row and its associated internal LOB value. To preserve the row and destroy only the reference to the LOB, you must update the row by replacing the LOB column value with `NULL` or an empty string, or by using the `EMPTY_B/CLOB()` function.

**Note:** Replacing a column value with `NULL` and using `EMPTY_B/CLOB` are not the same. Using `NULL` sets the value to null; using `EMPTY_B/CLOB` ensures that nothing is in the column.

A LOB is destroyed when the row containing the LOB column is deleted, when the table is dropped or truncated, or when all LOB data is updated.

You must explicitly remove the file associated with a `BFILE` by using the OS commands.

## Quiz



The BFILE data type stores a locator to the physical file.

- a. True
- b. False

ORACLE

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

**Answer: a**

## Quiz



Use the BFILENAME function to:

- a. Create a BFILE column
- b. Initialize a BFILE column
- c. Update a BFILE column

ORACLE

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

**Answer: b, c**

## Quiz



Which of the following statements are true?

- a. You should initialize the LOB column to a non-NULL value by using the EMPTY\_BLOB () and EMPTY\_CLOB () functions.
- b. Populate the LOB contents by using the DBMS\_LOB package routines.
- c. It is possible to see the data in a CLOB column by using a SELECT statement.
- d. It is not possible to see the data in a BLOB or BFILE column by using a SELECT statement in SQL\*Plus.

ORACLE

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



**Answer: a, b, c, d**

## Lesson Agenda

- Introduction to LOBs
- Using DBMS\_LOB package
- Working with BFILEs
- Working with CLOBS
- Reading LOBs from the database
- Using temporary LOBs
- Using SecureFile LOBs



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

## Temporary LOBS

- Temporary LOBS:
  - Provide an interface to support creation of LOBS that act like local variables
  - Can be BLOBS, CLOBs, or NCLOBs
  - Are not associated with a specific table
  - Are created by using the DBMS\_LOB.CREATETEMPORARY procedure
  - Use DBMS\_LOB routines
- The lifetime of a temporary LOB is a session.
- Temporary LOBS are useful for transforming data in permanent internal LOBS.



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

Temporary LOBS provide an interface to support the creation and deletion of LOBS that act like local variables. Temporary LOBS can be BLOBS, CLOBs, or NCLOBs.

The following are the features of temporary LOBS:

- Data is stored in your temporary tablespace, not in tables.
- Temporary LOBS are faster than persistent LOBS, because they do not generate redo or rollback information.
- Temporary LOBS lookup is localized to each user's own session. Only the user who creates a temporary LOB can access it, and all temporary LOBS are deleted at the end of the session in which they were created.
- You can create a temporary LOB by using DBMS\_LOB.CREATETEMPORARY.

Temporary LOBS are useful when you want to perform a transformational operation on a LOB (for example, changing an image type from GIF to JPEG). A temporary LOB is empty when created and does not support the EMPTY\_B/CLOB functions.

Use the DBMS\_LOB package to use and manipulate temporary LOBS.

## Creating a Temporary LOB

The PL/SQL procedure to create and test a temporary LOB:

```
CREATE OR REPLACE PROCEDURE is_templob_open(
  p_lob IN OUT BLOB, p_retval OUT INTEGER) IS
BEGIN
  -- create a temporary LOB
  DBMS_LOB.CREATETEMPORARY(p_lob, TRUE);
  -- see if the LOB is open: returns 1 if open
  p_retval := DBMS_LOB.ISTEMPORARY (p_lob);
  DBMS_OUTPUT.PUT_LINE('You have created a
                        temporary LOB in the PL/SQL block');
  -- free the temporary LOB
  DBMS_LOB.FREETEMPORARY(p_lob);
END;
/
```



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

The example in the slide shows a user-defined PL/SQL procedure, `is_templob_open`, which creates a temporary LOB. This procedure accepts a LOB locator as input, creates a temporary LOB, opens it, and tests whether the LOB is open.

The `is_templob_open` procedure uses the procedures and functions from the `DBMS_LOB` package as follows:

- The `CREATETEMPORARY` procedure is used to create the temporary LOB.
- The `ISOPEN` function is used to test whether a LOB is open: This function returns the value 1 if the LOB is open.
- The `FREETEMPORARY` procedure is used to free the temporary LOB. Memory increases incrementally as the number of temporary LOBs grows, and you can reuse the temporary LOB space in your session by explicitly freeing temporary LOBs.

## Lesson Agenda

- Introduction to LOBs
- Using DBMS\_LOB package
- Working with BFILEs
- Working with CLOBs
- Reading LOBs from the database
- Using temporary LOBs
- Using SecureFile LOBs



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

## SecureFile LOBS

Oracle Database offers a reengineered large object (LOB) data type that:

- Improves performance
- Simplifies application development
- Offers advanced, next-generation functionality such as intelligent compression and transparent encryption



ORACLE

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

With SecureFile LOBS, the LOB data type is completely reengineered with dramatically improved performance, manageability, and ease of application development. This implementation also offers advanced, next-generation functionality such as intelligent compression and transparent encryption. This feature significantly strengthens the native content management capabilities of Oracle Database.

SecureFiles improve many aspects of managing LOBs, including disk format, caching, locking and space management algorithms. SecureFiles allow the LOBs to be deduplicated, compressed and encrypted using parameter settings.

## Storage of SecureFile LOBS

An efficient new storage paradigm is available in Oracle Database for LOB storage.

- If the SECUREFILE storage keyword appears in the CREATE TABLE statement, the new storage is used.
- If the BASICFILE storage keyword appears in the CREATE TABLE statement, the old storage paradigm is used.
- By default, the storage is BASICFILE, unless you modify the setting for the DB\_SECUREFILE parameter in the init.ora file.



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

Starting with Oracle Database 11g, you have the option of using the new SecureFile storage paradigm for LOBs. You can specify the use of the new paradigm by using the SECUREFILE keyword in the CREATE TABLE statement. If that keyword is left out, the old storage paradigm for basic file LOBs is used. This is the default behavior.

You can modify the init.ora file and change the default behavior for the storage of LOBs by setting the DB\_SECUREFILE initialization parameter. The following values are allowed:

- **ALWAYS:** Attempts to create all LOB files as SECUREFILES but creates any LOBs not in ASSM tablespaces as BASICFILE LOBs
- **FORCE:** Creates all LOBs in the system as SECUREFILE LOBs
- **PERMITTED:** The default; allows SECUREFILES to be created when specified with the SECUREFILE keyword in the CREATE TABLE statement
- **NEVER:** Creates LOBs that are specified as SECUREFILE LOBs as BASICFILE LOBs
- **IGNORE:** Ignores the SECUREFILE keyword and all SECUREFILE options

## Creating a SecureFile LOB

- Create a tablespace for the LOB data:

```
-- have your dba do this:  
CREATE TABLESPACE sf_tbs1  
  DATAFILE 'sf_tbs1.dbf' SIZE 1500M REUSE  
  AUTOEXTEND ON NEXT 200M  
  MAXSIZE 3000M  
  SEGMENT SPACE MANAGEMENT AUTO;
```

1

- Create a table to hold the LOB data:

```
CONNECT oe/oe  
CREATE TABLE customer_profiles_sf  
(id NUMBER,  
 first_name VARCHAR2 (40),  
 last_name VARCHAR2 (80),  
 profile_info BLOB)  
LOB(profile_info) STORE AS SECUREFILE  
(TABLESPACE sf_tbs1);
```

2



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

To create a column to hold a LOB that is a SecureFile, you:

- Create a tablespace to hold the data
- Define a table that contains a LOB column data type that is used to store the data in the SecureFile format

In the example shown in the slide:

1. The sf\_tbs1 tablespace is defined. This tablespace stores the LOB data in the SecureFile format. When you define a column to hold SecureFile data, you must have Automatic Segment Space Management (ASSM) enabled for the tablespace to support SecureFiles.
2. The CUSTOMER\_PROFILE\_SF table is created. The PROFILE\_INFO column holds the LOB data in the SecureFile format, because the storage clause identifies the format.

## Quiz



Which of the following statements are true about temporary LOBS?

- a. Data is stored in your temporary tablespace, not in tables.
- b. Temporary LOBS are faster than persistent LOBS, because they do not generate redo or rollback information.
- c. Only the user who creates a temporary LOB can access it.
- d. All temporary LOBS are deleted at the end of the session in which they were created.
- e. You can create a temporary LOB by using DBMS\_LOB.CREATETEMPORARY.

ORACLE

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

**Answer: a, b, c, d, e**

## Summary

In this lesson, you should have learned how to:

- Identify four built-in types for large objects: `BLOB`, `CLOB`, `NCLOB`, and `BFILE`
- Describe how `LOBs` replace `LONG` and `LONG RAW`
- Describe two storage options for `LOBs`:
  - Oracle server (internal `LOBs`)
  - External host files (external `LOBs`)
- Use the `DBMS_LOB` PL/SQL package to provide routines for `LOB` management
- Describe SecureFile `LOB`



ORACLE

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

There are four `LOB` data types:

- A `BLOB` is a binary large object.
- A `CLOB` is a character large object.
- An `NCLOB` stores multiple-byte national character set data.
- A `BFILE` is a large object stored in a binary file outside the database.

`LOBs` can be stored internally (in the database) or externally (in an OS file). You can manage `LOBs` by using the `DBMS_LOB` package and its procedure.

Temporary `LOBs` provide an interface to support the creation and deletion of `LOBs` that act like local variables.

You learned about SecureFile format, and also learned that the performance of SecureFile format `LOBs` is faster than the BasicFile format `LOBs`.

## Practice 5: Overview

This practice covers the following topics:

- Creating object types of the CLOB and BLOB data types
- Creating a table with the LOB data types as columns
- Using the DBMS\_LOB package to populate and interact with the LOB data
- Setting up the environment for LOBS



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

In this practice, you create a table with both the BLOB and CLOB columns. Then, you use the DBMS\_LOB package to populate the table and manipulate the data.

Use the OE schema for this practice.

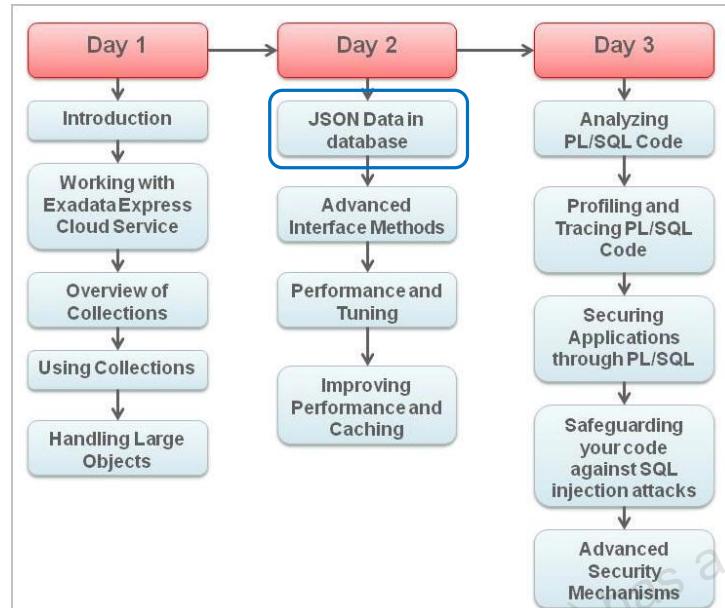
# 6

## Working with JSON Data

ORACLE

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

## Course Agenda



ORACLE®

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



## Objectives

After completing this lesson, you will be in a position to:

- Understand what is JSON data.
- Store JSON Data in database tables.
- Retrieve and perform operations on JSON data.
- Manipulate JSON data from a PL/SQL block.



ORACLE®

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

## Lesson Agenda

- JSON Data
- JSON data in Oracle Database
- SQL/JSON generation functions
- JSON Data in PL/SQL blocks
- PL/SQL object types



ORACLE®

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

## What Is JSON?

JSON(JavaScript Object Notation) is

- Programming language used by web browsers and web servers to communicate.
- Light weight data interchange format.
- Readable format for data structuring.
- Easy for humans to read and write.
- Easy for software to parse and generate.



ORACLE®

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

JSON (JavaScript Object Notation) is a general programming language widely used by web browsers and web servers for data interchange. It is an object notation that stores the objects in human readable format.

The JavaScript based notation of the objects is inherently compatible with JavaScript programs, eliminating the need of parsing and serializing while working with JavaScript programs. Apart from JavaScript a variety of programming languages can parse and generate JSON data.

JSON is defined in standards ECMA-404(JSON Data Interchange Format) and ECMA-262 (ECMA Script Language Specification)

## Structure of JSON Data

JSON represents an object as a set of property-value pairs.

- A JSON object is enclosed in a pair of braces - { }.
- The property name and property value pairs are enclosed in the braces.
- The property name and property value is separated by a colon.

```
{ property 1: value 1,  
  property 2:value 2,  
  property 3: value 3 ..}
```



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

JSON object is stored in text format. Each object is represented as a set of property and value pairs. The values assumed by each property can be an object, array, number, string, boolean type or a JSON object.

Each property value pair is separated by a comma.

When a property value is an array, then the array elements are enclosed in square brackets [].

## JSON Data: Example

```
{
  CUSTOMER_ID : 120,
  CUST_FIRST_NAME: 'Diane',
  CUST_LAST_NAME: 'Higgins',
  CUST_ADDRESS: {STREET: '113 Washington Sq N',
                 POSTAL_CODE: '48933',
                 CITY: 'Lansing',
                 STATE_PROVINCE: 'MI',
                 COUNTRY_ID:'US'}
  PHONE_NUMBERS: {'+1 517 123 4199'}
  NLS_LANGUAGE: 'US',
  NLS_TERRITORY: 'AMERICA',
  CREDIT_LIMIT: 200,
  ...
}
```



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

The object notation shows the name-value pairs of properties of the customer object.

Here is a JSON of a customer object.

```
{
  CUSTOMER_ID : 120
  CUST_FIRST_NAME: 'Diane'
  CUST_LAST_NAME: 'Higgins'
  CUST_ADDRESS: {STREET: '113 Washington Sq N',
                 POSTAL_CODE: '48933',
                 CITY: 'Lansing',
                 STATE_PROVINCE: 'MI',
                 COUNTRY_ID:'US'},
  PHONE_NUMBERS: {'+1 517 123 4199'}
  NLS_LANGUAGE: 'US'
  NLS_TERRITORY: 'AMERICA'
  CREDIT_LIMIT: 200
  CUST_EMAIL: Diane.Higgins@TANGER.EXAMPLE.COM
  ACCOUNT_MGR_ID: 145
  DATE_OF_BIRTH: '26-DEC-84'
  MARITAL_STATUS: 'SINGLE'
  GENDER: 'M'
  INCOME_LEVEL: '150,000 - 169,999'
  CREDIT_CARDS: [ {CARD_TYPE: MC, CARD_NUM: 11111111}, {CARD_TYPE: VISA, CARD_NUM:
  2323232323}
}
```

The customer object data has number fields, character strings and also some composite fields such as `customer_address`, phone numbers and credit cards.

Customer address is a composite data type which in turn have other fields such as street, postal code and so on. You can represent it as a JSON object in itself.

Phone numbers is also a composite data type where the number of phone numbers each customer might have is not known. You may define it as a varray or a nested table as per your requirement. Here you are representing as a JSON object.

Credit cards is also a composite data type where you are saving a pair of values, credit card type and card number. You may use an appropriate composite data type for storing credit card information in the table.

In the JSON object representation of data you can see how each of these composite data types are represented.

## Why JSON?

- JSON is a simple way of data interchange.
- Written in text format.
- It's interoperable.
- Inherently compatible with JavaScript.
- Easy for machines to parse and generate.



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

JSON's popularity as data interchange language among web components is due to it's simple human readable object notation. Most of the applications today have components which interact over the web. JSON being simple makes the interaction over web simpler. Though JSON is language independent, it uses conventions that are used in languages such as C, C++, C#, Java, JavaScript, Perl, Python and so on. These properties make it an ideal data interchange language.

JSON is written in text format with a simple syntax structure. Parsing a JSON object is fairly simple. The text only data format of JSON makes it compatible with any programming language.

A JavaScript program can convert JSON data into native JavaScript objects, without any parsers. Therefore it is inherently compatible with JavaScript.

The text only format of JSON makes it simpler for the machines to parse and generate JSON objects.

## Lesson Agenda

- JSON Data
- JSON data in Oracle Database
- SQL/JSON generation functions
- JSON Data in PL/SQL blocks
- PL/SQL object types



ORACLE®

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

## JSON Data in Oracle Database - Scenario

- Consider a scenario in Order Entry schema:
  - You want to collate the order history of each customer in your database. You can use the order history of the customer to predict the future requirements of the customer. Based on the predictions you can make product suggestions to the customer and enhance customer experience.
- However the order history of each customer may not be identical. The structure of data in the order history may change from customer to customer.
- You need something which allows flexibility in structure.

Storing order history as a JSON object would be a good design choice in this scenario.



ORACLE®

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

## JSON Data in Oracle Database

### Oracle Database

- Declarative querying language
- Reliable ACID transactions
- Complex Data processing

### JSON

- Flexible schema structure for data
- Efficient data interchange language for web applications

Oracle Database supports JSON natively with relational database features including transactions, indexing, declarative querying and views.



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

JSON data is generally stored in NoSQL databases such as Oracle NoSQL database and Oracle Berkeley DB, because they allow storage and retrieval of data that is not based on a schema. NoSQL databases however don't offer the reliability and consistency models of relational databases. Applications which require flexibility and consistency end up using relational databases and NoSQL databases in parallel.

Native support for JSON data by Oracle database obviates such workarounds. It provides all the benefits of the relational database features for use with JSON, including transactions, indexing, declarative querying and views.

## JSON Data in Oracle Database

- Oracle database provides various functions to interpret and generate JSON data.
- You can store JSON data as table columns.
- JSON data columns can coexist with non-JSON columns.
- You can use SQL to join JSON data with relational data.
- You can project JSON data to make it available for relational processes and tools.



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

In Oracle database, JSON data is stored in table columns like non-JSON data. JSON is stored using the common SQL data types `VARCHAR2`, `CLOB` and `BLOB`. JSON columns can co-exist with non-JSON columns.

You can write SQL queries on JSON columns just as you write on non-JSON columns.

**Note:** Oracle recommends that you always use an `is_json` check constraint to ensure that the column values are valid JSON instances.

## Creating a Table with JSON Column

- JSON column can assume any one of the following standard SQL data types in a table:
  - VARCHAR2
  - CLOB
  - BLOB
- Decide on the data type to be used based on the volume of data you would store in the column.
- Storing JSON data using standard SQL data types allows all the features of Oracle Database to be applied onto it.



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

You can use standard SQL data types to store JSON objects. You can use one of the two variants of VARCHAR2

VARCHAR2 (4000) where there is an upper limit of 4000 on the number of characters that can be stored in each value.

When your JSON object has more than 4000 characters you can use VARCHAR2 (32767). If there is a probability that the JSON object can have more than 32767 characters then you can use a CLOB or BLOB type of column.

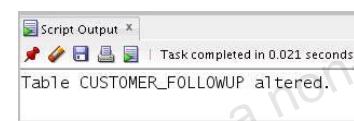
When you store JSON data using standard SQL types, you can manipulate JSON data just like any other type of data used in SQL. All the Oracle database features such as advanced replication are applicable to the JSON objects as well.

## Creating a Table with JSON Column

```
CREATE TABLE customer_followup AS (SELECT customer_id, cust_first_name,  
                                     cust_last_name, cust_address,  
                                     cust_email, account_mgr_id  
                                FROM customers);
```



```
ALTER TABLE customer_followup ADD (order_history VARCHAR2(4000));
```



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

The code in the slide is creating a table `customer_followup` based on the data in the `customers` table. You are copying a subset of the `customers` table information into the `customer_followup` table.

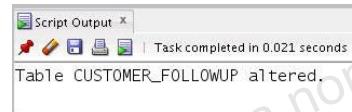
You can define a JSON data column in a table either while creating the table or later using an `ALTER TABLE` command. Here in the slide a JSON column is added to the table using the `ALTER TABLE` command. You can see that the new column is also a `VARCHAR2` column.

## JSON or Not?

When we store JSON as VARCHAR2 column, how do we differentiate a JSON and non-JSON column?

- JSON objects should have right structure.
- You can use `is_json` SQL/JSON condition to check the structure of JSON data.
- Oracle recommends that you define a `is_json` constraint on every JSON column.

```
ALTER TABLE customer_followup
ADD CONSTRAINT json_check CHECK(order_history IS JSON);
```



You've used VARCHAR2 data type for creating the JSON data column. The difference between a JSON VARCHAR2 column and non-JSON VARCHAR2 column is the structure of the text data. To store valid JSON data in the column you have to ensure that the data is structured according to JSON format.

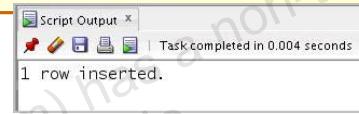
You can check whether the data stored in a column is valid JSON or not by using `is_json` condition.

Oracle recommends that you define a `is_json` constraint on the JSON column to ensure that the data is structured according to the JSON format.

## Inserting Data into JSON Columns

- You can insert data into JSON column using INSERT statement.

```
insert into customer_followup (customer_id, cust_first_name,cust_last_name,
cust_address, cust_email, account_mgr_id, order_history)
select customer_id, cust_first_name,
cust_last_name,cust_address,cust_email, account_mgr_id,
(select JSON_OBJECT('id' VALUE o.order_id,
                    'orderTotal' VALUE o.order_total,
                    'sales_rep_id' VALUE o.sales_rep_id)
 FROM orders o
 where o.customer_id = co.customer_id)
from customers co
where co.customer_id = 120;
```



ORACLE

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

You are using `INSERT` statement to add values to the table `customer_followup`. The JSON data is being inserted into the `order_history` column with appropriate format.

You are retrieving the data from the `orders` table and generating a JSON object from the data retrieved from the `orders` table. This JSON is then inserted into the `order_history` column of the `customer_followup` table.

You might've noticed that we are using a `JSON_OBJECT` function to generate JSON before adding the object to the column. `JSON_OBJECT` is one of the SQL/JSON generation functions. We will discuss these functions in the following slides.

## Lesson Agenda

- JSON Data
- JSON data in Oracle Database
- SQL/JSON generation functions
- JSON Data in PL/SQL blocks
- PL/SQL object types



ORACLE®

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

## SQL/JSON Generation Functions

You can use the following SQL/JSON functions to generate JSON data from non-JSON data:

- `JSON_OBJECT`
- `JSON_ARRAY`
- `JSON_OBJECTAGG`
- `JSON_ARRAYAGG`



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

You can use SQL/JSON generation functions to generate JSON data from non-JSON data. These functions make it simple to construct JSON data from SQL query. They return JSON objects and JSON arrays. You can also generate complex, hierarchical JSON documents by nesting calls to these functions.

It is always recommended to generate JSON data through these functions for the following reasons:

- Using string concatenation to generate JSON documents is error prone. These functions always return well formed JSON documents reducing the probability of error.
- By using sub queries you can generate an entire set of JSON documents using single SQL statement. The JSON generation operation is therefore optimized.

JSON values within the returned data are derived from SQL values in the input as follows:

- A SQL number is converted into a JSON number.
- A non-NULL and non-number SQL value is converted to a JSON string.
- A SQL NULL value is handled by the optional NULL-handling clause.

## JSON\_OBJECT Function

- JSON\_OBJECT constructs JSON objects from name-value pairs.
- The name-value pairs are explicitly provided as arguments to the function.

```
SELECT JSON_OBJECT('id' VALUE o.order_id,
                   'orderTotal' VALUE o.order_total,
                   'sales_rep_id' VALUE o.sales_rep_id)
FROM orders o ;
```

JSON_OBJECT('id' VALUE o.order_id, 'orderTotal' VALUE o.order_total, 'sales_rep_id' VALUE o.sales_rep_id)
1 {"id":2458,"orderTotal":78279.6,"sales_rep_id":153}
2 {"id":2397,"orderTotal":42283.2,"sales_rep_id":154}
3 {"id":2454,"orderTotal":6653.4,"sales_rep_id":154}
4 {"id":2354,"orderTotal":46257,"sales_rep_id":155}
5 {"id":2358,"orderTotal":7826,"sales_rep_id":155}



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

The `JSON_OBJECT` function returns a JSON object where data is stored in name-value pairs. The query creates a JSON object with three properties – `id`, `order_total` and `sales_rep_id`.

The number of JSON objects created in this query is 50. There is a JSON object created for each row in the `orders` table.

## JSON\_ARRAY Function

- JSON\_ARRAY function generate a JSON array based on the parameters passed.

```
SELECT JSON_ARRAY(customer_id, sales_rep_id, order_total)
FROM orders o ;
```

Query Result	
	SQL
Fetched 50 rows in 0.005 seconds	
1	JSON_ARRAY(CUSTOMER_ID,SALES_REP_ID,ORDER_TOTAL)
2	[101,153,78279.6]
3	[102,154,42283.2]
4	[103,154,6653.4]
5	[104,155,46257]
5	[105,155,7826]



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

JSON\_ARRAY function accepts the array elements are parameters and returns a JSON array. The query shown in the slide has JSON\_ARRAY function accepting three parameters – customer\_id, sales\_rep\_id and order\_total. Therefore a JSON array is created for each row in the orders table.

Array elements are enclosed in square brackets [] whereas JSON objects are enclosed in braces {}. The query here returns 50 JSON arrays based on the number of rows in the orders table.

## JSON\_OBJECTAGG Function

- JSON\_OBJECTAGG constructs a JSON object by aggregating data from multiple rows in the table.

```
SELECT JSON_OBJECTAGG(product_name VALUE product_status)
FROM product_information where min_price>2000 ;
```

The screenshot shows the Oracle SQL Developer interface. The 'Query Result' tab is active, displaying the output of the query. The output is a single row of JSON data:

```
1 {"Laptop 128/12/56/v90/110": "orderable", "Laptop 64/10/56/220": "orderable", "Desk - W/48/R": "orderable", "Desk - OS/O/F": "orderable"}
```

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

JSON\_OBJECTAGG creates a JSON object by aggregating data JSON objects from multiple rows in a table.

The query in slide is creating an object from multiple rows in the `product_information` table. You know that a JSON object is a set of name-value pairs. Note that in the query the name field of the object is also populated from one of the column name. The name field is always expected to be a character string. Make sure that you provide a column with character string data for name. Here we use the `product_name` field of the table for the name.

The value field can be a string, number or `null`. In the query here, we provide `product_status` for value.

The resulting JSON object here has four name value pairs as you see in the Query Result tab. These four products satisfy the condition `min_price>2000` in the `product_information` table.

## JSON\_ARRAYAGG Function

- JSON\_ARRAYAGG function constructs a JSON array from multiple rows in the table.

```
SELECT JSON_ARRAYAGG(order_total)
FROM orders
WHERE sales_rep_id = 161 ;
```

The screenshot shows the Oracle SQL Developer interface with a 'Query Result' window. The window title is 'Query Result'. Below it, there are tabs for 'SQL', 'Script', and 'Table'. The status bar indicates 'All Rows Fetched: 1 in 0.003 seconds'. The results pane displays a single row under the heading 'JSON\_ARRAYAGG(ORDER\_TOTAL)'. The value is a JSON array containing 12 numerical elements: [17848.2, 34930, 2854.2, 268651.8, 6394.8, 103679.3, 33893.6, 26632, 23431.9, 25270.3, 69286.4, 48552, 310].

ORACLE

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

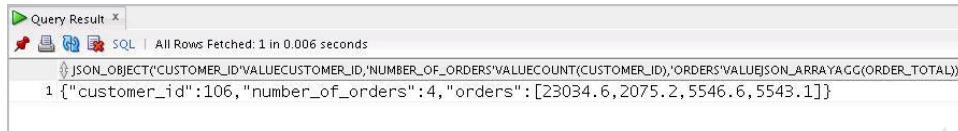
JSON\_ARRAYAGG function creates a JSON array by aggregating data from multiple rows in a table. The query is identifying the order values of certain sales representative with sales\_rep\_id value equal to 161.

A JSON array is created with 12 order\_total values put together.

## SQL/JSON Functions

You can nest multiple SQL/JSON functions to create complicated JSON objects.

```
SELECT JSON_OBJECT('customer_id' VALUE customer_id,
                   'number_of_orders' VALUE count(customer_id),
                   'orders' VALUE JSON_ARRAYAGG(order_total))
  FROM orders
 WHERE customer_id = 106 group by customer_id;
```



The screenshot shows the 'Query Result' window of Oracle SQL Developer. The title bar says 'Query Result'. Below it, there are tabs for 'SQL', 'Script', and 'Table'. A status message says 'All Rows Fetched: 1 in 0.006 seconds'. The results pane displays a single row of JSON data:

```
1 {"customer_id":106,"number_of_orders":4,"orders": [23034.6,2075.2,5546.6,5543.1]}
```

ORACLE

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

## Lesson Agenda

- JSON Data
- JSON data in Oracle Database
- SQL/JSON generation functions
- SQL data from JSON objects
- PL/SQL object types



ORACLE®

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

## Retrieving SQL Data from JSON Object

- Simple dot notation access
- SQL/JSON path expression
- SQL/JSON query functions



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

You can access JSON data in a table column in multiple ways.

Simple dot notation is the simplest and the straight forward method of accessing JSON data in table column.

SQL/JSON path expressions are used to access JSON data by providing the access path to it. You can provide an absolute path expression or a relative path expression.

These techniques can be used when the volume of JSON data is minimal, to handle huge volumes of JSON data, you may have to write PL/SQL blocks which can retrieve multiple values of JSON data according to your application logic.

You can use SQL/JSON query functions such as `JSON_VALUE` and `JSON_QUERY` in such cases.

## Accessing JSON Data

- You can access certain value in the JSON object using simple dot notation.
- To access the order\_id value in the JSON column order\_history in the customer\_followup table, you can write a simple SQL query

```
SELECT cf.order_history.id FROM customer_followup  
WHERE customer_id = 120;
```

Query Result	
	SQL
1 2373	All Rows Fetched: 1 in 0.006 seconds



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

## Accessing JSON Data

- You can define a path expression to query JSON data in the database.
- Each path expression can select zero or more values that match and satisfy the expression.
- Path expressions can use wildcards and matching is case-sensitive.
- We pass the path expressions as an argument to the SQL/JSON function.



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

When JSON data is stored in the database you can query it using path expressions that are somewhat analogous to XQuery or XPath expressions for XML data. Similar to the way that SQL/XML allows SQL access to XML data using XQuery expressions, Oracle Database provides SQL access to JSON data using SQL/JSON path expressions.

An absolute simple path expression begins with a dollar sign (\$), which represents the path-expression context item, that is, the JSON data to be matched.

## JSON\_VALUE Function

- `JSON_VALUE` function is used to retrieve data from an object in a JSON column.
- It returns a scalar SQL value.
- The default return value is `VARCHAR2` value.
- You can change it using a `RETURNING` clause.
- These functions are useful when you work on JSON data in PL/SQL blocks.

```
SELECT JSON_VALUE(order_history,'$.orderTotal')
FROM customer_followup
WHERE customer_id =120;
```

Query Result	
SQL	All Rows Fetched: 1 in 0.002 seconds
JSON_VALUE(ORDER_HISTORY,'\$.ORDERTOTAL')	1 416



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

Similar to `JSON_VALUE`, you have `JSON_QUERY` which selects one or more values from JSON data and returns a string (`VARCHAR2`) value.

You can use different clauses with the `JSON_VALUE` or `JSON_QUERY` functions such as `RETURNING`, `Wrapper`, `Error` and `Empty-Field` clause.

For more information on JSON Data in Oracle Database, you can refer

<https://docs.oracle.com/database/122/ADJSN/toc.htm>

## Using SQL/JSON Functions

```
CREATE OR REPLACE FUNCTION get_order_value(ord_his VARCHAR2)
RETURN NUMBER AS
BEGIN
    RETURN JSON_VALUE(ord_his,'$.orderTotal' RETURNING NUMBER);
END;
```

Script Output X  
Task completed in 0.017 seconds  
Function GET\_ORDER\_VALUE compiled

```
DECLARE
    jsonData VARCHAR2(4000);
    ord_val  NUMBER;
BEGIN
    SELECT order_history INTO jsonData FROM customer_followup
    WHERE customer_id = 120;
    ord_val := get_order_value(jsonData);
    DBMS_OUTPUT.PUT_LINE('The old order value is'||ord_val);
    ord_val := ord_val + (0.1*ord_val);
    DBMS_OUTPUT.PUT_LINE('The new order value is'||ord_val);
END;
```

Script Output X  
Task completed in 0.004 seconds  
PL/SQL procedure successfully completed.  
The old order value was: 416  
The new order value will be: 457.6

ORACLE

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

## Lesson Agenda

- JSON Data
- JSON data in Oracle Database
- SQL/JSON generation functions
- JSON Data in PL/SQL blocks
- PL/SQL object types



ORACLE®

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

## PL/SQL Objects for JSON

- PL/SQL object types allow fine-grained programmatic construction and manipulation of in-memory JSON data.
- PL/SQL object types are transient.
- To persist the data in PL/SQL objects, you must serialize them into VARCHAR2 or LOB and store it in database table.



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

Now that you have learnt how to generate JSON data from relational data. We will further learn how we can accept JSON data and work on it in PL/SQL.

## JSON Object Types in PL/SQL

Following are the object types in PL/SQL, you can use them to work on JSON in PL/SQL program units:

- `JSON_ELEMENT_T`
- `JSON_OBJECT_T`
- `JSON_ARRAY_T`
- `JSON_SCALAR_T`
- `JSON_KEY_LIST`



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

JSON Object types are transient and are used to manipulate JSON data in-memory. You can perform following operations on JSON data:

- Check the structure, types or values of existing JSON data. For example, you can check whether the `minimum_price` of a JSON object product.
- Transform existing JSON data.

The type `JSON_ELEMENT_T` is the super type of all other JSON object types. You can construct an object of this type only by parsing JSON data. All other object types are subtypes of `JSON_ELEMENT_T` type.

`JSON_OBJECT_T` and `JSON_ARRAY_T` are used for JSON objects and arrays. `JSON_SCALAR_T` is used for JSON scalar values such as strings, numbers, boolean values and `null`.

`JSON_KEY_LIST` is a varray of size `VARCHAR2(4000)`. This object is primarily used as a return type of `get_keys` method. This holds the names of all the properties of a JSON object.

## JSON Object Methods

Following categories methods enable you to perform operations on JSON data

- Parsing and Serializing methods
- Getter and Setter methods
- Introspection methods



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

Parsing is the process of converting a VARCHAR2, CLOB or BLOB object into JSON object. Static function `parse()` is used to return an instance of `JSON_ELEMENT_T`, `JSON_OBJECT_T` or `JSON_ARRAY_T`.

Serializing is the process of converting PL/SQL object into textual equivalent. `to_string()` is a serialization method which returns the VARCHAR2 representation of a JSON object. Serialization of the JSON object is required for storing the JSON object in the database.

`JSON_OBJECT_T` and `JSON_OBJECT_ARRAY_T` have getter and setter methods which obtain and update the values of an object.

Type `JSON_ELEMENT_T` has introspection methods which can determine whether an instance is a JSON object, array, scalar, string, number or Boolean. The names of these methods begin with prefix `'is_'`.

`JSON_OBJECT_T` and `JSON_ARRAY_T` also have a `remove()` and `clone()` methods which are used for deleting a JSON object or creating a copy of the JSON object.

## Getter and Setter Methods

- Getter and Setter methods are used with PL/SQL object types `JSON_OBJECT_T` and `JSON_ARRAY_T`
- They are used to retrieve and update objects.

Getter methods

```
get()
get_string()
get_clob()
get_blob()
```

Setter methods

```
put()
put_null()
append() (for
JSON_ARRAY_T only)
```



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

### Getter and Setter methods

These methods are used to retrieve and update JSON data respectively. Getter and Setter methods are used with `JSON_OBJECT_T` and `JSON_ARRAY_T` object types.

The `get()` method returns a reference to the original object as an instance of `JSON_ELEMENT_T`. The modifications you make to the `JSON_ELEMENT_T` instance apply to the original object.

When you pass a JSON object as an argument to `get_string()` method, it returns a `VARCHAR2` equivalent of the JSON object. Similarly `get_clob()` and `get_blob()` methods return a `CLOB` and `BLOB` equivalent respectively.

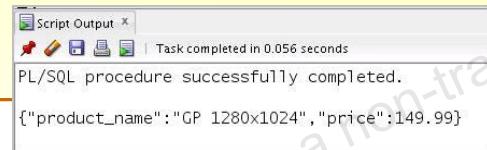
The setter methods are `put()`, `put_null()` and `append()`. These methods update the object or array instance to which they are applied.

Method `put()` requires a second argument (besides the object field name or array element position), which is the new value to set. For an array, `put()` also accepts an optional third argument, `OVERWRITE`. This is a `BOOLEAN` value (default `FALSE`) that says whether to replace an existing value at the given position.

- If the object already has a field of the same name then `put()` replaces that value with the new value.
- If the array already has an element at the given position then, by default, `put()` shifts that element and any successive elements forward (incrementing their positions by one) to make room for the new element, which is placed at the given position. But if optional argument `OVERWRITE` is present and is `TRUE`, then the existing element at the given position is simply replaced by the new element.

## JSON Methods in PL/SQL Example

```
CREATE OR REPLACE PROCEDURE JSON_PUT_METHOD AS
DECLARE
je JSON_ELEMENT_T;
jo JSON_OBJECT_T;
BEGIN
je := JSON_ELEMENT_T.parse('{"product_name": "GP 1280x1024"}');
IF(je.is_Object) THEN
  jo := treat(je AS JSON_OBJECT_T);
  jo.put('price', 149.99);
END IF;
DBMS_OUTPUT.PUT_LINE(je.to_string);
END JSON_PUT_METHOD;
```



ORACLE

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

The code in the slide shows the usage of JSON methods in PL/SQL JSON object types.

You are creating an object of `JSON_ELEMENT_T` and `JSON_OBJECT_T` type.

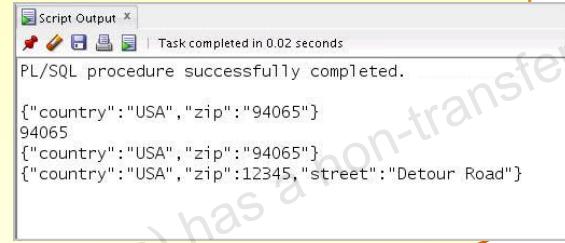
You use `parse()` method to create a `JSON_ELEMENT_T` instance from the string which is passed as a parameter to this method and then type cast the instance into a `JSON_OBJECT_T` type using `treat()` method.

You use `put()` method to update the JSON object. In the example, you are adding the price field to the JSON object.

You can display the final JSON object using `je.to_string` method.

## JSON Methods in PL/SQL Example

```
CREATE OR REPLACE PROCEDURE JSON_GET_METHODS AS
in_data JSON_OBJECT_T;
address JSON_OBJECT_T;
zip NUMBER;
BEGIN
in_data := new JSON_OBJECT_T('{"first_name" : "John", "last_name" : "Doe",
                             "address"   : {"country" : "USA", "zip" : "94065"} }');
address := in_data.get_object('address');
DBMS_OUTPUT.PUT_LINE(address.to_string);
zip := address.get_number('zip');
DBMS_OUTPUT.PUT_LINE(zip);
DBMS_OUTPUT.PUT_LINE(address.to_string);
address.PUT('zip', 12345);
address.PUT('street', 'Detour Road');
DBMS_OUTPUT.PUT_LINE(address.to_string);
END JSON_GET_METHODS;
/
```



The screenshot shows the Oracle SQL Developer interface with a 'Script Output' window. The window title is 'Script Output X'. It displays the message 'Task completed in 0.02 seconds' and 'PL/SQL procedure successfully completed.' Below this, three lines of JSON output are shown:  
{"country": "USA", "zip": "94065"}  
94065  
{"country": "USA", "zip": "94065"}  
{"country": "USA", "zip": 12345, "street": "Detour Road"}



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

The example in the slide has nested JSON objects. The JSON object has three fields, `first_name`, `last_name` and `address` which in turn is a JSON object.

You have initialized the JSON object through a constructor. The `address` field is also a JSON object which in turn has `country` and `zip` fields.

The code example uses `get_number` and `put` methods to access and update the `address` object.

## Summary

In this lesson you should've learnt how to:

- Create a table with a JSON column in it.
- Insert data into a JSON column.
- Generate JSON data from the database tables.
- Use PL/SQL JSON object types.
- Use JSON methods in PL/SQL blocks.



ORACLE

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

## Practice 6: Overview

- 6-1: JSON data in tables
  - Create a table with JSON column.
  - Insert data into the column.
  - Generate JSON data from relational data.
- 6-2: JSON data in PL/SQL blocks
  - Usage of various methods on JSON objects in a PL/SQL block.



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

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

PELISSIER LOIC (loic.pelissier@gmail.com) has a non-transferable  
license to use this Student Guide.

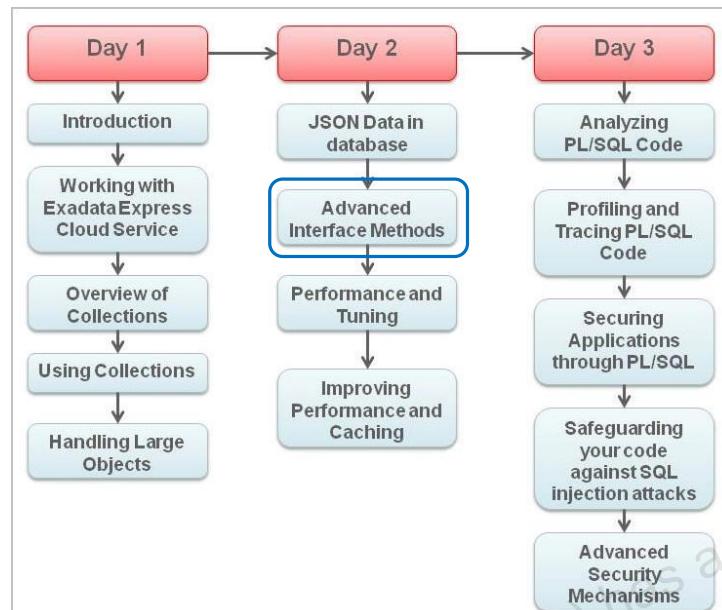
# 7

## Using Advanced Interface Methods

ORACLE®

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

## Course Agenda



ORACLE

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



In this lesson, you will learn to interface PL/SQL programs with other language programs such as C and Java.

## Objectives

After completing this lesson, you should be able to :

- Understand the architecture of execution of external procedures.
- Execute External C programs from PL/SQL
- Execute Java programs from PL/SQL



ORACLE®

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

In this lesson, you learn how to implement an external C routine from PL/SQL code and how to incorporate Java code into your PL/SQL programs.

## Lesson Agenda

- Understanding External Procedures
- Defining an external C procedure
- Executing Java programs from PL/SQL



ORACLE®

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

## PL/SQL External Procedures

- An external procedure is a procedure written in a different programming language
- You register the procedure with base language and then call it for special purpose processing.
- Interface the database server with external systems.
- Extends the functionality of the database server.



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

There might be situations where you might want to implement your application in multiple languages. For example: As Java runs within the address space of the server, you might want to import existing Java applications into the database and leverage this technology by calling Java functions from PL/SQL and SQL.

The strengths and capabilities of different languages are available to you through external procedures, regardless of your program environment. You are not limited to one language. External procedures promote reusability and modularity, because you can deploy specific languages for specific purposes.

An external procedure is stored in a dynamic link library(DLL) which is prototyped in a call specification using PL/SQL. Whenever you invoke a PL/SQL procedure, the language compiler loads the target library at runtime and executes the external procedure in the Oracle Database.

## Oracle Database with Different Languages

Oracle Database allows programmers to work with different programming languages:

- C, C++
- Java
- COBOL
- Visual Basic
- .NET



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

Oracle database allows you work with different programming languages:

- You can use Oracle Call Interface or Oracle C++ Call Interface or Pro\*C/++ compiler to work with C or C++ languages.
- To work with COBOL you can use Pro\*COBOL compiler.
- To work with Java programs you can use JDBC and SQLJ client-side API or use Java in the database.
- You can work with Visual Basic using Oracle Provider for OLE DB.
- You can work with .NET through Oracle Provider for .NET.

Each of these languages offers different advantages such as ease of use, need for portability, availability of programmers with specific expertise. The choice of the language depends on factors like:

- Computation intensive tasks are executed efficiently in a lower level language such as C.
- For security and portability you might choose Java.
- For familiarity with Microsoft programming languages you might choose .NET.

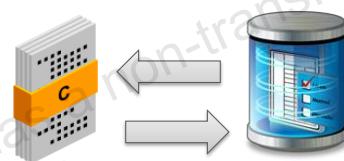
Therefore it is essential for these different programming languages to work together.

## Scenario

A company has very complicated statistics programs written in C. The customer wants to access the data stored in an Oracle database and pass the data into the C programs. After the execution of the C programs, depending on the result of the evaluations, data is inserted into the appropriate Oracle database tables.



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



The scenario in the slide indicates a situation in which the C program is more efficient in processing the company data and generate insights based on it. Instead of implementing the C programs again through database instructions it will be efficient if you can reuse the existing C programs.

The benefits of external procedures:

- External procedures integrate the strength and capability of different languages to give transparent access to these routines within the database.
- They provide functionality in the database that is specific to a particular application, company, or technological area.
- They can be shared by all users on a database, and they can be moved to other databases or computers, thereby providing standard functionality with limited cost in development, maintenance, and deployment.

Using an external procedure, you can invoke an external routine through PL/SQL. By using external procedures, you can integrate the powerful programming features of 3GLs with the ease of data access of SQL and PL/SQL commands.

You can extend the database and provide backward compatibility. For example, you can invoke different index or sorting mechanisms as an external procedure to implement data cartridges.

## Lesson Agenda

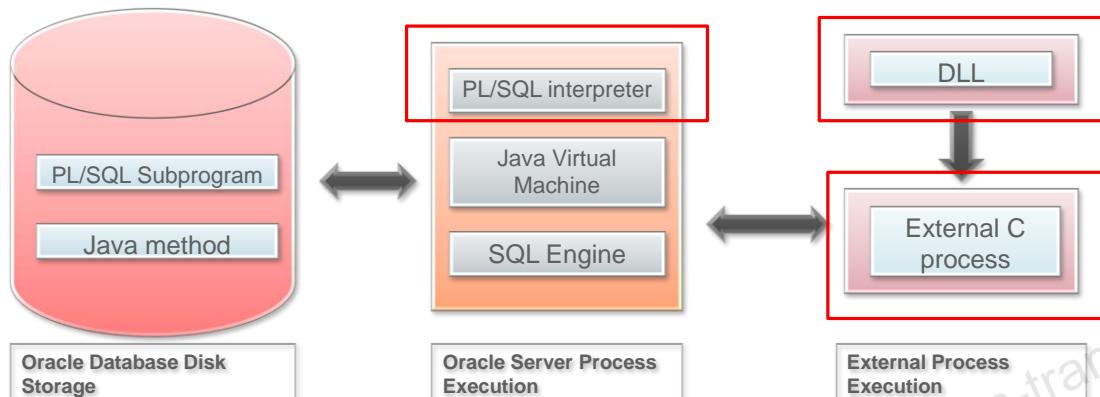
- Understanding External Procedures
- Defining an external C procedure
- Executing Java programs from PL/SQL



ORACLE®

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

## External Procedure Execution Architecture



ORACLE®

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

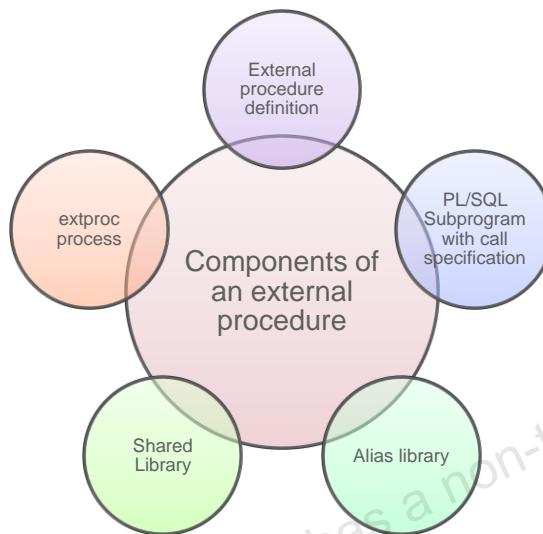
In this lesson we will look at two variants of external procedures – C external procedure and Java external procedure.

A C external procedure is stored in a DLL(Dynamic Linked Library) which is invoked. The PL/SQL interpreter while executing a PL/SQL block invokes the external procedure based on the call specification defined in the PL/SQL block.

The components highlighted are active while executing an external C procedure.

## Components for External C Procedure Execution

Various components work together to interface a C program with Oracle Database.



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

- **External procedure:** A unit of code written in C.
- **Shared library:** An operating system file that stores the external procedure, this is usually the Dynamic Link Library(DLL).
- **Alias library:** A schema object that represents the operating system shared library. You create this object to refer the actual DLL in which the external procedure is present.
- **PL/SQL subprograms:** Packages, procedures, or functions that define the program unit specification and mapping to the PL/SQL library.
- **extproc process:** A session-specific process that executes external procedures. It's a process started by the Oracle Database or database listener for each session. It passes the parameters and other relevant information to the external procedure and receives the results from the external procedure and returns it to the PL/SQL unit. This process is started by the listener process of the database instance.

You can learn more about extproc process here:

<http://docs.oracle.com/database/122/DBSEG/managing-security-for-application-developers.htm#DBSEG656>

## Defining an External C Procedure

Following are the steps involved in defining an external procedure:

1. Define the procedure in external language(C, Java etc).
2. Link the external procedure in a shared library such as Dynamic Link Library(DLL).
3. Create an alias library as a database object.
4. Grant execute privileges on the library.
5. Publish the C library procedure by creating a call specification in the PL/SQL program unit.
6. You can now execute the external procedure.



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

## Define a C Function

1. Create a C program, you generally save it as a .c file.

```
#include <ctype.h>
int calc_tax(int n)
{
    int tax;
    tax = (n*8)/100;
    return(tax);
}
```

2. Link the external procedure by creating a shared object(.so) file for the .c file. And place it in \$ORACLE\_HOME/bin.

**Note:** These steps vary for each operating system; consult the documentation.



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

The slide demonstrates the process of creating a C program and then converting it into a shared object file.

You can use different methods for generating the .so file. The creation of the .so file is dependent on the underlying operating system you use.

The C program is a simple tax calculation program.

## Creating an Alias Library

3. Use the CREATE LIBRARY statement to create an alias library object.

```
CREATE OR REPLACE LIBRARY library_name IS|AS  
'file_path';
```

4. Grant the EXECUTE privilege on the alias library.

```
GRANT EXECUTE ON library_name TO user|ROLE|PUBLIC;
```



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

An alias library is a database object that is used to map to an external shared library. An external procedure that you want to use must be stored in a DLL or a shared object library (SO) operating system file. The DBA controls access to the DLL or .so files by using the CREATE LIBRARY statement to create a schema object called an alias library that represents the external file. The DBA must give you EXECUTE privileges on the library object so that you can publish the external procedure, and then call it from a PL/SQL program.

### Steps

- 1., 2. Steps 1 and 2 vary for each operating system. Consult your operating system or the compiler documentation.
3. Create an alias library object by using the CREATE LIBRARY command:

```
CREATE OR REPLACE LIBRARY c_utility  
AS $ORACLE_HOME/bin/calc_tax.so';  
/
```

The example shows the creation of a database object called `c_utility`, which references the location of the file and the name of the operating system file, `calc_tax.so`.

4. Grant the EXECUTE privilege on the library object:

```
GRANT EXECUTE ON c_utility TO OE;
```

5. Publish the external C routine.
6. Call the external C routine from PL/SQL.

### **Dictionary Information**

The alias library definitions are stored in the `USER_LIBRARIES` and `ALL_LIBRARIES` data dictionary views.

## Publishing External C Procedures

Publish the external procedure in PL/SQL through call specifications. A call specification performs the following tasks:

- Registers the external procedure.
- Dispatches external procedure call to the right target.
- Performs Data type conversions.
- Performs parameter mode mappings.
- Automatic memory allocation and clean up.

Access to the external procedure is controlled through the alias library.



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

Oracle Database can use only external procedures that are published through a call specification, which maps names, parameter types, and return types for your Java class method or C external procedure to their SQL counterparts. It is written like any other PL/SQL stored procedure except that, in its body, instead of declarations and a `BEGIN END` block, you code the `AS LANGUAGE` clause.

Call specifications can be specified in any of the following locations:

- Stand-alone PL/SQL procedures and functions
- PL/SQL package specifications
- PL/SQL package bodies
- Object type specifications
- Object type bodies

## Call Specification Syntax

- Identify the external body within a PL/SQL program to publish the external C procedure.

```
CREATE OR REPLACE FUNCTION function_name
(parameter_list)
RETURN datatype
    regularbody | externalbody
END;
```

- The external body contains the external C procedure information.

```
IS|AS LANGUAGE C
LIBRARY libname
[NAME C_function_name]
[CALLING STANDARD C | PASCAL]
[WITH CONTEXT]
[PARAMETERS (param_1, [param_n]);]
```



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

You create the PL/SQL procedure or function and use the IS|AS LANGUAGE C to publish the external C procedure. The external body contains the external routine information.

### Syntax Definitions

where: LANGUAGE

Is the language in which the external routine was written (defaults to C)

LIBRARY libname

Is the name of the library database object

NAME "C\_function\_name"

Represents the name of the C function; if omitted, the external procedure name must match the name of the PL/SQL subprogram

CALLING STANDARD

Specifies the Windows NT calling standard (C or Pascal) under which the external routine was compiled (defaults to C)

WITH CONTEXT

Specifies that a context pointer is passed to the external routine for callbacks

parameters

Identifies arguments passed to the external routine

## Call Specification

- The parameter list:

```
parameter_list_element
[ , parameter_list_element ]
```

- The parameter list element:

```
{ formal_parameter_name [indicator]
| RETURN INDICATOR
| CONTEXT }
[BY REFERENCE]
[external_datatype]
```



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

The formal parameter list can be used to specify the position and the types of arguments, as well as to indicate whether they should be passed by value or by reference.

### Syntax Definitions

where:	formal_parameter_name [INDICATOR]	Is the name of the PL/SQL parameter that is being passed to the external routine; the INDICATOR keyword is used to map a C parameter whose value indicates whether the PL/SQL parameter is null
	RETURN INDICATOR	Corresponds to the C parameter that returns a null indicator for the function return value
	CONTEXT	Specifies that a context pointer will be passed to the external routine
	BY REFERENCE	In C, you can pass IN scalar parameters by value (the value is passed) or by reference (a pointer to the value is passed). Use BY REFERENCE to pass the parameter by reference.
	External_datatype	Is the external data type that maps to a C data type

**Note:** The PARAMETER clause is optional if the mapping of the parameters is done on a positional basis, and indicators, reference, and context are not needed.

## Publishing an External C Routine

### Example

- Publish a C function called calc\_tax from a PL/SQL function:

```
CREATE OR REPLACE FUNCTION tax_amt (
  x BINARY_INTEGER)
RETURN BINARY_INTEGER
AS LANGUAGE C
LIBRARY sys.c_utility
NAME "calc_tax";
```

- The C prototype:

```
int calc_tax (n);
```



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

You have an external C function called calc\_tax that takes in one argument, the total sales amount. The function returns the tax amount calculated at 8%. The prototype for your calc\_tax function is as follows:

```
int calc_tax (n);
```

To publish the calc\_tax function in a stored PL/SQL function, use the AS LANGUAGE C clause within the function definition. The NAME identifies the name of the C function. Double quotation marks are used to preserve the case of the function defined in the C program. LIBRARY identifies the library object that locates the C file. The PARAMETERS clause is not needed in this example, because the mapping of parameters is done on a positional basis.

## Executing an External C Procedure

Following are the steps involved in executing an external procedure:

1. The user process invokes a PL/SQL program.
2. The server process executes a PL/SQL subprogram.
3. PL/SQL subprogram looks up the alias library.
4. Oracle Database starts the external procedure agent, extproc.
5. The extproc process loads the shared library.
6. The extproc process links the server to the external file and executes the external procedure.
7. The data and status are returned to the server.



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

```
BEGIN
  DBMS_OUTPUT.PUT_LINE(tax_amt(1000));
END;
-- Additional code for experimenting
DECLARE
  CURSOR cur_orders IS
    SELECT order_id, order_total
      FROM   orders;
  v_tax  NUMBER(8,2);
BEGIN
  FOR order_record IN cur_orders
  LOOP
    v_tax := tax_amt(order_record.order_total);
    DBMS_OUTPUT.PUT_LINE('Total tax: ' || v_tax);
  END LOOP;
END;
```

## Lesson Agenda

- Understanding External Procedures
- Defining an external C procedure
- Executing Java programs from PL/SQL

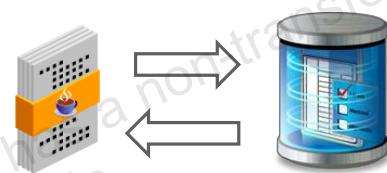


ORACLE®

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

## Executing Java Programs from PL/SQL

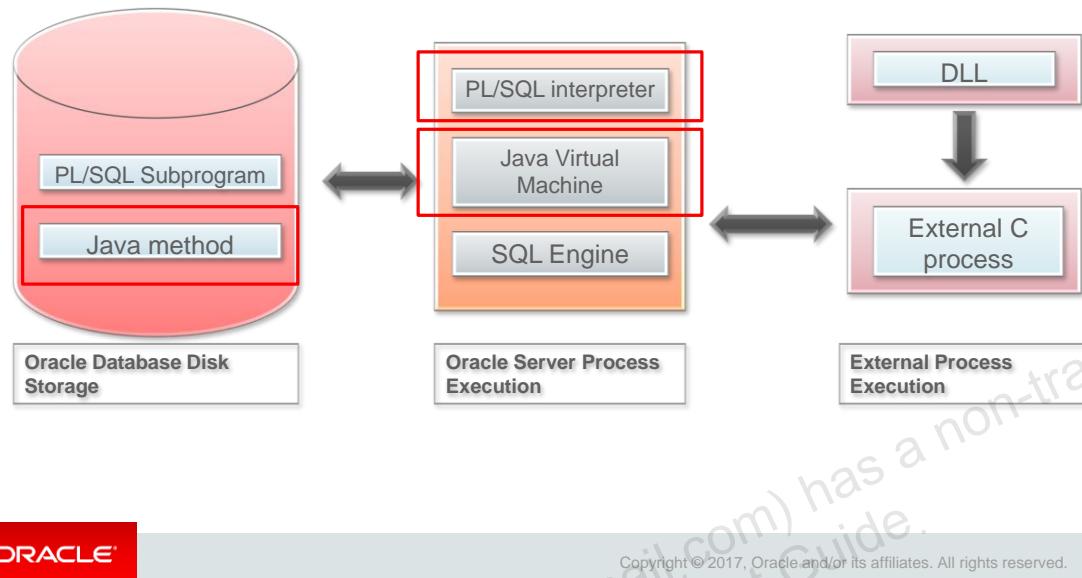
- Java programs can natively execute in Oracle Database like C programs.
- Unlike C programs Java programs are stored as Schema objects.
- Java programs use a Java shared library or libunit for execution.
- Libunits are analogous to DLLs in case of C.
- There is a libunit for each Java class.
- Oracle Database executes these libunits on Java Virtual Machine which resides natively on the in the database.



ORACLE

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

## External Procedure Execution Architecture



Java programs are stored as schema objects, following is the process of executing a java program from a PL/SQL block.

1. Use the `loadjava` utility from command line or from an application to upload the Java binaries and resources into a system-generated database table, where they are stored as Java schema objects.
2. The `loadjava` command-line utility uses the SQL `CREATE JAVA` statements to load Java source, class, or resource files into the RDBMS libunits. Libunits can be considered analogous to the DLLs written in C, although they map one-to-one with Java classes, whereas DLLs can contain multiple routines. Alternatively, you can implicitly call `CREATE JAVA` from SQL\*Plus.
3. When you load a Java class into the database, its methods are not published automatically, because Oracle Database does not know which methods are safe entry points for calls from SQL. To publish the Java class method, create the PL/SQL subprogram unit specification that references the Java class methods.
4. Execute the PL/SQL subprogram that invokes the Java class method.

The components highlighted in the architecture diagram are used while executing a Java program.

## Development Steps for Java Class Methods

1. Upload the Java file using the `loadjava` utility.
2. Publish the Java class method by creating the call specification in PL/SQL program unit.
3. Execute the PL/SQL subprogram that invokes the Java class method.



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

Similar to using external C routines, the following steps are required to complete the setup before executing the Java class method from PL/SQL:

1. Upload the Java file. This takes an external Java binary file and stores the Java code in the database.
2. Publish the Java class method by creating the PL/SQL subprogram unit specification that references the Java class methods.
3. Execute the PL/SQL subprogram that invokes the Java class method.

## Loading Java Class Methods

- Upload the Java file.
  - At the operating system level, use the `loadjava` command-line utility to load either the Java class file or the Java source file.
- To load the Java source file, use:

```
>loadjava -user oe/oe@pdborcl Factorial.java
```



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

Java classes and their methods are stored in RDBMS libunits where the Java sources, binaries, and resources can be loaded.

Use the `loadjava` command-line utility to load and resolve the Java classes. Using the `loadjava` utility, you can upload the Java source, class, or resource files into an Oracle database, where they are stored as Java schema objects. You can run `loadjava` from the command line or from an application.

After the file is loaded, it is visible in the data dictionary views.

```
SELECT object_name, object_type FROM user_objects
WHERE object_type like 'J%';
OBJECT_NAME          OBJECT_TYPE
-----
Factorial           JAVA SOURCE

SELECT text FROM user_source WHERE name = 'Factorial';
TEXT
-----
public class Factorial {
    public static int calcFactorial (int n) {
        if (n == 1) return 1;
        else return n * calcFactorial (n - 1) ;    }}
```

## Publishing a Java Class Method

- Publish the Java class method by creating the call specification in PL/SQL
  - Identify the external body within a PL/SQL program to publish the Java class method.
  - The external body contains the name of the Java class method.
- The call specification can be located in PL/SQL program or PL/SQL package or ADT definition.

Example:

```
CREATE OR REPLACE FUNCTION plstojavafac_fun
(N NUMBER)
RETURN NUMBER
AS
LANGUAGE JAVA
NAME 'Factorial.calcFactorial
(int) return int';
```



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

The publishing of Java class methods is specified in the AS LANGUAGE clause. This call specification identifies the appropriate Java target routine, data type conversions, parameter mode mappings, and purity constraints. You can publish value-returning Java methods as functions and void Java methods as procedures.

You want to publish a Java method named calcFactorial that returns the factorial of its argument, as shown in the slide:

- The PL/SQL function plstojavafac\_fun is created to identify the parameters and the Java characteristics.
- The NAME clause string uniquely identifies the Java method.
- The parameter named N corresponds to the int argument.

You can publish the Java method directly from the JVM.

Java calcFactorial method definition:

```
public class Factorial {
    public static int calcFactorial (int n) {
        if (n == 1) return 1;
        else return n * calcFactorial (n - 1) ;
    }
}
```

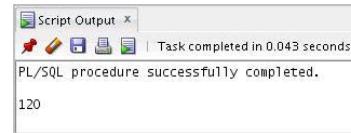
Here is another example:

```
create or replace function dept_sal(p_deptno VARCHAR2)
return VARCHAR2
AS
LANGUAGE JAVA
NAME 'dept.sal(java.lang.String) return java.lang.String';
/
```

## Executing the Java Routine

You can call the `calcFactorial` class method by using the following command:

```
EXECUTE DBMS_OUTPUT.PUT_LINE(plstojavafac_fun(5));
```



Alternatively, to execute a SELECT statement from the DUAL table:

```
SELECT plstojavafac_fun(5) FROM dual;
```



ORACLE

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

## Creating Call Specifications in Packages

You can create call specifications in PL/SQL packages. Here is an example:

```
CREATE OR REPLACE PACKAGE Demo_pack
AUTHID DEFINER
AS
  PROCEDURE plsToJ_InSpec_proc
    (x BINARY_INTEGER, y VARCHAR2, z DATE)
END;
```

```
CREATE OR REPLACE PACKAGE BODY Demo_pack
AS
  PROCEDURE plsToJ_InSpec_proc
    (x BINARY_INTEGER, y VARCHAR2, z DATE)
  IS LANGUAGE JAVA
  NAME 'pkg1.class4.J_InSpec_meth
        (int, java.lang.String, java.sql.Date)';
```



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

The examples in the slide create a package specification and body named `Demo_pack`.

The package declaration has the specification of the PL/SQL procedure named `plsToJ_InSpec_proc`.

You have the procedure definition in the package body.

Note that you cannot tell whether this procedure is implemented by PL/SQL or by way of an external procedure. The details of the implementation appear only in the package body in the declaration of the procedure body.

## Quiz



Which of the following statements is true about extproc?

- a. Oracle Database starts the external procedure agent, extproc.
- b. The extproc process loads the shared library.
- c. The extproc process compiles the external procedure.
- d. All of the above.

ORACLE

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

**Answer: a,b**

## Quiz



Which of the following are true about call specifications:

- a. Link the server to the external file and execute the external procedure
- b. Dispatch the appropriate C or Java target procedure
- c. Perform data type conversions
- d. Perform parameter mode mappings
- e. Perform automatic memory allocation and cleanup
- f. Call Java methods or C procedures from database triggers

ORACLE

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

**Answer: b,c,d,e,f**

## Quiz



Select the correct order of the steps required to execute a Java class method from PL/SQL:

- A. Publish the Java class method by creating the PL/SQL subprogram unit specification that references the Java class methods.
- B. Upload the Java file by using the `loadjava` command-line utility.
- C. Execute the PL/SQL subprogram that invokes the Java class method.
  - a. A, B, C
  - b. B, A, C
  - c. C, A, B
  - d. C, B, A



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



**Answer: b**

## Summary

In this lesson, you should have learned how to use:

- External C routines and call them from your PL/SQL programs
- Java methods and call them from your PL/SQL programs



ORACLE®

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

You can embed calls to external C programs from your PL/SQL programs by publishing the external routines in a PL/SQL block. You can take external Java programs and store them in the database to be called from PL/SQL functions, procedures, and triggers.

## Practice 7: Overview

This practice covers writing programs to interact with:

- C routines
- Java code



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

In this practice, you write two PL/SQL programs: one program calls an external C routine and the other calls a Java routine.

Use the OE schema for this practice.

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

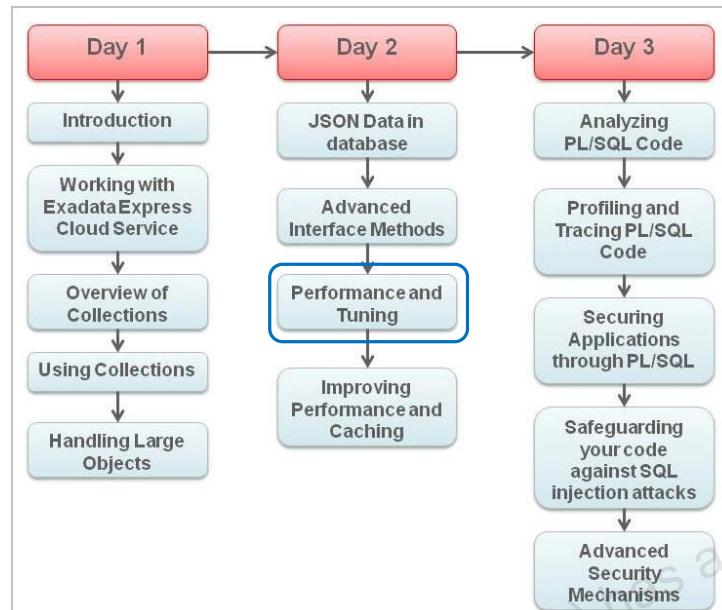
PELISSIER LOIC (loic.pelissier@gmail.com) has a non-transferable  
license to use this Student Guide.

# Performance and Tuning



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

## Course Agenda



ORACLE

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



In this lesson you will learn about various aspects of improving performance of the application. You will learn about different compiler options which can be used for performance tuning.

## Objectives

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

- Configure your compiler to appropriate compilation mode.
- Enable intra unit inlining to improve performance.
- Tune PL/SQL code to improve performance.



ORACLE®

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

In this lesson, you will learn various methods of improving performance of a PL/SQL unit. You can broadly identify two ways of improving the performance.

1. Configure the compiler right.
  2. Write efficient code.
- In the following slides you will look into this in detail.

## Lesson Agenda

- Configuring the compiler
- Enabling intraunit inlining
- Tuning PL/SQL code



ORACLE®

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

## Compiling a PL/SQL Unit

- The performance of a PL/SQL unit can vary based on the compilation method you use.
- There are two modes of compilation:
  - Interpreted compilation
    - Default compilation mode
    - Interpreted at run time
  - Native compilation
    - Compiles into native code
    - Stored in the SYSTEM table space



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

During execution PL/SQL program unit is compiled into machine code — called M-Code. However, the target machine is virtual: PVM — PL/SQL virtual machine. At run time the M-Code is scanned by a subroutine on the PVM. The scanning detects op code and its operands and then calls the subroutine that implements this op code with the appropriate actual arguments. The fetching of the op code and operands happen at the runtime. This is interpreted compilation.

In native mode, a platform-specific DLL is produced at compile time. This DLL at runtime executes on the PVM subroutines with the same arguments as would have been called by scanning the M-Code. The difference is that the scanning effort has been moved from runtime to compile time, this improves runtime performance. This is native compilation. The DLL compiled is stored in SYSTEM table space.

You have to choose appropriate compile mode based on your purpose.

## Deciding on a Compilation Method

- Use the interpreted mode when (typically during development):
  - You use a debugging tool, such as SQL Developer
  - You need the code compiled quickly
- Use the native mode when (typically post development):
  - Your code is heavily PL/SQL based
  - You want increased performance in production



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

When deciding on a compilation method, you need to examine:

- Where you are in the development cycle
- What the program unit does

If you are debugging and recompiling program units frequently, the interpreted mode has the following advantages:

- You can use PL/SQL debugging tools on program units compiled for interpreted mode (but not for those compiled for native mode).
- Compiling for interpreted mode is faster than compiling for native mode.

After completing the debugging phase of development, consider the following when determining whether to compile a PL/SQL program unit for native mode:

- The native mode provides the greatest performance gains for computation-intensive procedural operations. Examples are data warehouse applications and applications with extensive server-side transformations of data for display.
- The native mode provides the least performance gains for PL/SQL subprograms that spend most of their time executing SQL.
- When many program units (typically over 15,000) are compiled for native execution, and are simultaneously active, the large amount of shared memory required might affect system performance.

## Configuring the Compiler

- PLSQL\_CODE\_TYPE: Specifies the compilation mode for the PL/SQL library units

```
PLSQL_CODE_TYPE = { INTERPRETED | NATIVE }
```

- PLSQL\_OPTIMIZE\_LEVEL: Specifies the optimization level to be used to compile the PL/SQL library units

```
PLSQL_OPTIMIZE_LEVEL = { 0 | 1 | 2 | 3 }
```

- In general, for faster performance, use the following setting:

```
PLSQL_CODE_TYPE = NATIVE
PLSQL_OPTIMIZE_LEVEL = 2
```



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

### The PLSQL\_CODE\_TYPE Parameter

The PLSQL\_CODE\_TYPE compilation parameter determines whether the PL/SQL code is natively compiled or interpreted.

If you choose INTERPRETED:

- PL/SQL library units are compiled to PL/SQL bytecode format
- These modules are executed by the PL/SQL interpreter engine

If you choose NATIVE:

- PL/SQL library units (with the possible exception of top-level anonymous PL/SQL blocks) are compiled to native (machine) code
- Such modules are executed natively without incurring interpreter overhead

When the value of PL\_SQL\_CODE\_TYPE parameter is changed, it has no effect on the PL/SQL library units that have already been compiled. The value of this parameter is stored persistently with each library unit. If a PL/SQL library unit is compiled natively, all subsequent automatic recompilations of that library unit use the native compilation. In Oracle Database 12c, native compilation is easier and more integrated, with fewer initialization parameters to set.

### The PLSQL\_OPTIMIZE\_LEVEL Parameter

This parameter specifies the optimization level that is used to compile the PL/SQL library units. The higher the setting of this parameter, the more effort the compiler makes to optimize the PL/SQL library units. The available values are 0, 1, 2, and 3:

- 0: Maintains the evaluation order and, therefore, the pattern of side effects, exceptions, and package initializations of Oracle9*i* and earlier releases. It also removes the new semantic identity of `BINARY_INTEGER` and `PLS_INTEGER`, and restores the earlier rules for the evaluation of integer expressions.
- 1: Applies a wide range of optimizations to PL/SQL programs, including the elimination of unnecessary computations and exceptions, but generally does not move source code out of its original source order.
- 2: Applies a wide range of modern optimization techniques beyond those of level 1, including changes that may move source code relatively far from its original location.
- 3: Is available in Oracle Database 12c. It applies a wide range of optimization techniques beyond those of level 2, automatically including techniques not specifically requested. This enables subprogram inlining, which is an optimization process that replaces procedure calls with a copy of the body of the procedure to be called. The copied procedure almost always runs faster than the original call.

To allow subprogram inlining, either accept the default value of the `PLSQL_OPTIMIZE_LEVEL` initialization parameter (which is 2) or set it to 3. With `PLSQL_OPTIMIZE_LEVEL = 2`, you must specify each subprogram to be inlined. With `PLSQL_OPTIMIZE_LEVEL = 3`, the PL/SQL compiler seeks opportunities to inline subprograms beyond those that you specify.

Generally, setting this parameter to 2 pays off in terms of better execution performance. If the compiler runs slowly on a particular source module or if optimization does not make sense for some reason (for example, during rapid turnaround development), setting this parameter to 1 results in almost as good a compilation with less use of compile-time resources. The value of this parameter is stored persistently with the library unit.

## Viewing the Compilation Settings

**DESCRIBE ALL\_PLSQL\_OBJECT\_SETTINGS**

→ Displays the settings for a PL/SQL object

Name	Null	Type
OWNER	NOT NULL	VARCHAR2(128)
NAME	NOT NULL	VARCHAR2(128)
TYPE		VARCHAR2(12)
PLSQL_OPTIMIZE_LEVEL		NUMBER
PLSQL_CODE_TYPE		VARCHAR2(4000)
PLSQL_DEBUG		VARCHAR2(4000)
PLSQL_WARNINGS		VARCHAR2(4000)
NLS_LENGTH_SEMANTICS		VARCHAR2(4000)
PLSQL_CCFLAGS		VARCHAR2(4000)
PLSCOPE_SETTINGS		VARCHAR2(4000)
ORIGIN_CON_ID		NUMBER



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

The `USER|ALL|DBA_PLSQL_OBJECT_SETTINGS` data dictionary views are used to display the settings for a PL/SQL object.

The columns of the `USER_PLSQL_OBJECTS_SETTINGS` data dictionary view include:

- **Owner:** The owner of the object. This column is not displayed in the `USER_PLSQL_OBJECTS_SETTINGS` view.
- **Name:** The name of the object
- **Type:** The available choices are PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER, TYPE, or TYPE BODY.
- **PLSQL\_OPTIMIZE\_LEVEL:** The optimization level that was used to compile the object
- **PLSQL\_CODE\_TYPE:** The compilation mode for the object
- **PLSQL\_DEBUG:** Specifies whether or not the object was compiled for debugging
- **PLSQL\_WARNINGS:** The compiler warning settings used to compile the object
- **NLS\_LENGTH\_SEMANTICS:** The national language support (NLS) length semantics used to compile the object
- **PLSQL\_CCFLAGS:** The conditional compilation flag used to compile the object
- **PLSCOPE\_SETTINGS:** Controls the compile time collection, cross reference, and storage of PL/SQL source code identifier data

## Viewing the Compilation Settings

```
SELECT name, plsql_code_type, plsql_optimize_level  
FROM user_plsql_object_settings;
```

→ To view the compilation settings

NAME	PLSQL_CODE_TYPE	PLSQL_OPTIMIZE_LEVEL
1 ACTIONS_T	INTERPRETED	2
2 ACTION_T	INTERPRETED	2
3 ACTION_V	INTERPRETED	2
4 ADD_ORDER_ITEMS	INTERPRETED	2
5 ALLOCATE_NEW_PROJ_LIST	INTERPRETED	2
6 CATALOG_TYP	INTERPRETED	2
7 CATALOG_TYP	INTERPRETED	2
8 CATEGORY_TYP	INTERPRETED	2
9 CHANGE_CREDIT	INTERPRETED	2
10 COMPOSITE_CATEGORY_TYP	INTERPRETED	2



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

Set the values of the compiler initialization parameters by using the ALTER SYSTEM or ALTER SESSION statements.

The parameters' values are accessed when the CREATE OR REPLACE or ALTER statements are executed.

## Setting Up a Database for Native Compilation

- You can set up native compilation mode for all the PL/SQL packages executing in the database instance.
- To setup a database for native compilation:
  - You require DBA privileges
  - You have to set `PLSQL_CODE_TYPE` compilation parameter to `NATIVE`

```
SELECT name, plsql_code_type, plsql_optimize_level
FROM   user_plsql_object_settings
WHERE  name = 'ADD_ORDER_ITEMS';
```

NAME	PLSQL_CODE_TYPE	PLSQL_OPTIMIZE_LEVEL
1 ADD_ORDER_ITEMS	INTERPRETED	2

ORACLE

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

If you have DBA privileges, you can set up a new database for PL/SQL native compilation by setting the `PLSQL_CODE_TYPE` compilation parameter to `NATIVE`. The performance benefits apply to all built-in PL/SQL packages that are used for many database operations.

## Modifying Compilation Mode of a Program Unit

The screenshot shows the Oracle SQL Developer interface. At the top, a yellow box contains the command:

```
ALTER SYSTEM SET PLSQL_CODE_TYPE = NATIVE;
```

Below this is a 'Script Output' window showing the message:

```
System SET altered.
```

Further down, another yellow box contains the command:

```
ALTER PROCEDURE add_order_items COMPILE;
```

Below this is a 'Script Output' window showing the message:

```
Procedure ADD_ORDER_ITEMS altered.
```

At the bottom, a yellow box contains the query:

```
SELECT name, plsql_code_type, plsql_optimize_level
FROM user_plsql_object_settings
WHERE name = 'ADD_ORDER_ITEMS';
```

Below this is a 'Query Result' window displaying the results of the query:

NAME	PLSQL_CODE_TYPE	PLSQL_OPTIMIZE_LEVEL
ADD_ORDER_ITEMS	NATIVE	2

The 'PLSQL\_CODE\_TYPE' column for the row 'ADD\_ORDER\_ITEMS' is highlighted with a red box.

ORACLE

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

To change a compiled PL/SQL object from interpreted code type to native code type, you must set the `PLSQL_CODE_TYPE` parameter to `NATIVE` (optionally, set the other parameters), and then recompile the program.

In the example in the slide:

1. The compilation type is checked on the `ADD_ORDER_ITEMS` program unit.
2. The compilation method is set to `NATIVE` at the session level.
3. The `ADD_ORDER_ITEMS` program unit is recompiled.
4. The compilation type is checked again on the `ADD_ORDER_ITEMS` program unit to verify that it changed.

If you want to compile an entire database for native or interpreted compilation, scripts are provided to help you do so.

- You must have DBA privileges.
- Set `PLSQL_CODE_TYPE` at the system level.
- Run the `dbmsupgnv.sql`-supplied script that is found in the `\Oraclehome\product\12.1.0\db_1\RDBMS\ADMIN` folder.

For detailed information, see the *Oracle Database PL/SQL Language Reference 12c* reference manual.

## Lesson Agenda

- Configuring the compiler
- Enabling Subprogram inlining
- Tuning PL/SQL code



ORACLE®

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

## PL/SQL Optimizer

- PL/SQL optimizer rearranges the code for better performance.
- Enabled by default.
- `PLSQL_OPTIMIZE_LEVEL` compilation parameter determines the extent of optimization.
- Subprogram inlining is one of the optimization techniques implemented by the optimizer.



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

## Subprogram Inlining: Introduction

- Definition:
  - Inlining is the replacement of a call to a subroutine with a copy of the body of the subroutine that is called.
  - The copied procedure generally runs faster than the original.
  - The PL/SQL compiler can automatically find the calls that should be inlined.
- Benefits:
  - When applied judiciously, inlining can provide large performance gains (by a factor of 2–10).



ORACLE

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

Subprogram inlining replaces a subprogram invocation with a copy of the invoked subprogram (if the invoked and invoking subprograms are in the same program unit). To allow subprogram inlining, either accept the default value of the `PLSQL_OPTIMIZE_LEVEL` compilation parameter (which is 2) or set it to 3.

The copied procedure almost always runs faster than the original call, because:

- The need to create and initialize the stack frame for the called procedure is eliminated
- The optimization can be applied over the combined text of the call context and the copied procedure body
- Propagation of constant actual arguments often causes the copied body to collapse under optimization

When inlining is achieved, you can see performance gains of 2–10 times.

With the recent releases, the PL/SQL compiler can automatically find calls that should be inlined, and can do the inlining correctly and quickly. There are some controls to specify where and when the compiler should do this work (using the `PLSQL_OPTIMIZATION_LEVEL` database parameter), but usually a general request is sufficient.

## Using Inlining

- Implement inlining via two methods:
  - Oracle parameter PLSQL\_OPTIMIZE\_LEVEL
  - PRAGMA INLINE
- It is recommended that you inline:
  - Small programs
  - Programs that are frequently executed



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

When implementing inlining, it is recommended that the process be applied to smaller programs, and programs that execute frequently. For example, you may want to inline small helper programs.

There are two ways to use inlining:

1. Set the PLSQL\_OPTIMIZE\_LEVEL parameter to 3. When this parameter is set to 3, the PL/SQL compiler searches for calls that might profit from inlining and inlines the most profitable calls. Profitability is measured by those calls that help the program speed up the most and keep the compiled object program as short as possible.

```
ALTER SESSION SET plsql_optimize_level = 3;
```
2. Use PRAGMA INLINE in your PL/SQL code. This identifies whether a specific call should be inlined. Setting this pragma to “YES” has an effect only if the optimize level is set to two or higher.

## Inlining Concepts

Noninlined program:

```
CREATE OR REPLACE PROCEDURE small_pgm
IS
  a NUMBER;
  b NUMBER;

  PROCEDURE touch(x IN OUT NUMBER, y NUMBER)
  IS
  BEGIN
    IF y > 0 THEN
      x := x+1;
    END IF;
  END;

  BEGIN
    a := b;
    FOR i IN 1..10 LOOP
      touch(a, -17);
      a := a*b;
    END LOOP;
  END small_pgm;
```

Without inlining this block of code would execute 10 times, irrespective of the value of 'y' passed to touch function



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

The example shown in the slide will be expanded to show you how a procedure is inlined.

The `a := a * b` assignment at the end of the loop looks like it could be moved before the loop; however, it cannot, because `a` is passed as an `IN OUT` parameter to the `TOUCH` procedure. The compiler cannot be certain what the procedure does to its parameters. This results in the multiplication and in the assignment being completed 10 times instead of only once, even though multiple executions are not necessary.

## Inlining Concepts

Examine the loop after inlining:

```
...
BEGIN
  a := b;
  FOR i IN 1..10 LOOP
    IF -17 > 0 THEN
      a := a+1;
    END IF;
    a := a*b;
  END LOOP;
END small_pgm;
...
```



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

The code in the slide shows what happens to the loop after inlining.

## Inlining Concepts

The code in the loop is transformed as follows:

```
a := b;  
FOR i IN 1..10 LOOP  
...  
IF false THEN  
a := a+1;  
END IF;  
a := a*b;  
END LOOP;
```

```
a := b;  
FOR i IN 1..10 LOOP  
...  
a := a*b;  
END LOOP;
```

```
a := b;  
a := a*b;  
FOR i IN 1..10 LOOP  
...  
END LOOP;
```

```
a := b*b;  
FOR i IN 1..10 LOOP  
...  
END LOOP;
```



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

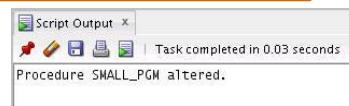
After inlining, procedure internals are visible to the compiler. It can transform the loop in several steps, as shown in the slide.

Instead of 11 assignments (one outside of the loop) and 10 multiplications, only one assignment and one multiplication are performed. If the loop ran a million times (instead of 10), the savings would be a million assignments. For code that contains deep loops that are executed frequently, inlining offers tremendous savings.

## Inlining: How to Enable It ?

- Set the `PLSQL_OPTIMIZE_LEVEL` session-level parameter to a value of 2 or 3:

```
ALTER PROCEDURE small_pgm COMPILE  
PLSQL_OPTIMIZE_LEVEL = 3 REUSE SETTINGS;
```



- Setting it to 2 means no automatic inlining is attempted.
  - Setting it to 3 means automatic inlining is attempted, but no pragmas are necessary.

- Use `PRAGMA INLINE`, with the PL/SQL Subprogram.

ORACLE

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

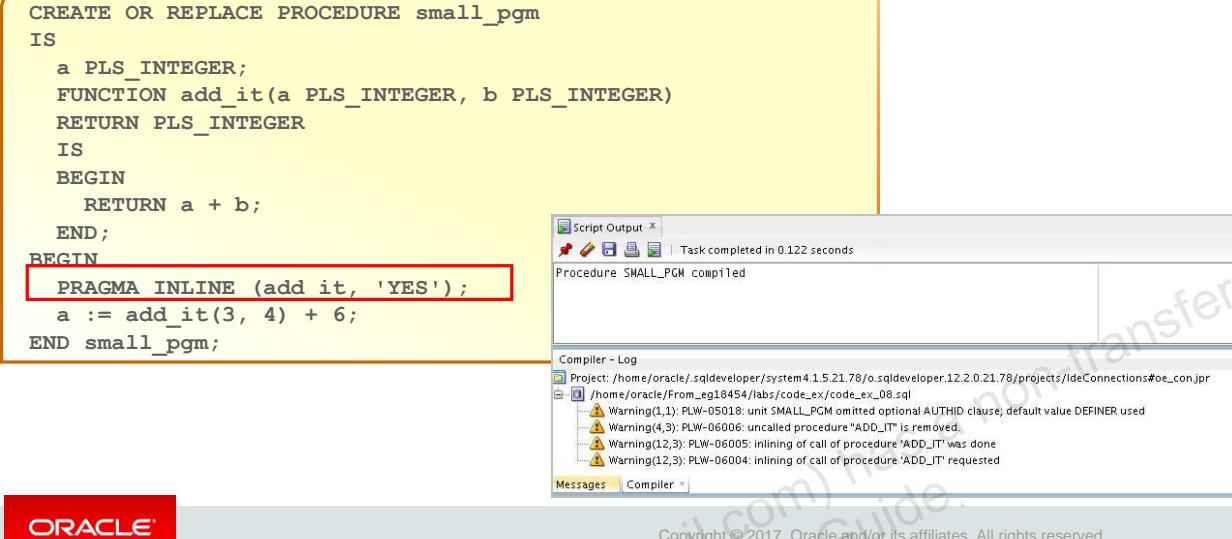
To influence the optimizer to use inlining, you can set the `PLSQL_OPTIMIZE_LEVEL` parameter to a value of 2 or 3. By setting this parameter, you are making a request that inlining be used. It is up to the compiler to analyze the code and determine whether inlining is appropriate. When the optimize level is set to 3, the PL/SQL compiler searches for calls that might profit from inlining and inlines the most profitable calls.

In rare cases, if the overhead of the optimizer makes the compilation of very large applications take too long, you can lower the optimization by setting `PLSQL_OPTIMIZE_LEVEL=1` instead of its default value of 2. In even rarer cases, you might see a change in exception action, either an exception that is not raised at all, or one that is raised earlier than expected. Setting `PLSQL_OPTIMIZE_LEVEL=1` prevents the code from being rearranged.

To enable inlining within a PL/SQL subroutine, you can use `PRAGMA INLINE` to suggest that a specific call be inlined.

## PRAGMA INLINE Example

After setting the `PLSQL_OPTIMIZE_LEVEL` parameter, use a `PRAGMA`:



```

CREATE OR REPLACE PROCEDURE small_pgm
IS
    a PLS_INTEGER;
    FUNCTION add_it(a PLS_INTEGER, b PLS_INTEGER)
    RETURN PLS_INTEGER
    IS
    BEGIN
        RETURN a + b;
    END;
BEGIN
    PRAGMA INLINE (add_it, 'YES');
    a := add_it(3, 4) + 6;
END small_pgm;

```

The screenshot shows the Oracle SQL Developer interface. On the left, the code editor displays the `small_pgm` procedure. A red box highlights the `PRAGMA INLINE` line. On the right, the "Script Output" window shows the message "Procedure SMALL\_PGM compiled". Below it, the "Compiler - Log" window lists several warnings, including one about the pragma being used. The Oracle logo is at the bottom left, and a copyright notice is at the bottom right.

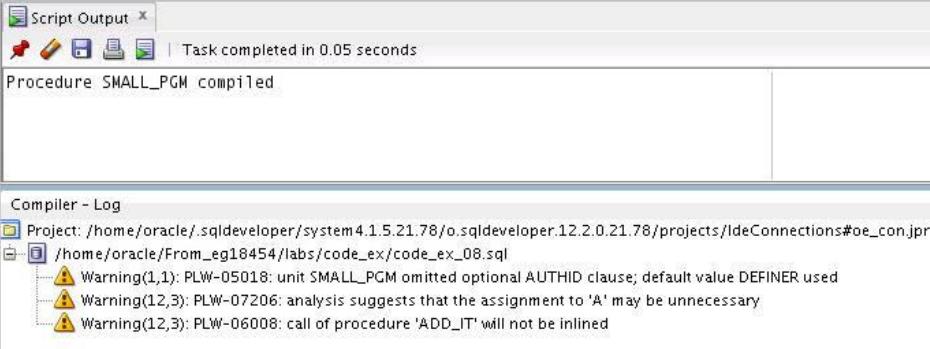
Within a PL/SQL subroutine, you can use `PRAGMA INLINE` to suggest that a specific call be inlined. When using `PRAGMA INLINE`, the first argument is the simple name of a subroutine, a function name, a procedure name, or a method name. The second argument is either the constant string '`NO`' or '`YES`'. The `PRAGMA` can go before any statement or declaration. If you put it in the wrong place, you receive a syntax error message from the compiler.

In the slide you have set the second parameter to '`YES`', to enable inlining.

To identify that a specific call should not be inlined, use:

```
PRAGMA INLINE (function_name, 'NO');
```

Setting the `PRAGMA INLINE` to '`NO`' always works, regardless of any other pragmas that might also apply to the same statement. The pragma also applies at all optimization levels, and it applies no matter how badly the compiler would like to inline a particular call. If you are certain that you do not want some code inlined (perhaps due to the large size), you can set this to `NO`. You can see the response when you enable the compiler warnings.



```

CREATE OR REPLACE PROCEDURE small_pgm
IS
    a PLS_INTEGER;
    FUNCTION add_it(a PLS_INTEGER, b PLS_INTEGER)
    RETURN PLS_INTEGER
    IS
    BEGIN
        RETURN a + b;
    END;
BEGIN
    PRAGMA NOINLINE (add_it);
    a := add_it(3, 4) + 6;
END small_pgm;

```

This screenshot shows the same setup as the previous one, but the `PRAGMA INLINE` line has been replaced with `PRAGMA NOINLINE`. The "Compiler - Log" window now includes a warning about the assignment to variable `a` being unnecessary.

## Inlining: Summary

- Pragmas apply only to calls in the next statement following the pragma.
- Programs that make use of smaller helper subroutines are good candidates for inlining.
- Only local subroutines can be inlined.
- You cannot inline an external subroutine.
- Inlining can increase the size of a unit.
- Use inlining with deterministic functions.



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

The compiler inlines code automatically, provided that you are using native compilation and have set the `PLSQL_OPTIMIZE_LEVEL` to 3. If you have set `PLSQL_Warnings = 'enable:all'`, using the `SQL*Plus SHOW ERRORS` command displays the name of the code that is inlined.

- The `PLW-06004` compiler message tells you that a pragma `INLINE ('YES')` referring to the named procedure was found. The compiler will, if possible, inline this call.
- The `PLW-06005` compiler message tells you the name of the code that is inlined.

Alternatively, you can query the `USER/ALL/DBA_ERRORS` dictionary view.

Deterministic functions compute the same outputs for the same inputs every time the functions are invoked, and have no side effects. The PL/SQL compiler can figure out whether a function is deterministic; it may not find all that truly are, but it finds many of them. It never mistakes a nondeterministic function for a deterministic function.

## Lesson Agenda

- Configuring the compiler
- Enabling intraunit inlining
- Tuning PL/SQL code



ORACLE®

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

## Why PL/SQL Tuning

- Executing PL/SQL code adds a CPU overhead and memory overhead to the application performance.
- Tuning is the process of modifying your code to improve performance, while keeping the functionality intact.
- You must identify programming constructs which are adding overhead to the execution and tune them for better performance.



ORACLE®

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

## Tuning PL/SQL Code

You can tune your PL/SQL code to meet your performance needs by:

- Identifying the data type and constraint issues
  - Data type conversion
  - The NOT NULL constraint
  - PLS\_INTEGER
  - SIMPLE\_INTEGER
- Writing smaller executable sections of code.
- Using bulk binds.
- Using the FORALL support with bulk binding.
- Handling and saving exceptions with the SAVE EXCEPTIONS syntax.
- Tuning Conditional constructs.
- Tuning PL/SQL procedure calls.



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

By tuning your PL/SQL code, you can customize its performance to best meet your needs.

In the following slides, you learn about some of the main PL/SQL tuning issues that can improve the performance of your PL/SQL applications.

## Avoid Implicit Data Type Conversion

- PL/SQL performs implicit conversions between structurally different data types.
- Implicit conversion is context sensitive therefore might generate unpredictable results.
- To avoid unexpected values use explicit conversion wherever required.



ORACLE

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

At run time, PL/SQL automatically performs implicit conversions between structurally different data types. By avoiding implicit conversions, you can improve the performance of your code.

Following guidelines can help you minimize implicit conversions:

1. If a variable is to be either inserted into a table column or assigned a value from a table column, then give the variable the same data type as the table column.
2. Make each literal the same data type as the variable to which it is assigned or the expression in which it appears.
3. Convert values from SQL data types to PL/SQL data types and then use the converted values in expressions.

For example, convert NUMBER values to PLS\_INTEGER values and then use the PLS\_INTEGER values in expressions. PLS\_INTEGER operations use hardware arithmetic, so they are faster than NUMBER operations, which use library arithmetic. To avoid implicit data type conversion, use the following built-in functions:

- TO\_DATE
  - TO\_NUMBER
  - TO\_CHAR
  - CAST
4. Before assigning a value of one SQL data type to a variable of another SQL data type, explicitly convert the source value to the target data type, using a SQL conversion function.
  5. Overload your subprograms with versions that accept parameters of different data types and optimize each version for its parameter types.

## NOT NULL Constraint

```

PROCEDURE calc_m IS
  m NUMBER NOT NULL:=0;
  a NUMBER;
  b NUMBER;
BEGIN
  m := a + b;
END;

```

The value of the expression  $a + b$  is assigned to a temporary variable, which is then tested for nullity.

```

PROCEDURE calc_m IS
  m NUMBER; --no constraint
  ...
BEGIN
  m := a + b;
  IF m IS NULL THEN
    -- raise error
  END IF;
END;

```

This is a better way to check nullity; no performance overhead.

ORACLE

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

In PL/SQL, using the NOT NULL constraint incurs a small performance cost. Therefore, use it with care. Consider the first example in the slide that uses the NOT NULL constraint for  $m$ .

Because  $m$  is constrained by NOT NULL, the value of the expression  $a + b$  is assigned to a temporary variable, which is then tested for nullity. If the variable is not null, its value is assigned to  $m$ . Otherwise, an exception is raised. However, if  $m$  were not constrained, the value would be assigned to  $m$  directly.

A more efficient way to write the same example is with an IF statement as shown in the slide.

Note that the subtypes NATURALN and POSTIVEN are defined as the NOT NULL subtypes of NATURAL and POSITIVE. Using them incurs the same performance cost as seen in the slide's first example.

Using the NOT NULL Constraint	Not Using the Constraint
Slower	Faster
No extra coding is needed.	It requires extra coding, which is error prone.
When an error is implicitly raised, the value of $m$ is preserved.	When an error is explicitly raised, the old value of $m$ is lost.

## PLS\_INTEGER Data Type for Integers

Use PLS\_INTEGER when dealing with integer data.

- It is an efficient data type for integer variables.
- It requires less storage than INTEGER or NUMBER.
- Its operations use machine arithmetic, which is faster than library arithmetic.



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

When you need to declare an integer variable, use the PLS\_INTEGER data type, which is the most efficient numeric type. That is because PLS\_INTEGER values require less storage than INTEGER or NUMBER values, which are represented internally as 22-byte Oracle numbers. Also, PLS\_INTEGER operations use machine arithmetic, so they are faster than BINARY\_INTEGER, INTEGER, or NUMBER operations, which use library arithmetic.

Furthermore, INTEGER, NATURAL, NATURALN, POSITIVE, POSITIVEN, and SIGNTYPE are constrained subtypes. Their variables require precision checking at run time that can affect the performance.

The BINARY\_FLOAT and BINARY\_DOUBLE data types are also faster than the NUMBER data type.

## Using the SIMPLE\_INTEGER Data Type

- Definition:
  - Is a predefined subtype
  - Has the range  $-2147483648 \dots 2147483648$
  - Does not include a null value
  - Is allowed anywhere in PL/SQL where the PLS\_INTEGER data type is allowed
- Benefits:
  - Eliminates the overhead of overflow checking
  - Is estimated to be 2–10 times faster when compared with the PLS\_INTEGER type with native PL/SQL compilation



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

The SIMPLE\_INTEGER data type is a predefined subtype of the BINARY\_INTEGER (or PLS\_INTEGER) data type that has the same numeric range as BINARY\_INTEGER. It differs significantly from PLS\_INTEGER in its overflow semantics. Incrementing the largest SIMPLE\_INTEGER value by one produces the smallest value, and decrementing the smallest value by one produces the largest value. These “wrap around” semantics conform to the Institute of Electrical and Electronics Engineers (IEEE) standard for 32-bit integer arithmetic.

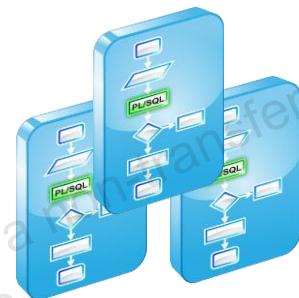
The key features of the SIMPLE\_INTEGER predefined subtype are the following:

- Includes the range of  $-2147483648 \dots +2147483648$
- Has a NOT NULL constraint
- Wraps rather than overflows
- Is faster than PLS\_INTEGER

Without the overhead of overflow checking and null checking, the SIMPLE\_INTEGER data type provides significantly better performance than PLS\_INTEGER when the PLSQL\_CODE\_TYPE parameter is set to native, because arithmetic operations on the former are performed directly in the machine’s hardware. The performance difference is less noticeable when the PLSQL\_CODE\_TYPE parameter is set to interpreted; however, even with this setting, the SIMPLE\_INTEGER type is faster than the PLS\_INTEGER type.

## Modularizing Your Code

- Limit the number of lines of code between a `BEGIN` and an `END` to about a page or 60 lines of code.
- Use packaged programs to keep each executable section small.
- Use local procedures and functions to hide logic.
- Use a function interface to hide formulas and business rules.



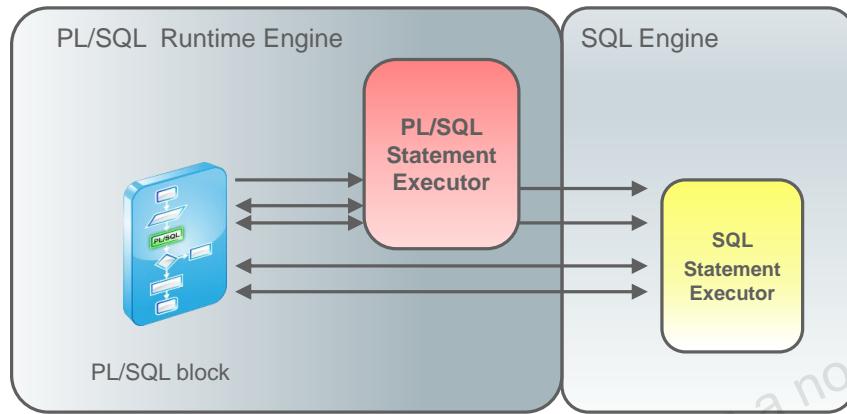
ORACLE®

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

By writing smaller sections of executable code, you can make the code easier to read, understand, and maintain. When developing an application, use a stepwise refinement. Make a general description of what you want your program to do, and then implement the details in subroutines. Using local modules and packaged programs can help keep each executable section small. This makes it easier for you to debug and refine your code.

## Bulk Binding

Use bulk binds to reduce context switches between the PL/SQL engine and the SQL engine.



ORACLE®

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

While executing a PL/SQL block, based on the requirement there might be context switches between PL/SQL statement executor and SQL statement executor.

With bulk binds, you can improve performance by decreasing the number of context switches between the SQL and PL/SQL engines. When a PL/SQL program executes, each time a SQL statement is encountered, there is a switch between the PL/SQL engine and the SQL engine. The more the number of switches, the less the efficiency.

### Improved Performance

Bulk binding enables you to implement array fetching. With bulk binding, entire collections, not just individual elements, are passed back and forth. Bulk binding can be used with nested tables, varrays, and associative arrays.

The more the rows affected by a SQL statement, the greater is the performance gain with bulk binding.

## FORALL Instead of FOR

Bind whole arrays of values simultaneously, rather than looping to perform fetch, insert, update, and delete on multiple rows.

If you have to execute a FOR loop, instead of:

```
...
FOR i IN 1 .. 50000 LOOP
  INSERT INTO bulk_bind_example_tbl
    VALUES(...);
END LOOP; ...
```

Use:

```
...
FORALL i IN 1 .. 50000
  INSERT INTO bulk_bind_example_tbl
    VALUES(...);
...
```



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

In the first example shown in the slide, one row at a time is inserted into the target table. In the second example, the FOR loop is changed to a FORALL (which has an implicit loop) and all the immediately subsequent DML statements are processed in bulk. The entire code examples, along with the timing statistics for running each FOR loop example, are as follows.

First, create the demonstration table:

```
CREATE TABLE bulk_bind_example_tbl (
  num_col NUMBER,
  date_col DATE,
  char_col VARCHAR2(40));
```

Then, set the SQL\*Plus TIMING variable on. Setting it on enables you to see the approximate elapsed time of the last SQL statement:

```
SET TIMING ON
```

Then, run this block of code that includes a FOR loop to insert 50,000 rows:

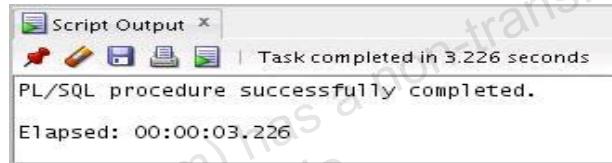
```
DECLARE
  TYPE typ_numlist IS TABLE OF NUMBER;
  TYPE typ_datelist IS TABLE OF DATE;
  TYPE typ_charlist IS TABLE OF VARCHAR2(40)
    INDEX BY PLS_INTEGER;
```

```

n typ_numlist := typ_numlist();
d typ_datelist := typ_datelist();
c typ_charlist;

BEGIN
  FOR i IN 1 .. 50000 LOOP
    n.extend;
    n(i) := i;
    d.extend;
    d(i) := sysdate + 1;
    c(i) := lpad(1, 40);
  END LOOP;
  FOR I in 1 .. 50000 LOOP
    INSERT INTO bulk_bind_example_tbl
      VALUES (n(i), d(i), c(i));
  END LOOP;
END;
/

```



Finally, run this block of code that includes a `FORALL` loop to insert 50,000 rows. Note the significant decrease in the timing when using `FORALL` processing:

```

DECLARE
  TYPE typ_numlist IS TABLE OF NUMBER;
  TYPE typ_datelist IS TABLE OF DATE;
  TYPE typ_charlist IS TABLE OF VARCHAR2(40)
    INDEX BY PLS_INTEGER;
  n typ_numlist := typ_numlist();
  d typ_datelist := typ_datelist();
  c typ_charlist;

BEGIN
  FOR i IN 1 .. 50000 LOOP
    n.extend;
    n(i) := i;
    d.extend;
    d(i) := sysdate + 1;
    c(i) := lpad(1, 40);
  END LOOP;
  FORALL I in 1 .. 50000
    INSERT INTO bulk_bind_example_tbl
      VALUES (n(i), d(i), c(i));
END;
/

```



## BULK COLLECT

Use BULK COLLECT to improve performance:

```

CREATE OR REPLACE PROCEDURE process_customers
  (p_account_mgr customers.account_mgr_id%TYPE)
IS
  TYPE typ_numtab IS TABLE OF
    customers.customer_id%TYPE;
  TYPE typ_chartab IS TABLE OF
    customers.cust_last_name%TYPE;
  TYPE typ_emailtab IS TABLE OF
    customers.cust_email%TYPE;
  v_custnos    typ_numtab;
  v_last_names typ_chartab;
  v_emails     typ_emailtab;
BEGIN
  SELECT customer_id, cust_last_name, cust_email
    BULK COLLECT INTO v_custnos, v_last_names, v_emails
   FROM customers
  WHERE account_mgr_id = p_account_mgr;
  ...
END process_customers;

```

**ORACLE**

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

When you require a large number of rows to be returned from the database, you can use the BULK COLLECT option for queries. This option enables you to retrieve multiple rows of data in a single request. The retrieved data is then populated into a series of collection variables. This query runs significantly faster than if it were done without the BULK COLLECT.

You can use the BULK COLLECT option with explicit cursors too:

```

BEGIN
  OPEN cv_customers INTO customers_rec;
  FETCH cv_customers BULK COLLECT INTO
    v_custnos, v_last_name, v_mails;
  ...

```

You can also use the LIMIT option with BULK COLLECT. This gives you control over the amount of processed rows in one step.

```

FETCH cv_customers BULK COLLECT
  INTO v_custnos, v_last_name, v_email
  LIMIT 200;

```

## BULK COLLECT

Use the RETURNING clause to retrieve information about the rows that are being modified:

```

DECLARE
    TYPE      typ_replist IS VARRAY(100) OF NUMBER;
    TYPE      typ_numlist IS TABLE OF
                orders.order_total%TYPE;
    repids   typ_replist := 
                typ_replist(153, 155, 156, 161);
    totlist  typ_numlist;
    c_big_total CONSTANT NUMBER := 60000;
BEGIN
    FORALL i IN repids.FIRST..repids.LAST
        UPDATE orders
        SET order_total = .95 * order_total
        WHERE sales_rep_id = repids(i)
        AND order_total > c_big_total
        RETURNING order_total BULK COLLECT INTO Totlist;
END;

```



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

Often, applications need information about the row that is affected by a SQL operation; for example, to generate a report or take action. Using the RETURNING clause, you can retrieve information about the rows that you modified with the INSERT, UPDATE, and DELETE statements. This can improve performance, because it enables you to make changes, and at the same time, collect information about the data being changed. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required. Without the RETURNING clause, you need two operations: one to make the change, and a second operation to retrieve information about the change. In the example in the slide, the order\_total information is retrieved from the ORDERS table and collected into the totlist collection. The totlist collection is returned in bulk to the PL/SQL engine.

If you did not use the RETURNING clause, you would need to perform two operations: one for the UPDATE, and another for the SELECT:

```

UPDATE orders SET order_total = .95 * order_total
WHERE sales_rep_id = p_id
AND order_total > c_big_total;

SELECT order_total FROM orders
WHERE sales_rep_id = p_id AND order_total > c_big_total;

```

In the following example, you update the credit limit of a customer and at the same time retrieve the customer's new credit limit into a SQL Developer environment variable:

```
CREATE OR REPLACE PROCEDURE change_credit
  (p_in_id    IN    customers.customer_id%TYPE,
   o_credit OUT NUMBER)
IS
BEGIN
  UPDATE customers
  SET    credit_limit = credit_limit * 1.10
  WHERE  customer_id = p_in_id
  RETURNING credit_limit INTO o_credit;
END change_credit;
/
VARIABLE g_credit NUMBER
EXECUTE change_credit(109, :g_credit)
PRINT g_credit
```

## Exception While Bulk Collecting

- You can use the `SAVE EXCEPTIONS` keyword in your `FORALL` statements:

```
FORALL index IN lower_bound..upper_bound
  SAVE EXCEPTIONS
    {insert_stmt | update_stmt | delete_stmt}
```

- Exceptions raised during execution are saved in the `%BULK_EXCEPTIONS` cursor attribute.
- The attribute is a collection of records with two fields:

Field	Definition
<code>ERROR_INDEX</code>	Holds the iteration of the <code>FORALL</code> statement where the exception was raised
<code>ERROR_CODE</code>	Holds the corresponding Oracle error code



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

To handle the exceptions encountered during a BULK BIND operation, you can add the keyword `SAVE EXCEPTIONS` to your `FORALL` statement. Without it, if a row fails during the `FORALL` loop, the loop execution is terminated. `SAVE_EXCEPTIONS` allows the loop to continue processing and is required if you want the loop to continue.

All exceptions raised during the execution are saved in the `%BULK_EXCEPTIONS` cursor attribute, which stores a collection of records. This cursor attribute is available only from the exception handler.

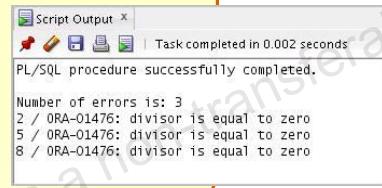
Each record has two fields. The first field, `%BULK_EXCEPTIONS(i).ERROR_INDEX`, holds the “iteration” of the `FORALL` statement during which the exception was raised. The second field, `BULK_EXCEPTIONS(i).ERROR_CODE`, holds the corresponding Oracle error code.

The values stored by `%BULK_EXCEPTIONS` always refer to the most recently executed `FORALL` statement. The number of exceptions is saved in the count attribute of `%BULK_EXCEPTIONS`; that is, `%BULK_EXCEPTIONS.COUNT`. Its subscripts range from 1 to `COUNT`. If you omit the `SAVE EXCEPTIONS` keyword, execution of the `FORALL` statement stops when an exception is raised. In that case, `SQL%BULK_EXCEPTIONS.COUNT` returns 1, and `SQL%BULK_EXCEPTIONS` contains just one record. If no exception is raised during the execution, `SQL%BULK_EXCEPTIONS.COUNT` returns 0.

## Handling FORALL Exceptions

```

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  num_tab  NumList :=
    NumList(100,0,110,300,0,199,200,0,400);
  bulk_errors EXCEPTION;
  PRAGMA      EXCEPTION_INIT (bulk_errors, -24381 );
BEGIN
  FORALL i IN num_tab.FIRST..num_tab.LAST
  SAVE EXCEPTIONS
  DELETE FROM orders WHERE order_total < 500000/num_tab(i);
EXCEPTION WHEN bulk_errors THEN
  DBMS_OUTPUT.PUT_LINE('Number of errors is: '
    || SQL%BULK_EXCEPTIONS.COUNT);
  FOR j in 1..SQL%BULK_EXCEPTIONS.COUNT
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      TO_CHAR(SQL%BULK_EXCEPTIONS(j).error_index) ||
      ' ' ||
      SQLERRM(-SQL%BULK_EXCEPTIONS(j).error_code) );
  END LOOP;
END;
  
```



**ORACLE**

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

In the example in the slide, the `EXCEPTION_INIT` pragma defines an exception named `BULK_ERRORS` and associates the name with the `ORA-24381` code, which is an “Error in Array DML.” The PL/SQL block raises the predefined exception `ZERO_DIVIDE` when `i` equals 2, 5, 8. After the bulk bind is completed, `SQL%BULK_EXCEPTIONS.COUNT` returns 3, because the code tried to divide by zero three times. To get the Oracle error message (which includes the code), you pass `SQL%BULK_EXCEPTIONS(i).ERROR_CODE` to the error-reporting function `SQLERRM`.

## Tuning Conditional Control Statements

In logical expressions, PL/SQL stops evaluating the expression as soon as the result is determined.

- In an OR conditional statement, place the condition which is most likely to evaluate TRUE first in the evaluation order.
- In an AND conditional statement, place the condition which is most likely to evaluate FALSE first evaluation order.



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

In logical expressions, improve performance by carefully ordering conditional constructs.

When evaluating a logical expression, PL/SQL stops evaluating the expression as soon as the result is determined. While evaluating an OR condition, the evaluation would stop when it encounters a TRUE condition. Therefore place the condition, which is most likely to evaluate to TRUE first in evaluation order.

In case of an AND expression, the evaluation of the expression will stop when it encounters a FALSE condition. Therefore place the condition, which is most likely to evaluate to FALSE first in the evaluation order.

Such placement of conditions will reduce redundant comparisons, thus improving the performance.

## Tuning Conditional Control Statements

If your business logic results in one condition being true, use the `ELSIF` syntax for mutually exclusive clauses:

```
IF v_acct_mgr = 145 THEN
    process_acct_145;
END IF;
IF v_acct_mgr = 147 THEN
    process_acct_147;
END IF;
IF v_acct_mgr = 148 THEN
    process_acct_148;
END IF;
IF v_acct_mgr = 149 THEN
    process_acct_149;
END IF;
```

```
IF v_acct_mgr = 145
THEN
    process_acct_145;
ELSIF v_acct_mgr = 147
THEN
    process_acct_147;
ELSIF v_acct_mgr = 148
THEN
    process_acct_148;
ELSIF v_acct_mgr = 149
THEN
    process_acct_149;
END IF;
```



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

If you have a situation where you are checking a list of choices for a mutually exclusive result, use the `ELSIF` syntax, because it offers the most efficient implementation. With `ELSIF`, after a branch evaluates to TRUE, the other branches aren't executed.

In the example shown on the right in the slide, every `IF` statement is executed. In the example on the left, after a branch is found to be true, the rest of the branch conditions aren't evaluated.

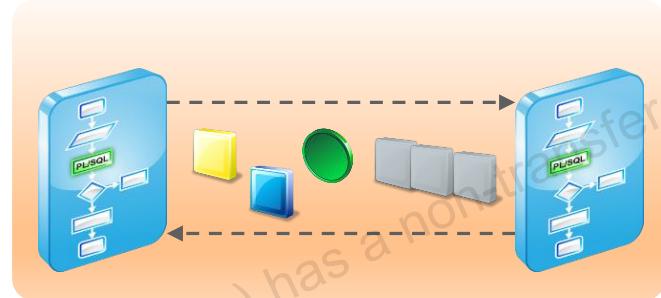
Sometimes you don't need an `IF` statement. For example, the following code can be rewritten without an `IF` statement:

```
IF date_ordered < sysdate + 7 THEN
    late_order := TRUE;
ELSE
    late_order := FALSE;
END IF;

--rewritten without an IF statement:
late_order := date_ordered < sysdate + 7;
```

## Passing Data Between PL/SQL Programs

- The flexibility built into PL/SQL enables you to pass:
  - Simple scalar variables
  - Complex data structures
- You can use the `NOCOPY` hint to improve performance with the `IN OUT` parameters.



ORACLE®

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

You can pass simple scalar data or complex data structures between PL/SQL programs.

When passing collections as parameters, you may encounter a slight decrease in performance as compared with passing scalar data, but the performance is still comparable. However, when passing `IN OUT` parameters that are complex (such as collections) to a procedure, you will experience significantly more overhead, because a copy of the parameter value is stored before the routine is executed. The stored value must be kept in case an exception occurs.

You can use the `NOCOPY` compiler hint to improve performance in this situation. `NOCOPY` instructs the compiler not to make a backup copy of the parameter that is being passed. However, be careful when you use the `NOCOPY` compiler hint, because your results are not predictable if your program encounters an exception.

## Passing Data Between PL/SQL Programs

Pass records as parameters to encapsulate data, and write and maintain less code:

```

DECLARE
  TYPE CustRec IS RECORD (
    customer_id      customers.customer_id%TYPE,
    cust_last_name   VARCHAR2(20),
    cust_email       VARCHAR2(30),
    credit_limit     NUMBER(9,2));
  ...
  PROCEDURE raise_credit (cust_info CustRec);

```



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

You can declare user-defined records as formal parameters of procedures and functions as shown in the slide. By using records to pass values, you are encapsulating the data being passed. This requires less coding than defining, assigning, and manipulating each record field individually.

When you call a function that returns a record, use the notation:

function\_name(parameters).field\_name

For example, the following call to the NTH\_HIGHEST\_ORD\_TOTAL function references the ORDER\_TOTAL field in the ORD\_INFO record:

```

DECLARE
  TYPE OrdRec IS RECORD (
    v_order_id      NUMBER(6),
    v_order_total   REAL,
    v_middle_total  REAL);
  FUNCTION nth_highest_total (n INTEGER) RETURN OrdRec IS
    order_info OrdRec;
  BEGIN
    ...
    RETURN order_info; -- return record
  END;
  BEGIN
    -- call function
    v_middle_total := nth_highest_total(10).v_order_total;
  ...

```

## Passing Data Between PL/SQL Programs

Use collections as arguments:

```
PACKAGE cust_actions IS
    TYPE NameTabTyp IS TABLE OF
        customer.cust_last_name%TYPE
        INDEX BY PLS_INTEGER;
    TYPE CreditTabTyp IS TABLE OF
        customers.credit_limit%TYPE
        INDEX BY PLS_INTEGER;
    ...
    PROCEDURE credit_batch( name_tab    IN NameTabTyp ,
                           credit_tab  IN CreditTabTyp,
                           ... );
    PROCEDURE log_names ( name_tab IN NameTabTyp );
END cust_actions;
```



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

You can declare collections as formal parameters of procedures and functions. In the example in the slide, associative arrays are declared as the formal parameters of two packaged procedures. If you were to use scalar variables to pass the data, you would need to code and maintain many more declarations.

## Quiz



Which of the following statements are true?

- a. Use the native mode during development.
- b. Because the native code does not have to be interpreted at run time, it runs faster.
- c. The interpreted compilation is the default compilation method.
- d. To change a compiled PL/SQL object from interpreted code type to native code type, you must set the `PLSQL_CODE_TYPE` parameter to `NATIVE`, and then recompile the program.

ORACLE

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

**Answer: b, c, d**

## Quiz



You can tune your PL/SQL code by:

- a. Writing longer executable sections of code
- b. Avoiding bulk binds
- c. Using the `FORALL` support with bulk binding
- d. Handling and saving exceptions with the `SAVE EXCEPTIONS` syntax
- e. Rephrasing conditional statements

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

ORACLE



**Answer: c, d, e**

## Quiz



Which of the following statements are true with reference to inlining?

- a. Pragmas apply only to calls in the next statement following the pragma.
- b. Programs that make use of smaller helper subroutines are bad candidates for inlining.
- c. Only local subroutines can be inlined.
- d. You cannot inline an external subroutine.
- e. Inlining can decrease the size of a unit.

**ORACLE**

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

**Answer: a, c, d**

## Summary

In this lesson, you should have learned how to:

- Decide when to use native or interpreted compilation
- Tune your PL/SQL application. Tuning involves:
  - Using the RETURNING clause and bulk binds when appropriate
  - Rephrasing conditional statements
  - Identifying data type and constraint issues
  - Understanding when to use SQL and PL/SQL
- Identify opportunities for inlining PL/SQL code
- Use native compilation for faster PL/SQL execution



ORACLE

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

There are several methods that help you tune your PL/SQL application.

When tuning PL/SQL code, consider using the RETURNING clause and bulk binds to improve processing. Be aware of conditional statements with an OR clause. Place the fastest processing condition first. There are several data type and constraint issues that can help in tuning an application.

By using native compilation, you can benefit from performance gains for computation-intensive procedural operations.

## Practice 8: Overview

This practice covers the following topics:

- Tuning PL/SQL code to improve performance
- Coding with bulk binds to improve performance



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

In this practice, you tune some of the code that you created for the OE application.

- Break a previously built subroutine into smaller executable sections
- Pass collections into subroutines
- Add error handling for BULK INSERT

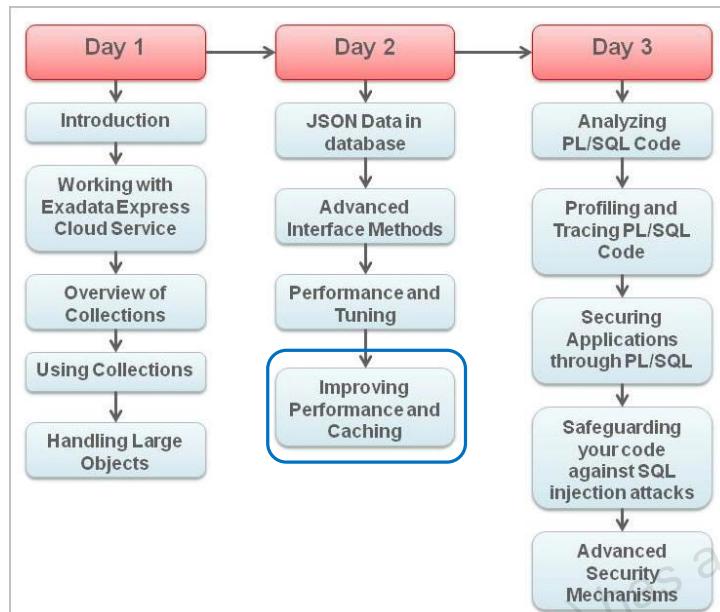
Use the OE schema for this practice.

# Improving Performance with Caching



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

## Course Agenda



ORACLE®

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



## Objectives

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

- Understand Result Cache
- Write queries that use the result cache hint
- Use the DBMS\_RESULT\_CACHE package
- Set up PL/SQL functions to use PL/SQL result caching



ORACLE

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

In this lesson, you learn about the Oracle Database 12c caching techniques that can improve performance. You examine the improvement on the performance of queries by caching the results of a query in memory, and then using the cached results in future executions of the query or query fragments. The cached results reside in the result cache memory portion of the shared global area (SGA).

The result-caching mechanism of the PL/SQL cross-session function provides applications with a language-supported and system-managed means for storing the results of PL/SQL functions in an SGA, which is available to every session that runs the application.

## Lesson Agenda

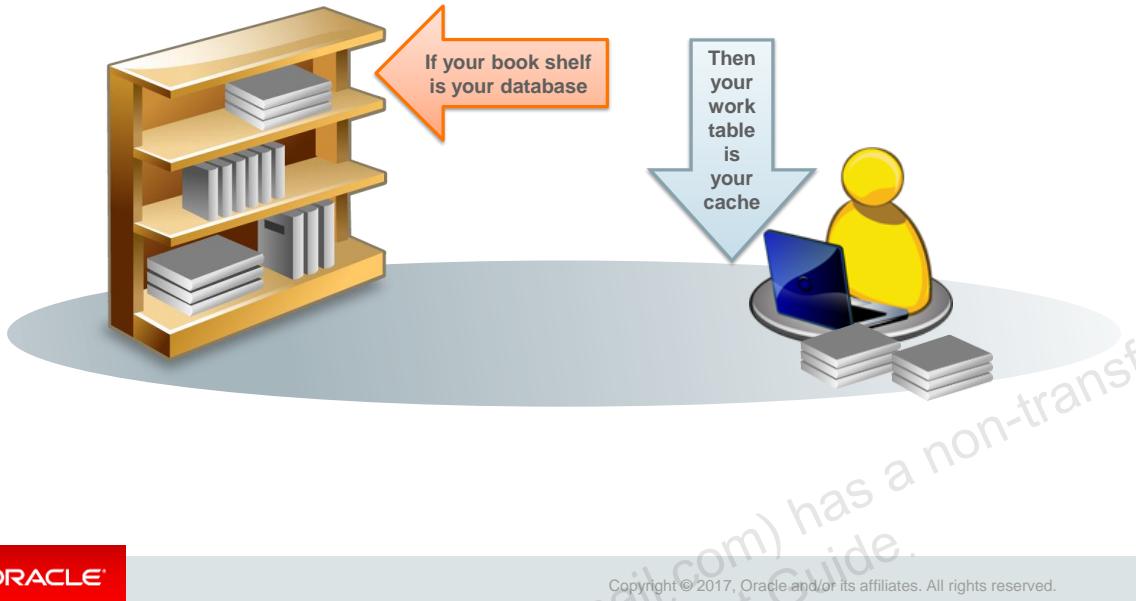
- Caching in the Database Instance
- Result Caching
- SQL Result Caching
- PL/SQL Result Caching
- Oracle Database In-Memory



ORACLE®

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

## What Is Caching?



ORACLE®

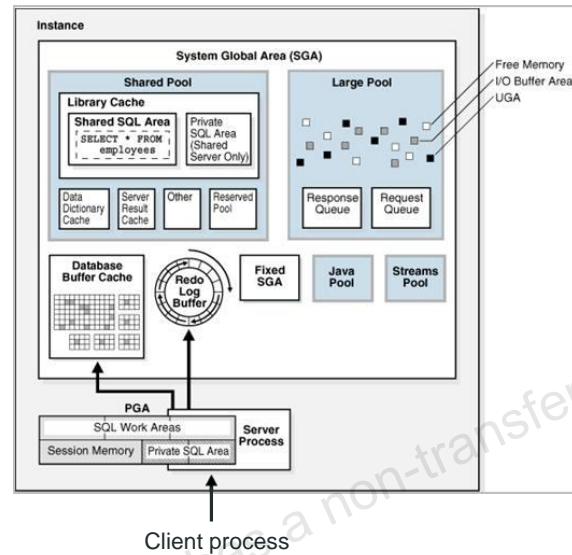
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Caching is a technique used to improve the performance of an application. The performance overhead on applications is generally due to memory access and I/O. In order to reduce this overhead you create a small high speed memory area called cache. This cache stores frequently used data, thus reducing the overhead of frequent memory access.

As shown in the slide, if a bookshelf is your database where you have all the files then your work table is your cache. You prefer to keep frequently used books on your work table to avoid the overhead of walking to the bookshelf to fetch the book. Oracle database instance uses caching at various levels in its memory architecture to improve performance.

## Memory Architecture

- A database instance has the following memory structures
  - System Global Area(SGA).
  - Program Global Area(PGA).
  - User Global Area(UGA).
  - Software code areas.



ORACLE

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

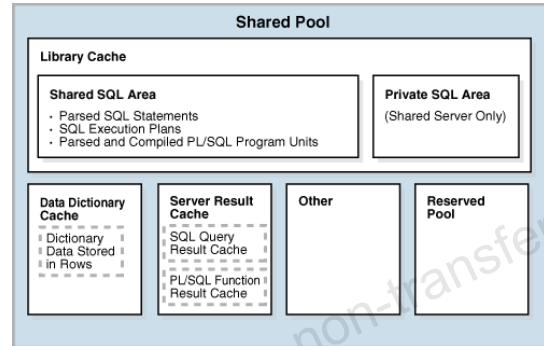
An Oracle Database server comprises of database and a database instance. Database is the set of files stored on the disk. Database instance refers to a set of memory structures that manage database files. An instance can exist independently of the database.

A database instance comprises of the following memory structures:

- **System Global Area (SGA)**  
The SGA is a group of shared memory structures that contain data and control information for one database instance. Examples of SGA components include the database buffer cache and shared SQL areas. Starting in Oracle Database 12c Release 1 (12.1.0.2), the SGA can contain an In Memory Column Store (IM column store), which enables data to be populated in memory in a columnar format.
- **Program Global Area(PGA)**  
A PGA is a memory region that contain data and control information for a server or background process. Each client accessing the database a PGA associated with it.
- **User Global Area(UGA)**  
The UGA is memory associated with a user session. Stores the session state of an user.
- **Software code areas**  
Software code areas are portions of memory used to store code that is being run or can be run.

## Caching in the Database Instance

- Database Buffer Cache in the database instance stores currently being used or recently used database blocks.
- The shared pool of the database instance has the following cache components:
  - Library cache.
  - Data Dictionary cache.
  - Server Result cache(optional).



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

The database buffer cache, also called the *buffer cache*, is the memory area that stores copies of data blocks read from data files. A buffer is a main memory address in which the buffer manager temporarily caches a currently or recently used data block to optimize physical I/O. All users concurrently connected to a database instance share access to the buffer cache.

The shared pool in the SGA has Library Cache, Data Dictionary Cache and an optional Server Result Cache.

The Library cache stores the executable (parsed or compiled) form of recently referenced SQL and PL/SQL code.

The data dictionary cache stores data referenced from the data dictionary.

The server result cache is an optional cache that stores query and PL/SQL function results within the shared pool.

You can tune the performance of the application by tuning the configuration of Database Buffer Cache, Library Cache and Data Dictionary Cache. You can learn more on this in the course Oracle Database 12c: Performance Management and Tuning Ed1.

Server Result Cache is an optional cache, you can define whether the result of a SQL query or PL/SQL unit can be cached or not at the query level or unit level. You may also configure your instance to not use the result cache. We will look at the usage of Result Cache in the following slides.

## What Is Result Caching?

- You can store SQL query and PL/SQL function results in a result cache.
- Subsequent executions of the same query or function can be served directly out of the cache, improving response times.
- This technique can be especially effective for SQL queries and PL/SQL functions that are executed frequently with the same parameters
- Cached query results become invalid when the database data accessed by the query is modified.
- Result caching brings huge benefits when the data in the database doesn't change often.



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

The new SQL query result cache enables explicit caching of queries and query fragments in an area of the shared pool called “result cache memory.” When a query is executed, the result cache is built up and the result is returned. The database can then use the cached results for subsequent query executions with the same parameters, thereby resulting in faster response times. Cached query results become invalid when the data in the database objects being accessed by the query is modified.

## Lesson Agenda

- Caching in the Database Instance
- Result Caching
- SQL Result Caching
- PL/SQL Result Caching
- Oracle Database In-Memory



ORACLE®

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

## Configuring the Server Result Cache

- The size of the result cache is relative to the size of the shared pool and the memory management mechanism in place.
- You can use the following initialization parameters to configure the result cache:
  - RESULT\_CACHE\_MAX\_SIZE
  - RESULT\_CACHE\_MAX\_RESULT
  - RESULT\_CACHE\_REMOTE\_EXPIRATION



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

By default, on database startup, Oracle Database allocates memory to the server result cache in the shared pool. The memory size allocated depends on the memory size of the shared pool and the selected memory management system:

- **Automatic shared memory management:** If you are managing the size of the shared pool using the SGA\_TARGET initialization parameter, Oracle Database allocates 0.50% of the value of the SGA\_TARGET parameter to the result cache.
- **Manual shared memory management:** If you are managing the size of the shared pool using the SHARED\_POOL\_SIZE initialization parameter, then Oracle Database allocates 1% of the shared pool size to the result cache.

The size of the server result cache grows until it reaches the maximum size. Query results larger than the available space in the cache are not cached. The database employs a Least Recently Used (LRU) algorithm to age out cached results, but does not otherwise automatically release memory from the server result cache.

Parameter	Description
RESULT_CACHE_MAX_SIZE	Specifies the memory allocated to the server result cache. To disable the server result cache, set this parameter to 0.
RESULT_CACHE_MAX_RESULT	Specifies the maximum amount of server result cache memory (in percent) that can be used for a single result. Valid values are between 1 and 100. The default value is 5%. You can set this parameter at the system or session level.
RESULT_CACHE_REMOTE_EXPIRATION	Specifies the expiration time (in minutes) for a result in the server result cache that depends on remote database objects. The default value is 0, which specifies that results using remote objects will not be cached. If a non-zero value is set for this parameter, DML on the remote database does not invalidate the server result cache. Setting this parameter to a nonzero value can produce stale answers (for example, if the remote table used by a result is modified at the remote database).

## Setting Result\_Cache\_Max\_Size

- Set `Result_Cache_Max_Size` from the command line or in an initialization file created by a DBA.
- The cache size is dynamic and can be changed either permanently or until the instance is restarted.

```
SQL> SELECT name, value
  2  FROM  v$parameter
  3 WHERE  name = 'result_cache_max_size';
```

Query Result	
SQL   All Rows Fetched: 1 in 0.002 seconds	
NAME	VALUE
1 result_cache_max_size	12353536



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

By default, the server-side result cache is configured to use a very small portion of the shared pool. You can manually set the result cache memory size by using the `RESULT_CACHE_MAX_SIZE` initialization parameter. Setting `RESULT_CACHE_MAX_SIZE` to 0 during database startup disables the server-side result cache. `RESULT_CACHE_MAX_SIZE` cannot be dynamically changed if the value is set to 0 during database startup in the `SPFILE` (server parameter file) or the `init.ora` (initialization) file.

## Setting the Result Cache Mode

- Use the `RESULT_CACHE_MODE` initialization parameter in the database initialization parameter file to set the result cache mode at the database level
- `RESULT_CACHE_MODE` can be set to:
  - **MANUAL** (default): You must add the `RESULT_CACHE` hint to your queries for the results to be cached.
  - **FORCE**: Results are always stored in the result cache memory, if possible.
- You can set the result cache mode at table level using `ALTER TABLE` command.
- You can set the result cache mode at the query level or PL/SQL program unit level explicitly.



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

You can enable query result cache at the database level by using the `RESULT_CACHE_MODE` initialization parameter in the database initialization parameter file. The same parameter can also be used at the session level by using the `ALTER SESSION` command. `RESULT_CACHE_MODE` can be set to:

- **MANUAL (default):** You must add the `RESULT_CACHE` hint to your queries for the results to be cached or to be served out of the cache. The `RESULT_CACHE` hint can also be added in subqueries and inline views.
- **FORCE:** Results are always stored in the result cache memory, if possible. The result may not be cached in situations such as the result exceeds the maximum result size or cache size.

The use of the SQL query result cache introduces the `ResultCache` operator in the query execution plan.

## Using the DBMS\_RESULT\_CACHE Package

The DBMS\_RESULT\_CACHE package provides an interface for a DBA to manage memory allocation for SQL query result cache and the PL/SQL function result cache.

```
EXECUTE DBMS_RESULT_CACHE.MEMORY_REPORT
```

```
Script Output x | Task completed in 0.026 seconds
PL/SQL procedure successfully completed.

Result Cache Memory Report
[Parameters]
Block Size = 1K bytes
Maximum Cache Size = 12064K bytes (12064 blocks)
Maximum Result Size = 603K bytes (603 blocks)
[Memory]
Total Memory = 9504 bytes [0.001% of the Shared Pool]
... Fixed Memory = 9504 bytes [0.001% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]
```

**ORACLE**

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

You can use the DBMS\_RESULT\_CACHE package to perform various operations, such as bypassing the cache, retrieving statistics on the cache memory usage, and flushing the cache. For example, to view the memory allocation statistics, use `dbms_result_cache.memory_report`. Following are DBMS\_RESULT\_CACHE sub programs:

Subprogram	Description
BYPASS Procedure	Sets the bypass mode for the Result Cache
FLUSH function and procedure	Attempts to remove all the objects from the Result Cache, and depending on the arguments retains or releases the memory and retains or clears the statistics
INVALIDATE functions and procedures	Invalidates all the result-set objects that dependent upon the specified dependency object
INVALIDATE_OBJECT functions and procedures	Invalidates the specified result-set object(s)
MEMORY_REPORT procedure	Produces the memory usage report for the Result Cache
STATUS function	Checks the status of the Result Cache

## Lesson Agenda

- Caching in the Database Instance
- Result Caching
- SQL Result Caching
- PL/SQL Result Caching
- Oracle Database In-Memory



ORACLE®

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

## SQL Query Result Cache

- SQL Query Result Cache holds the results of a SQL query or a query fragment.
- The cached results are reused to improve performance when the query is executed again.



ORACLE®

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

You can improve the performance of your queries by caching the results of a query in memory, and then using the cached results in future executions of the query or query fragments. The cached results reside in the result cache memory portion of the SGA. This feature is designed to speed up query execution on systems with large memories.

## SQL Query Result Cache

- Scenario:
  - You need to find the greatest average value of credit limit grouped by state over the whole population.
  - The query returns a large number of rows being analyzed to yield a few rows or one row.
  - In your query, the data changes fairly slowly (say every hour), but the query is repeated fairly often (say every second).
- Solution:
  - Use the new optimizer hint `/*+ result_cache */` in your query:

```
SELECT /*+ result_cache */  
       AVG(cust_credit_limit), cust_state_province  
  FROM sh.customers  
 GROUP BY cust_state_province;
```



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

SQL result caching is useful when your queries need to analyze a large number of rows to return a small number of rows or a single row.

Two new optimizer hints are available to turn on and turn off SQL result caching:

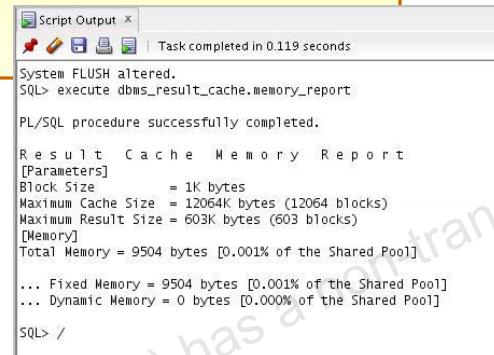
```
/*+ result_cache */  
/*+ no_result_cache */
```

These hints enable you to override the settings of the `RESULT_CACHE_MODE` initialization parameter.

You can execute `DBMS_RESULT_CACHE.MEMORY_REPORT` to produce a memory usage report of the result cache.

## Examining the Memory Cache

```
--- flush.sql  
--- Start with a clean slate. Flush the cache and shared pool.  
--- Verify that memory was released.  
SET ECHO ON  
SET FEEDBACK 1  
SET SERVEROUTPUT ON  
  
execute dbms_result_cache.flush  
alter system flush shared_pool  
/  
execute dbms_result_cache.memory_report
```



```
Script Output X | Task completed in 0.119 seconds  
System FLUSH altered.  
SQL> execute dbms_result_cache.memory_report  
PL/SQL procedure successfully completed.  
  
Result Cache Memory Report  
[Parameters]  
Block Size = 1K bytes  
Maximum Cache Size = 12064K bytes (12064 blocks)  
Maximum Result Size = 603K bytes (603 blocks)  
[Memory]  
Total Memory = 9504 bytes [0.001% of the Shared Pool]  
... Fixed Memory = 9504 bytes [0.001% of the Shared Pool]  
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]  
SQL> /
```



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

Executing the `memory_report` procedure of the `DBMS_RESULT_CACHE` package provides information about the result cache.

## Examining the Execution Plan for a Query

```
--- plan_query1.sql
--- Generate the execution plan.
--- (The query name Q1 is optional)
explain plan for
  select /*+ result_cache q_name(Q1) */ * from orders;

--- Display the execution plan.
select plan_table_output from
  table(dbms_xplan.display('plan_table',null,'serial'));
```

Id	Operation	Name	Rows	Bytes	Cost (%)CPU)	Time
0	SELECT STATEMENT		105	3885	3 (0)	00:00:01
1	RESULT CACHE	b3kc4dpapvgxf0v4futzyz8v				
2	TABLE ACCESS FULL	ORDERS	105	3885	3 (0)	00:00:01

ORACLE

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

You examine the execution plan for two queries, and then execute both the queries. After executing both the queries, you view the memory allocation and usage statistics.

First, execute the code shown in the slide and examine the execution plan for the first query. The query uses the `RESULT_CACHE` optimizer hint.

## Examining Another Execution Plan

```

--- plan_query2.sql
set echo on
--- Generate the execution plan.(The query name Q2 is optional)
explain plan for
  select c.customer_id, o.ord_count
  from (select /*+ result_cache q_name(Q2) */
         customer_id, count(*) ord_count
        from orders
       group by customer_id) o, customers c
  where o.customer_id = c.customer_id;

--- Display the execution plan.
--- using the code in ORACLE_HOME/rdbms/admin/utlxpls
select plan_table_output from table(dbms_xplan.display('plan_table',
  null,'serial'));

```

```

Result Cache Information (identified by operation id):
-----
3 - column-count=2; dependencies=(DE.ORDERS); name="select /*+ result_cache q_name(Q2) */ customer_id, count(*) ord_count from orders group by customer_id"

```



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

Execute the code shown in the slide and examine the execution plan for the second query. This query also uses the RESULT\_CACHE optimizer hint.

The screenshot shows the Oracle SQL Developer interface with the 'PLAN\_TABLE\_OUTPUT' tab selected. The output displays the execution plan for the second query, which includes a result cache entry for the first query and various execution steps for the second query itself.

```

Script Output X Query Result X
SQL | All Rows Fetched: 25 in 0.025 seconds
PLAN_TABLE_OUTPUT
1 Plan hash value: 2892511806
2
3
4 | Id | Operation          | Name           | Rows | Bytes | Cost (%CPU) | Time      |
5 -----
6 |   0 | SELECT STATEMENT   |                | 47  | 1410  |    1 (0) | 00:00:01 |
7 |   1 |  NESTED LOOPS      |                | 47  | 1410  |    1 (0) | 00:00:01 |
8 |* 2 |   VIEW              |                | 47  | 1222  |    1 (0) | 00:00:01 |
9 |  3 |   RESULT CACHE      | 2vaWk7zuptxxw7zddpsnpp96gz | 1    | 0     |    1 (0) | 00:00:01 |
10 |  4 |   HASH GROUP BY     |                | 47  | 188   |    1 (0) | 00:00:01 |
11 |  5 |   INDEX FULL SCAN  | ORD_CUSTOMER_IX | 105 | 420   |    1 (0) | 00:00:01 |
12 |* 6 |   INDEX UNIQUE SCAN | CUSTOMERS_PK    | 1    | 4     |    0 (0) | 00:00:01 |
13 -----
14
15 Predicate Information (identified by operation id):
16 -----
17
18 2 - filter("O"."CUSTOMER_ID>0)
19 6 - access("O"."CUSTOMER_ID"="C"."CUSTOMER_ID")
20   filter("C"."CUSTOMER_ID>0)
21
22 Result Cache Information (identified by operation id):
23 -----
24
25 3 - column-count=2; dependencies=(DE.ORDERS); name="select /*+ result_cache q_name(Q2) */ customer_id, count(*) ord_count from orders group by customer_id"

```

## Executing Both Queries

```
--- query3.sql
--- Cache result of both queries, then use the cached result.
Set timing on
set echo on
select /*+ result_cache q_name(Q1) */ * from orders;
select c.customer_id, o.ord_count
  from (select /*+ result_cache q_name(Q3) */
            customer_id, count(*) ord_count
       from orders
      group by customer_id) o, customers c
     where o.customer_id = c.customer_id;
```

```
Script Output X | Query Result X | Query Result 1 X
SQL> set timing on
SQL> set echo on
SQL> select /*+ result_cache q_name(Q1) */ * from orders
/
>>Query Run In:Query Result
Elapsed: 00:00:00.250
SQL> select c.customer_id, o.ord_count
  from (select /*+ result_cache q_name(Q3) */
            customer_id, count(*) ord_count
       from orders
      group by customer_id) o, customers c
     where o.customer_id = c.customer_id
/
>>Query Run In:Query Result 1
Elapsed: 00:00:00.251
SQL> set echo off
```

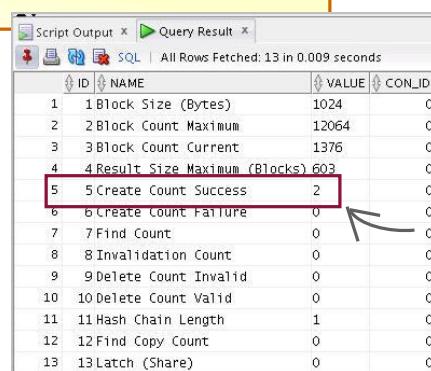


Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You have examined the execution plan for both the queries. Now execute both the queries by executing the code shown in the slide.

## Viewing Cache Results Created

```
col name format a55
select * from v$result_cache_statistics
/
```



ID	NAME	VALUE	CON_ID
1	1 Block Size (Bytes)	1024	0
2	2 Block Count Maximum	12064	0
3	3 Block Count Current	1376	0
4	4 Result Size Maximum (Blocks)	603	0
5	5 Create Count Success	2	0
6	6 Create Count Failure	0	0
7	7 Find Count	0	0
8	8 Invalidation Count	0	0
9	9 Delete Count Invalid	0	0
10	10 Delete Count Valid	0	0
11	11 Hash Chain Length	1	0
12	12 Find Copy Count	0	0
13	13 Latch (Share)	0	0

Number of cache results successfully created



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

The next step is to run a query against V\$RESULT\_CACHE\_STATISTICS to view the memory allocation and usage statistics. You can view the memory allocation by executing the code as shown in the slide.

Note that the CREATE COUNT has the number of cache results that were successfully created (one for each query statement).

In the next steps, you re-execute the queries and view the cache results found.

## Viewing Cache Results Found

- Re-execute the same query and check the cache statistics

```
col name format a55
select * from v$result_cache_statistics
/
```

ID	NAME	VALUE	CON_ID
1	1 Block Size (Bytes)	1024	0
2	2 Block Count Maximum	12064	0
3	3 Block Count Current	1376	0
4	4 Result Size Maximum (Blocks)	603	0
5	5 Create Count Success	3	0
6	6 Create Count Failure	0	0
7	7 Find Count	1	0
8	8 Invalidation Count	0	0
9	9 Delete Count Invalid	0	0
10	10 Delete Count Valid	0	0
11	11 Hash Chain Length	1	0
12	12 Find Copy Count	1	0
13	13 Latch (Share)	0	0

Successful finds in cache.



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

Query V\$RESULT\_CACHE\_STATISTICS again to view memory allocation and usage statistics. Do this by again executing the code shown in the slide.

The Find Count keeps track of the number of results that were successfully found in the cache.

## Lesson Agenda

- Caching in the Database Instance
- Result Caching
- SQL Result Caching
- PL/SQL Result Caching
- Oracle Database In-Memory

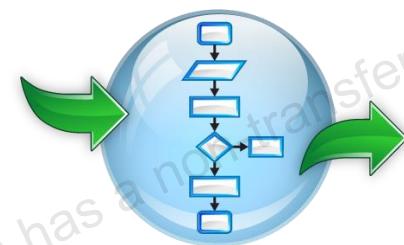


ORACLE®

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

## PL/SQL Function Result Cache

- PL/SQL Function Result Cache stores the results of function execution.
- You can cache a function result by including a RESULT CACHE clause in function definition.
- Good candidates for result caching are frequently invoked functions that depend on relatively static data.



ORACLE®

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

The PL/SQL function result cache is a subset of the server result cache that stores function result sets.

PL/SQL function code can include a request to cache its results. Upon invocation of this function, the system checks the cache. If the cache contains the result from a previous function call with the same parameter values, then the system returns the cached result to the invoker and does not execute the function body again. If the cache does not contain the result, then the system executes the function body and adds the result (for these parameter values) to the cache before returning control to the invoker.

## Marking PL/SQL Function Results to Be Cached

- Scenario:
  - You need a PL/SQL function that derives a complex metric.
  - The data that your function calculates changes slowly, but the function is frequently called.
- Solution:
  - Use the new `RESULT_CACHE` clause in your function definition.
  - You can also have the cache purged when a dependent table experiences a DML operation, by using the `RELIES_ON` clause.



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

To enable result caching for a function, use the `RESULT_CACHE` clause in your PL/SQL function. Using this clause results in the following:

- If a result-cached function is called, the system checks the cache.
- If the cache contains the result from a previous call to the function with the same parameter values, the system returns the cached result to the caller and doesn't execute the function body again.
- If the cache doesn't contain the result, the system executes the function body and adds the result (for these parameter values) to the cache before returning control to the caller.

The cache can accumulate many results—one result for every unique combination of parameter values with which each result-cached function was called. If the system needs more memory, it ages out (deletes) one or more cached results.

You can specify the database objects that are used to compute a cached result, so that if any of them is updated, the cached result becomes invalid and must be recomputed.

The best candidates for result caching are functions that are called frequently but depend on information that changes infrequently or never.

## Clearing the Shared Pool and Result Cache

```
--- flush.sql
--- Start with a clean slate. Flush the cache and shared pool.
--- Verify that memory was released.
SET ECHO ON
SET FEEDBACK 1
SET SERVEROUTPUT ON

execute dbms_result_cache.flush
alter system flush shared_pool
/
execute dbms_result_cache.memory_report
```

```
Script Output X | Task completed in 0.119 seconds
System FLUSH altered.
SQL> execute dbms_result_cache.memory_report
PL/SQL procedure successfully completed.

Result Cache Memory Report
[Parameters]
Block Size = 1K bytes
Maximum Cache Size = 12064K bytes (12064 blocks)
Maximum Result Size = 603K bytes (603 blocks)
[Memory]
Total Memory = 9504 bytes [0.001% of the Shared Pool]
... Fixed Memory = 9504 bytes [0.001% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]
SQL> /
```



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

To understand the use of PL/SQL function result caching, clear the shared pool and the result cache again by performing the steps shown in the slide.

## Creating a PL/SQL Function by Using the RESULT\_CACHE Clause

- Include the `RESULT_CACHE` option in the function definition.
- Optionally, include the `RELIES_ON` clause.

```
CREATE OR REPLACE FUNCTION ORD_COUNT(cust_no number)
RETURN NUMBER
RESULT_CACHE RELIES_ON (orders)
IS
  v_count NUMBER;
BEGIN
  SELECT COUNT(*) INTO v_count
  FROM orders
  WHERE customer_id = cust_no;

  return v_count;
end;
```

Specifies that the result should be cached

Specifies the table upon which the function has a dependency



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

When writing code for the PL/SQL result cache option, you need to:

- Include the `RESULT_CACHE` option in the function declaration section of a package
- Include the `RESULT_CACHE` option in the function definition
- Optionally, include the `RELIES_ON` clause to specify tables or views on which the function results depend

In the example shown in the slide, the `ORD_COUNT` function has result caching enabled through the `RESULT_CACHE` option in the function declaration. In this example, the `RELIES_ON` clause is used to identify the `ORDERS` table on which the function results depend.

You can also run `DBMS_RESULT_CACHE.MEMORY_REPORT` to view the result cache memory results.

## Calling the PL/SQL Function Inside a Query

```
select cust_last_name, ord_count(customer_id) no_of_orders  
  from customers  
 where cust_last_name = 'MacGraw'
```

CUST_LAST_NAME	NO_OF_ORDERS
MacGraw	3



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

Call the ORD\_COUNT PL/SQL function inside a query by performing the query shown in the slide. While defining the function you have defined that the results of function evaluation be cached. The result of the function invoked with a customer\_id of a customer whose last name is 'MacGraw' is stored in the cache.

## Viewing Cache Results Created

```
col name format a55
select * from v$result_cache_statistics
/
```

The screenshot shows the Oracle SQL Developer interface with two tabs: 'Script Output' and 'Query Result'. The 'Query Result' tab is active, displaying the results of the query. The results are presented in a table with four columns: ID, NAME, VALUE, and CON\_ID. The table contains 13 rows, each representing a different cache statistic. Row 5, which is 'Create Count Success', has a value of 1 and is highlighted with a red border.

ID	NAME	VALUE	CON_ID
1	Block Size (Bytes)	1024	0
2	Block Count Maximum	12064	0
3	Block Count Current	1376	0
4	Result Size Maximum (Blocks)	603	0
5	Create Count Success	1	0
6	Create Count Failure	0	0
7	Find Count	0	0
8	Invalidation Count	0	0
9	Delete Count Invalid	0	0
10	Delete Count Valid	0	0
11	Hash Chain Length	1	0
12	Find Copy Count	0	0
13	Latch (Share)	0	0



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

You can query V\$RESULT\_CACHE\_STATISTICS to view the memory allocation and usage statistics. Do this by performing the steps shown in the slide.

## Calling the PL/SQL Function Again

```
select cust_last_name, ord_count(customer_id) no_of_orders  
  from customers  
 where cust_last_name = 'MacGraw'
```

CUST_LAST_NAME	NO_OF_ORDERS
MacGraw	3



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

Run the query again, as shown in the slide.

## Viewing Cache Results Found

```
col name format a55
select * from v$result_cache_statistics
/
```

ID	NAME	VALUE	CON_ID
1	1 Block Size (Bytes)	1024	0
2	2 Block Count Maximum	12064	0
3	3 Block Count Current	1376	0
4	4 Result Size Maximum (Blocks)	603	0
5	5 Create Count Success	1	0
6	6 Create Count Failure	0	0
7	7 Find Count	1	0
8	8 Invalidation Count	0	0
9	9 Delete Count Invalid	0	0
10	10 Delete Count Valid	0	0
11	11 Hash Chain Length	1	0
12	12 Find Copy Count	1	0
13	13 Latch (Share)	0	0



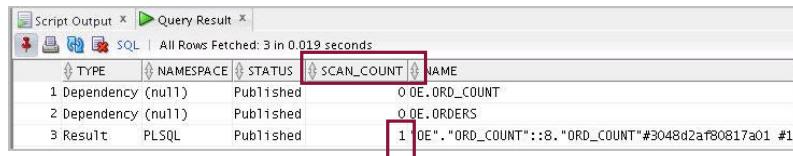
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

Query V\$RESULT\_CACHE\_STATISTICS again to view the memory allocation and usage statistics. Do this by performing the steps shown in the slide.

Note that the FIND COUNT statistic now has a value of 1. This is the number of cache results that were successfully found (one for each query statement).

## Confirming That the Cached Result Was Used

```
select type, namespace,status, scan_count,name  
from v$result_cache_objects  
/
```



TYPE	NAMESPACE	STATUS	SCAN_COUNT	NAME
1 Dependency	(null)	Published	0	OE.ORD_COUNT
2 Dependency	(null)	Published	0	OE.ORDERS
3 Result	PLSQL	Published	1	OE"."ORD_COUNT":>B."ORD_COUNT"#3048d2af80817a01 #1

ORACLE

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

Query V\$RESULT\_CACHE\_OBJECTS to confirm that the cached result was used, by performing the steps shown in the slide. The SCAN\_COUNT column indicates the number of times the object was accessed.

## Lesson Agenda

- Caching in the Database Instance
- Result Caching
- SQL Result Caching
- PL/SQL Result Caching
- Oracle Database In-Memory



ORACLE®

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

## Oracle Database In-Memory

- A dual format architecture of 12c supports both the traditional row format and in-memory column format.
- An In-Memory column store is maintained in the SGA.
- The column format provides high performance for analytical processing.
- You can enable and configure in-memory column store in your database as a system administrator.



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

Oracle Database 12c has an unique "dual-format" architecture that enables data to be maintained in both the existing Oracle row format, for OLTP operations, and a new purely in-memory columnar format, optimized for analytical processing.

Data is traditionally stored in a row format. In a row format database, each new transaction or record stored in the database is represented as a new row in a table. That row is made up of multiple columns, with each column representing a different attribute about that record. A row format is ideal for online transaction systems, as it allows quick access to all of the columns in a record since all of the data for a given record are kept together in-memory and on-storage.

A column format database stores each of the attributes about a transaction or record in a separate column structure. A column format is ideal for analytics, as it allows for faster data retrieval when only a few columns are selected but the query accesses a large portion of the data set.

Oracle Database In-Memory (Database In-Memory) provides the best of both worlds by allowing data to be simultaneously populated in both an in-memory row format (the buffer cache) and a new in-memory column format: a dual-format architecture.

Note that the dual-format architecture does not double memory requirements.

You can learn more on In-Memory column store here:

<http://www.oracle.com/technetwork/database/in-memory/overview/twp-oracle-database-in-memory-2245633.html>

## Quiz



Which of the following statements are true?

- a. When a query is executed, the result cache is built up in the result cache memory.
- b. Subsequent executions of the same query or function can be served directly out of the cache, improving response times.
- c. This technique should not be used for SQL queries and PL/SQL functions that are executed frequently.
- d. Cached query results remains valid even after the database data accessed by the query is modified.

ORACLE

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



**Answer: a, b**

## Quiz



You can set the RESULT\_CACHE\_MODE to FORCE at the session level by using the ALTER SESSION command, so that the results of all the queries are always stored in the result cache memory.

- a. True
- b. False

ORACLE

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

**Answer: a**

## Quiz



You can use the `DBMS_RESULT_CACHE` package to:

- a. Bypass the cache
- b. Retrieve statistics on the cache memory usage
- c. Flush the cache
- d. None of the above

ORACLE

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

**Answer: a, b, c**

## Quiz



On querying V\$RESULT\_CACHE\_STATISTICS to view the memory allocation and usage statistics, the number of cache results successfully **found** is denoted by:

- a. The CREATE COUNT SUCCESS statistic
- b. The FIND COUNT statistic
- c. The INVALIDATION COUNT statistic
- d. The HASH CHAIN LENGTH statistic

ORACLE

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

**Answer: b**

## Quiz



You can create a function successfully that has both invoker's rights and is result cached.

- a. True
- b. False

ORACLE

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

### Answer: a

It is possible to create a unit that has both invoker's rights (declared AUTHID CURRENT USER) and is result cached (with the declaration RESULT CACHE).

## Summary

In this lesson, you should have learned how to:

- Improve memory usage by caching SQL result sets
- Write queries that use the result cache hint
- Use the DBMS\_RESULT\_CACHE package
- Set up PL/SQL functions to use PL/SQL result caching



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

In this lesson, you saw the Oracle Database 12c caching techniques that can improve performance.

## Practice 9: Overview

This practice covers the following topics:

- Writing code to use SQL caching
- Writing code to use PL/SQL caching



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

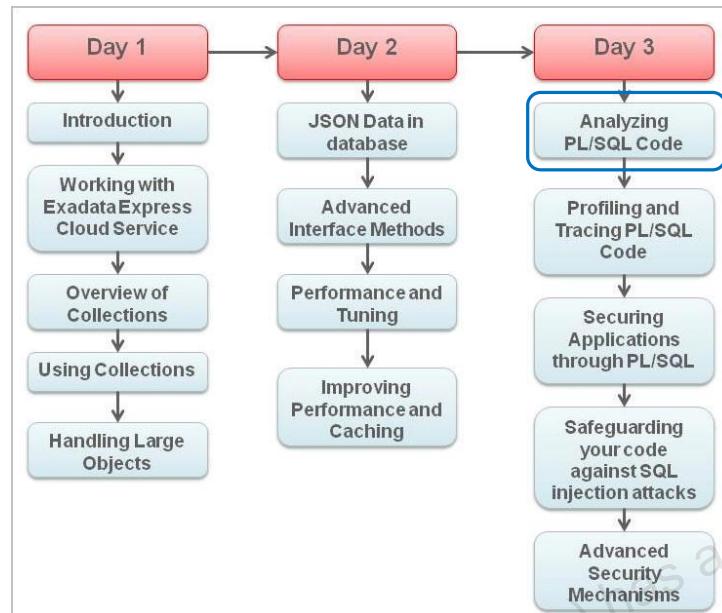
In this practice, you implement SQL query result caching and PL/SQL result function caching. You run scripts to measure the cache memory values, manipulate queries and functions to turn caching on and off, and then examine cache statistics. This practice uses the OE schema.

# Analyzing PL/SQL Code

ORACLE

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

## Course Agenda



ORACLE

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



## Objectives

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

- Use the supplied packages and dictionary views to find coding information
- Determine identifier types and usages with PL/Scope
- Use the `DBMS_METADATA` package



ORACLE

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

In this lesson, you learn how to write PL/SQL routines that analyze PL/SQL applications. You will learn about the dictionary views and packages that you can use to generate information about your code.

## Lesson Agenda

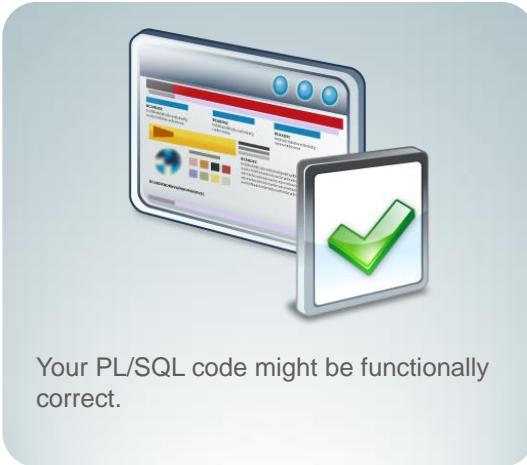
- Data dictionary views
- PL/Scope
- Oracle Supplied Packages for Code Analysis



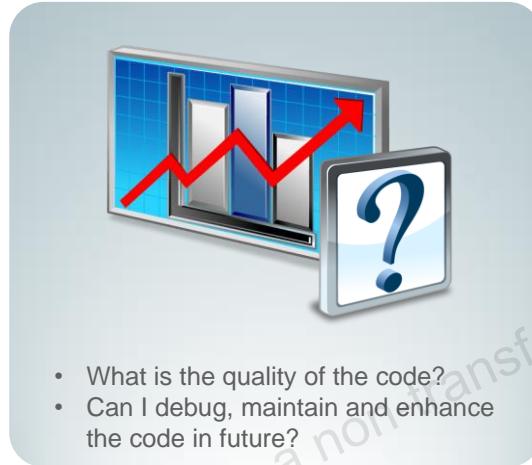
ORACLE®

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

## PL/SQL Code Analysis



Your PL/SQL code might be functionally correct.



- What is the quality of the code?
- Can I debug, maintain and enhance the code in future?

ORACLE®

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

Under the pressure of deadlines and meeting targets, developers often end up producing suboptimal code. The code might be functionally right, but how about long term maintenance of the code? Can you debug, maintain and successfully enhance the code in future? These factors are essential for every application.

Code analysis is essential to ascertain these facts about the PL/SQL code you write. Oracle provides various data dictionary views, tools and PL/SQL packages which enable you to analyze and understand code.

## PL/SQL Code Analysis

- You perform code analysis in PL/SQL to understand:
  - Variable usage
  - Object dependencies
  - The program execution flow
  - The performance profile of the application
- Oracle provides various tools and data dictionary views for efficient code analysis.



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

Oracle database provides various data dictionary views which enable developers to understand their PL/SQL code. You can access information about PL/SQL program units through these data dictionary views.

Oracle 11g has introduced a powerful analytical tool PL/Scope to better understand the application code base.

## Data Dictionary Views

- Oracle provides data dictionary views which you can use for PL/SQL code analysis
  - ALL\_SOURCE
  - ALL\_ARGUMENTS
  - ALL PROCEDURES
  - ALL\_DEPENDENCIES
- There are USER\_\* and DBA\_\* variants of these data dictionary views.



**ORACLE**

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

The Oracle dictionary views store information about your compiled PL/SQL code. You can write SQL statements against the views to find information about your code.

Dictionary View	Description
ALL_SOURCE	Includes the lines of source code for all programs you modify
ALL_ARGUMENTS	Includes information about the parameters for the procedures and functions that you can call
ALL PROCEDURES	Contains the list of procedures and functions that you can execute
ALL_DEPENDENCIES	Is one of the several views that give you information about the dependencies between database objects

The USER\_\* variants are USER\_SOURCE, USER\_ARGUMENTS, USER PROCEDURES and USER\_DEPENDENCIES. These views contain objects that are owned by an user.

The DBA\_\* variants are DBA\_SOURCE, DBA\_ARGUMENTS, DBA PROCEDURES and DBA\_DEPENDENCIES. These views has information of all the objects owned by the SYS user or by an user with DBA privileges.

## Analyzing PL/SQL Code

Find all instances of CHAR in your code:

```
SELECT NAME, line, text
FROM      user_source
WHERE     INSTR (UPPER(text), ' CHAR') > 0
          OR INSTR (UPPER(text), ' CHAR(') > 0
          OR INSTR (UPPER(text), ' CHAR ()) > 0;
```

Query Result		
NAME	LINE	TEXT
1 ACTION_T	1 TYPE	"ACTION_T" AS OBJECT ("SYS_XDBPD\$" "XDB"."
2 LINEITEM_T	1 TYPE	"LINEITEM_T" AS OBJECT ("SYS_XDBPD\$" "XDB"
3 PART_T	1 TYPE	"PART_T" AS OBJECT ("SYS_XDBPD\$" "XDB"."XDI
4 PURCHASEORDER_T	1 TYPE	"PURCHASEORDER_T" AS OBJECT ("SYS_XDBPD\$"
5 REJECTION_T	1 TYPE	"REJECTION_T" AS OBJECT ("SYS_XDBPD\$" "XDB
6 SHIPPING_INSTRUCTIONS_T	1 TYPE	"SHIPPING_INSTRUCTIONS_T" AS OBJECT ("SYS_
7 CUST_ADDRESS_TYP	8 , country_id	CHAR(2)



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

Consider a scenario where you want to find all occurrences of the CHAR data type. The CHAR data type is fixed in length and can cause false negatives on comparisons with VARCHAR2 strings.

You might want to locate the possibility of such occurrences in your code. The code in the slide is using the USER\_SOURCE data dictionary view to locate all the occurrences of CHAR string in your code.

## Analyzing PL/SQL Code

Let's create a package query\_code\_pkg, we will analyze this package through the data dictionary views:

```
CREATE OR REPLACE PACKAGE query_code_pkg
AUTHID CURRENT_USER
IS
  PROCEDURE find_text_in_code (str IN VARCHAR2);
  PROCEDURE encaps_compliance ;
END query_code_pkg;
/
```



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

QUERY\_CODE\_PKG has two validation procedures:

- **FIND\_TEXT\_IN\_CODE:** Displays all programs with a specified character string. It queries USER\_SOURCE to find occurrences of any text string. The text string is passed as a parameter. For efficiency, the BULK COLLECT statement is used to retrieve all matching rows into the collection variable.
- **ENCAP\_COMPLIANCE:** Identifies those programs that reference a table directly. This procedure queries the ALL\_DEPENDENCIES view to find the PL/SQL code objects that directly reference a table or a view.

```
CREATE OR REPLACE PACKAGE query_code_pkg
AUTHID CURRENT_USER
IS
  PROCEDURE find_text_in_code (str IN VARCHAR2);
  PROCEDURE encaps_compliance ;
END query_code_pkg;
/
```

**QUERY\_CODE\_PKG Code**

```

CREATE OR REPLACE PACKAGE BODY query_code_pkg IS
  PROCEDURE find_text_in_code (str IN VARCHAR2)
  IS
    TYPE info_rt IS RECORD (NAME user_source.NAME%TYPE,
                           text user_source.text%TYPE );
    TYPE info_aat IS TABLE OF info_rt INDEX BY PLS_INTEGER;
    info_aa info_aat;
  BEGIN
    SELECT NAME || '-' || line, text
    BULK COLLECT INTO info_aa FROM user_source
      WHERE UPPER (text) LIKE '%' || UPPER (str) || '%'
      AND NAME != 'VALSTD' AND NAME != 'ERRNUMS';
    DBMS_OUTPUT.PUT_LINE ('Checking for presence of ''|
                           str || ':');
    FOR indx IN info_aa.FIRST .. info_aa.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
        info_aa (indx).NAME || ',' || info_aa (indx).text);
    END LOOP;
  END find_text_in_code;

  PROCEDURE encap_compliance IS
    SUBTYPE qualified_name_t IS VARCHAR2 (200);
    TYPE refby_rt IS RECORD (NAME qualified_name_t,
                             referenced_by qualified_name_t );
    TYPE refby_aat IS TABLE OF refby_rt INDEX BY PLS_INTEGER;
    refby_aa refby_aat;
  BEGIN
    SELECT owner || '.' || NAME refs_table
      , referenced_owner || '.' || referenced_name
      AS table_referenced
    BULK COLLECT INTO refby_aa
      FROM all_dependencies
      WHERE owner = USER
      AND TYPE IN ('PACKAGE', 'PACKAGE BODY',
                   'PROCEDURE', 'FUNCTION')
      AND referenced_type IN ('TABLE', 'VIEW')
      AND referenced_owner NOT IN ('SYS', 'SYSTEM')
    ORDER BY owner, NAME, referenced_owner, referenced_name;
    DBMS_OUTPUT.PUT_LINE ('Programs that reference
                           tables or views');
    FOR indx IN refby_aa.FIRST .. refby_aa.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (refby_aa (indx).NAME || ',' ||
                            refby_aa (indx).referenced_by);
    END LOOP;
  END encap_compliance;
END query_code_pkg;
/

```

## Analyzing PL/SQL Code

```
EXECUTE query_code_pkg.encap_compliance
```

The screenshot shows the Oracle SQL Developer interface. A yellow box highlights the command `EXECUTE query_code_pkg.encap_compliance`. Below it, the **Script Output** window displays the results of the execution. The output window title is **Script Output**, and it shows a timestamp: **Task completed in 0.304 seconds**. The main content of the output window is:  
PL/SQL procedure successfully completed.  
  
Programs that reference tables or views  
OE.ADD\_ORDER\_ITEMS,OE.PORDER  
OE.ALLOCATE\_NEW\_PROJ\_LIST,HR.DEPARTMENTS  
OE.CHANGE\_CREDIT,OE.CUSTOMERS  
OE.CREDIT\_CARD\_PKG,OE.CUSTOMERS  
OE.MANAGE\_DEPT\_PROJ,OE.DEPARTMENT  
OE.ORD\_COUNT,OE.ORDERS  
OE.PROCESS\_CUSTOMERS,OE.CUSTOMERS  
OE.REPORT\_CREDIT,OE.CUSTOMERS



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

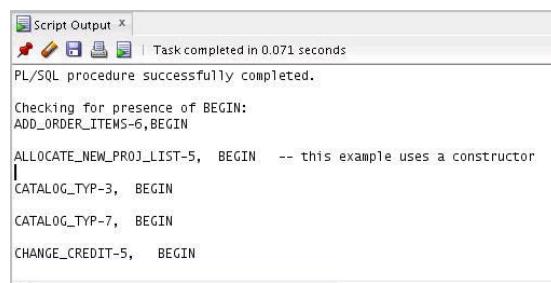
### QUERY\_CODE\_PKG: Examples

In the first example in the slide, the `ENCAP_COMPLIANCE` procedure displays all PL/SQL code objects that reference a table or view directly. Both the object name and table or view name are listed in the output.

**Note:** The output screenshot may slightly vary.

## Analyzing PL/SQL Code

```
EXECUTE query_code_pkg.find_text_in_code('BEGIN')
```



The screenshot shows the Oracle SQL Developer interface with a "Script Output" window. The window title is "Script Output X". It displays the output of the executed PL/SQL procedure. The output message says "PL/SQL procedure successfully completed." followed by the results of the search for "BEGIN". The results list several code objects: ADD\_ORDER\_ITEMS-6, BEGIN; ALLOCATE\_NEW\_PROJ\_LIST-5, BEGIN -- this example uses a constructor; CATALOG\_TYP-3, BEGIN; CATALOG\_TYP-7, BEGIN; and CHANGE\_CREDIT-5, BEGIN. The entire output is contained within a light yellow box.

Note: The output image has partial result of the query.

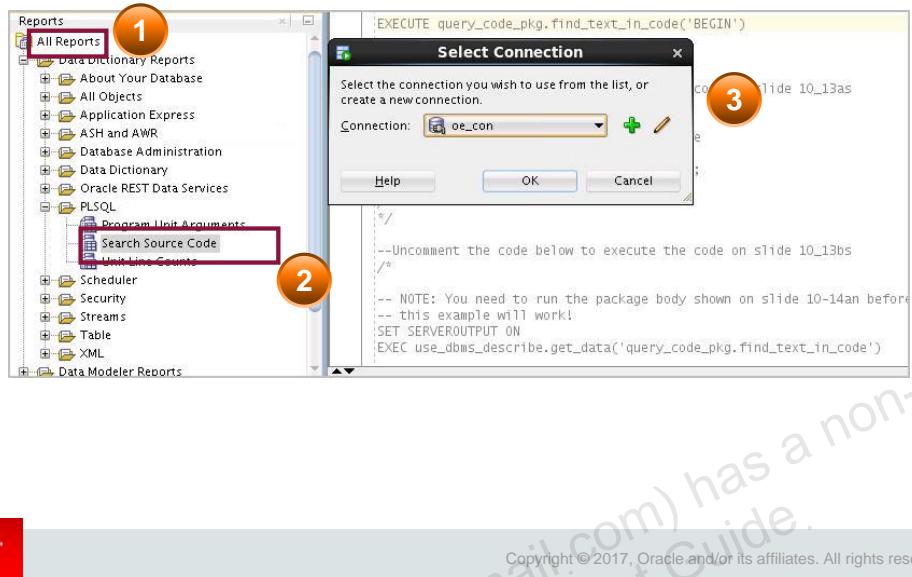


Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

In the second example, the FIND\_TEXT\_IN\_CODE procedure returns all PL/SQL code objects that contain the “BEGIN” text string. The code name, line number, and line are listed in the output.

## Using SQL Developer for Code Analysis

Use Reports:



SQL Developer comes with predefined reports that you can use to find information about PL/SQL coding.

If you want to find the occurrence of a text string or an object name, use the Reports feature.

1. Select the Reports tabbed page in SQL Developer.
2. Expand the PL/SQL node and select the Search Source Code option.
3. Enter the connection information.

## Using SQL Developer for Code Analysis

Enter the text you want to search

The screenshot shows the Oracle SQL Developer interface. On the left, a dialog box titled 'Enter Bind Values' is open, showing a 'Text Search' entry in the 'PL/SQL Object Name' dropdown and 'BEGIN' in the 'Value' field. A large orange circle labeled '4' is positioned above the dialog. On the right, a table titled 'Search Source Code' displays search results for 'new\_code\_ex\_10.sql'. The table has columns: Owner, PL/SQL Object Name, Type, Line, and Text. The results show various PL/SQL objects like ADD\_ORDER\_ITEMS, ALLOCATE\_NEW\_PROJ\_LIST, etc., with their respective line numbers and code snippets. A large orange circle labeled '5' is positioned below the table. At the bottom of the interface, there is an 'ORACLE' logo and a copyright notice: 'Copyright © 2017, Oracle and/or its affiliates. All rights reserved.'

Owner	PL/SQL Object Name	Type	Line	Text
1 OE	ADD_ORDER_ITEMS	PROCEDURE	6	BEGIN
2 OE	ALLOCATE_NEW_PROJ_LIST	PROCEDURE	5	BEGIN -- this example uses a constructor
3 OE	CATALOG_TYP	TYPE BODY	3	BEGIN
4 OE	CATALOG_TYP	TYPE BODY	7	BEGIN
5 OE	CHANGE_CREDIT	PROCEDURE	5	BEGIN
6 OE	COMPOSITE_CATEGORY_TYP	TYPE BODY	3	BEGIN
7 OE	CREDIT_CARD_PKG	PACKAGE BODY	9	BEGIN
8 OE	CREDIT_CARD_PKG	PACKAGE BODY	30	BEGIN
9 OE	CREDIT_CARD_PKG	PACKAGE BODY	55	BEGIN
10 OE	C_OUTPUT	PROCEDURE	6	BEGIN

Note: The output in the image is partial output

4. Select either an Object Name or a Text Search for the search type. In the Value field, enter the text string that you want to search in your PL/SQL source code for the connection that you specified in step 3.
5. View the results of your search.

## Using ALL\_ARGUMENTS

Query the ALL\_ARGUMENTS view to find information about arguments for procedures and functions:

```
SELECT object_name, argument_name, in_out, position, data_type
FROM   all_arguments
WHERE  package_name = 'CREDIT_CARD_PKG';
```

OBJECT_NAME	ARGUMENT_NAME	IN_OUT	POSITION	DATA_TYPE
1 UPDATE_CARD_INFO	P_CUST_ID	IN	1	NUMBER
2 UPDATE_CARD_INFO	P_CARD_TYPE	IN	2	VARCHAR2
3 UPDATE_CARD_INFO	P_CARD_NO	IN	3	VARCHAR2
4 DISPLAY_CARD_INFO	P_CUST_ID	IN	1	NUMBER



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

You can also query the ALL\_ARGUMENTS dictionary view to find information about the arguments of procedures and functions to which you have access.

In the example shown in the slide, the argument name, mode, position, and data type are returned for CREDIT\_CARD\_PKG.

As shown in the slide, ALL\_ARGUMENTS view has information about many attributes of the arguments. In the example we choose only object\_name, argument\_name, IN\_OUT mode, position and data\_type.

```

CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2);
    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

```

```

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type,
p_card_no);
            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;

```

```
Using ALL_ARGUMENTS (Notes Only) (continued)
PROCEDURE display_card_info
    (p_cust_id NUMBER)
IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
BEGIN
    SELECT credit_cards
        INTO v_card_info
        FROM customers
        WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
        FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
            DBMS_OUTPUT.PUT('Card Type: ' ||
v_card_info(idx).card_type
                           || ' ');
            DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
v_card_info(idx).card_num );
        END LOOP;
    ELSE
        DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
    END display_card_info;
END credit_card_pkg; -- package body
/
```

## ALL\_ARGUMENTS

Here is the list of all the attributes of **ALL\_ARGUMENTS** view .

You can query the view as per your requirement.

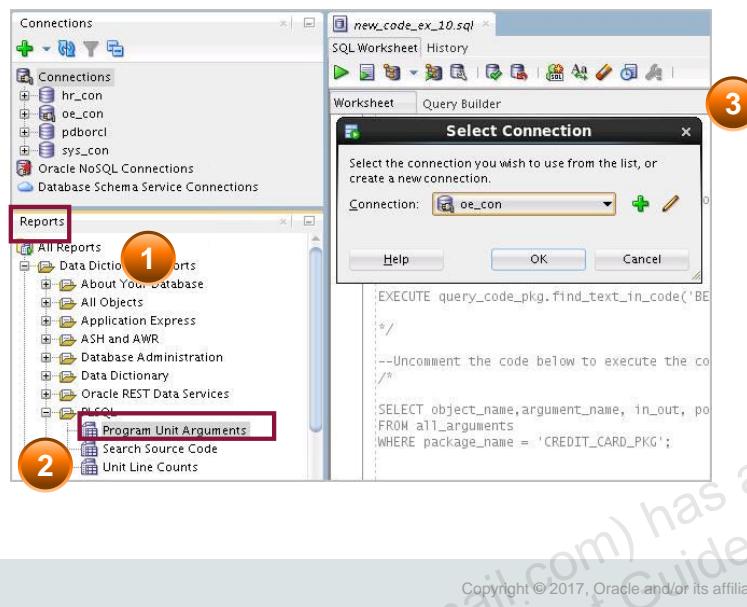
Name	Null	Type
OWNER	NOT NULL	VARCHAR2(128)
OBJECT_NAME		VARCHAR2(128)
PACKAGE_NAME		VARCHAR2(128)
OBJECT_ID	NOT NULL	NUMBER
OVERLOAD		VARCHAR2(40)
SUBPROGRAM_ID		NUMBER
ARGUMENT_NAME		VARCHAR2(128)
POSITION	NOT NULL	NUMBER
SEQUENCE	NOT NULL	NUMBER
DATA_LEVEL	NOT NULL	NUMBER
DATA_TYPE		VARCHAR2(30)
DEFAULTED		VARCHAR2(1)
DEFAULT_VALUE		LONG
DEFAULT_LENGTH		NUMBER
IN_OUT		VARCHAR2(9)
DATA_LENGTH		NUMBER
DATA_PRECISION		NUMBER
DATA_SCALE		NUMBER
RADIX		NUMBER
CHARACTER_SET_NAME		VARCHAR2(44)
TYPE_OWNER		VARCHAR2(128)
TYPE_NAME		VARCHAR2(128)
TYPE_SUBNAME		VARCHAR2(128)
TYPE_LINK		VARCHAR2(128)
PLS_TYPE		VARCHAR2(128)
CHAR_LENGTH		NUMBER
CHAR_USED		VARCHAR2(1)
ORIGIN_CON_ID		NUMBER



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

## Using SQL Developer to Report on Arguments

Use Reports:

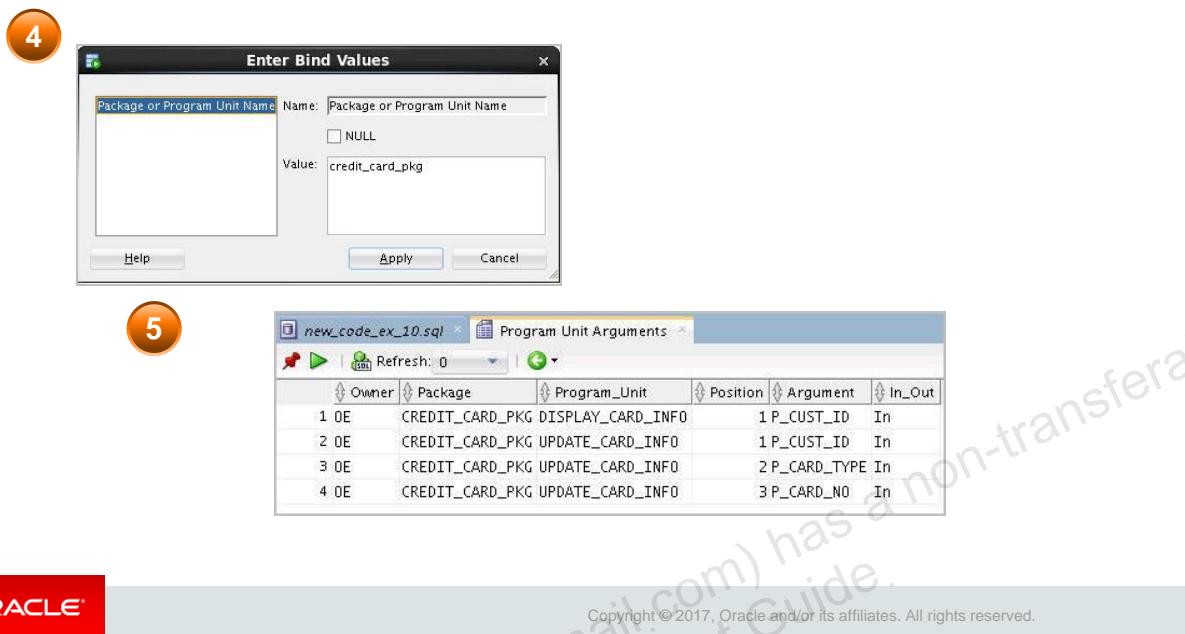


SQL Developer comes with predefined reports that you can use to find PL/SQL coding information about your program unit arguments.

Use the Reports feature to find information about program unit arguments.

1. Select the Reports tabbed page in SQL Developer.
2. Expand the PL/SQL node and select Program Unit Arguments.
3. When prompted, enter the connection information.

## Using SQL Developer to Report on Arguments



4. Enter the name of the program unit on which you want the arguments reported.
5. View the results of your search.

In the example shown in the slide, the output displays the owner of the stored PL/SQL package, the name of the PL/SQL package, the names of the subroutines within the PL/SQL package, the position of the arguments within the package, the argument names, and the argument types (IN, OUT, or IN OUT).

## Lesson Agenda

- Running reports on source code
- PL/Scope
- Oracle Supplied Packages for Code Analysis



ORACLE®

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

## PL/Scope

- Where and how a column x in table y is used in the PL/SQL code?
- What are the constants, variables and exceptions in my application that are declared but never used?
- Is my code at risk for SQL injection?
- What are the SQL statements with an optimizer hint coded in the application?
- Which SQL has a BULK COLLECT clause ? Where is the SQL called from ?

PL/Scope is a compiler driven tool which enables you to analyze SQL and PL/SQL code and answer such questions



ORACLE

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

You have the data dictionary views to provide information about PL/SQL code, arguments and so on. PL/Scope provides information on the 'how' aspect of SQL and PL/SQL code.

PL/Scope is a compiler-driven tool that collects PL/SQL and SQL identifiers as well as SQL statements usage in PL/SQL source code.

Starting with Oracle Database 12c Release 2 (12.2), PL/Scope has been enhanced to report on the occurrences of static SQL, and dynamic SQL call sites in PL/SQL units. PL/Scope provides insight into dependencies between tables, views and the PL/SQL units. This level of details can be used as a migration assessment tool to determine the extent of changes required.

## PL/Scope

- Is a tool that is used for extracting, organizing, and storing user-supplied identifiers from PL/SQL source code.
- Works with the PL/SQL compiler.
- PL/Scope collects metadata on PL/SQL identifiers, SQL identifiers, SQL statements during program compilation.
- The metadata collected is made available as views
- In 12.2, PL/Scope is enhanced to report on occurrences of static SQL and dynamic SQL call sites in PL/SQL units.
- PL/Scope provides insight into dependencies between tables, views and PL/SQL units.



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

With PL/Scope, you can produce a cross-referenced repository of PL/SQL identifiers to gather information about your PL/SQL applications.

With PL/Scope, you can collect the identifier types, usages (declaration, definition, reference, call, assignment) and location of usage.

With Oracle Database 12cR2, you can analyze SQL statements within PL/SQL code. This enables developers to perform impact analysis of changes to tables such as changing the name of the column in a table.

12cR2 comes with a support to analysis of SQL statements. Post 12cR2 PL/Scope can :

- Show where specific columns are referenced.
- Identify PL/SQL program units which are performing certain DML operations.
- Locate all SQL statements containing hints.
- Find all dynamic SQL call sites in PL/SQL to eliminate the possibility of SQL injection.
- Find all the locations in the code where you have performed a commit or a rollback operation.

## Using PL/Scope

- Set the PL/SQL compilation parameter `PLSCOPE_SETTINGS`.
- Valid values for `IDENTIFIERS`:
  - `ALL` – Collect all PL/SQL identifier actions found in compiled source.
  - `NONE` – Do not collect any identifier actions (the default).
- You can enable PL/Scope for a session, system, or library unit :
  - `ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS: ALL'`
  - `ALTER SYSTEM SET PLSCOPE_SETTINGS = 'IDENTIFIERS: ALL'`
  - `ALTER functionname COMPILE PLSCOPE_SETTINGS = 'IDENTIFIERS: ALL'`
- The `USER/ALL/DBA_IDENTIFIERS` catalog view holds the collected identifier values.



ORACLE

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

To use PL/Scope, you need to set the PL/SQL compilation parameter `PLSCOPE_SETTINGS` to '`IDENTIFIERS:ALL`'. You can use the `ALTER SESSION`, `ALTER SYSTEM`, or `ALTER COMPILE` statement to set this parameter. After the parameter is set, any code that you compile is analyzed for PL/Scope. The information collected is gathered in the `USER/ALL/DBA_IDENTIFIERS` dictionary view.

The identifier action describes how the identifier is used, such as in a declaration, definition, reference, assignment, or call.

Query the `USER|ALL|DBA_PLSQL_OBJECT_SETTINGS` views to view what `PLSCOPE_SETTINGS` are set to. This view contains a column called `PLSCOPE_SETTINGS`. When you query this column, by default, all objects are set to '`IDENTIFIERS:NONE`', unless you enable the PL/SQL compilation parameter and recompile the code.

## USER\_IDENTIFIER View

```
ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS: ALL';

DESCRIBE USER_IDENTIFIER;
```

Name	Null	Type
NAME		VARCHAR2(128)
SIGNATURE		VARCHAR2(32)
TYPE		VARCHAR2(18)
OBJECT_NAME	NOT NULL	VARCHAR2(128)
OBJECT_TYPE		VARCHAR2(12)
USAGE		VARCHAR2(11)
USAGE_ID		NUMBER
LINE		NUMBER
COL		NUMBER
USAGE_CONTEXT_ID		NUMBER
ORIGIN_CON_ID		NUMBER



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

The USER/ALL/DBA\_IDENTIFIER dictionary view is used to collect information about your identifiers for any code that is compiled or altered when the PL/SQL compilation parameter is set to PLSCOPE\_SETTINGS = 'IDENTIFIERS: ALL'.

Column	Description
OWNER	Owner of the identifier
NAME	Name of the identifier (may not be unique)
SIGNATURE	A unique signature for this identifier
TYPE	The type of the identifier, such as variable, formal, varray
OBJECT_NAME	The object name where the action occurred
OBJECT_TYPE	The type of object where the action occurred
USAGE	The action performed on the identifier, such as declaration, definition, reference, assignment, or call
USAGE_ID	The unique key for the identifier usage
LINE	The line where the identifier action occurred
COL	The column where the identifier action occurred
USAGE_CONTEXT_ID	The context USAGE_ID of the identifier action

## Sample Data for PL/Scope

Here is a sample package to use with PL/Scope for analysis:

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
    ...
  END update_card_info;
  PROCEDURE display_card_info
    (p_cust_id NUMBER)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
    ...
  END display_card_info;
END credit_card_pkg; -- package body
```



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

CREDIT\_CARD\_PKG created earlier is used for PL/SQL scoping.

The code is a simple package that contains one type, two procedures, and one function. Identifier information will be collected on this package as shown on the following pages.

## Collecting Information on Identifiers

```
ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';

ALTER PACKAGE credit_card_pkg COMPILE;
```

Identifier information is collected in the `USER_IDENTIFIER`s dictionary view, where you can:

- Perform a basic identifier search
- Use contexts to describe identifiers
- Find identifier actions
- Describe identifier actions



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

By default, PL/Scope does not collect data for identifiers and statements in the PL/SQL source program. To enable and control what is collected, set the PL/SQL compilation parameter `PLSCOPE_SETTINGS`.

Starting with Oracle Database 12c Release 2 (12.2), the `PLSCOPE_SETTINGS` has a new syntax that offers more controls and options to collect identifiers and SQL statements metadata. The metadata is collected in the static data dictionary views `DBA_IDENTIFIERS` and `DBA_STATEMENTS`. The possible values for the `IDENTIFIERS` clause are : ALL, NONE (default), PUBLIC, SQL, and PLSQL.

In the example shown in the slide, the PL/SQL compilation parameter `PLSCOPE_SETTINGS` is enabled to collect information about all identifiers. `CREDIT_CARD_PKG` is then recompiled. After it is recompiled, the identifier information is available in the `ALL|USER|DBA_IDENTIFIER` views.

To verify that you have enabled identifier information to be collected on the `CREDIT_CARD_PKG` package, issue the following statement:

```
SELECT PLSCOPE_SETTINGS
  FROM USER_PLSQL_OBJECT_SETTINGS
 WHERE NAME='CREDIT_CARD_PKG' AND TYPE='PACKAGE BODY';

PLSCOPE_SETTINGS
-----
IDENTIFIERS:ALL
```

## Viewing Identifier Information

Create a report based on the data in the `USER_IDENTIFIER`s view:

```
WITH v AS
  (SELECT Line,
   Col,
   INITCAP(NAME) Name,
   LOWER(TYPE) Type,
   LOWER(USAGE) Usage,
   USAGE_ID, USAGE_CONTEXT_ID
    FROM USER_IDENTIFIER
   WHERE Object_Name = 'CREDIT_CARD_PKG'
     AND Object_Type = 'PACKAGE BODY' )
  SELECT RPAD(LPAD(' ', 2*(Level-1)) ||
              Name, 20, '.')||' '||
         RPAD(Type, 20)|| RPAD(Usage, 20)
          IDENTIFIER_USAGE_CONTEXTS
   FROM v
  START WITH USAGE_CONTEXT_ID = 0
 CONNECT BY PRIOR USAGE_ID = USAGE_CONTEXT_ID
 ORDER SIBLINGS BY Line, Col;
```



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

The sample SQL statement shown in the slide retrieves and formats the identifier information that is collected for the `CREDIT_CARD_PKG` package body. Note that the inline view retrieves values from the Name, Type, and Usage columns in the `USER_IDENTIFIER`s dictionary view.

Apart from the inline select query, the code example has code for formatting and viewing data.

## Viewing Identifier Information

Results:

IDENTIFIER_USAGE_CONTEXTS		
1	Credit_Card_Pkg..... package	definition
2	Update_Card_Info... procedure	definition
3	P_Cust_Id..... formal in	declaration
4	Number..... number datatype	reference
5	P_Card_Type..... formal in	declaration
6	Varchar2..... character datatype	reference
7	P_Card_No..... formal in	declaration
8	Varchar2..... character datatype	reference
9	V_Card_Info..... variable	declaration
10	Typ_Cr_Card_Ns nested table	reference
11	I..... variable	declaration
12	Integer..... subtype	reference
13	V_Card_Info..... variable	reference
14	I..... variable	assignment
15	V_Card_Info... variable	reference
16	V_Card_Info..... variable	reference
17	V_Card_Info..... variable	assignment
18	I..... variable	reference

ORACLE

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

The results are formatted and display the name of the identifier, type, and how the identifier is used.

## Performing a Basic Identifier Search

Display the unique identifiers in your schema by querying for all 'DECLARATION' identifier actions:

```
SELECT NAME, SIGNATURE, TYPE
FROM USER_IDENTIFIERS
WHERE USAGE='DECLARATION'
ORDER BY OBJECT_TYPE, USAGE_ID;
```

NAME	SIGNATURE	TYPE
1 CALL_C	C14F1B5E2834F79501C1413D42762A80	FUNCTION
2 ORD_COUNT	017E034C553FC8A0911254900BE5208	FUNCTION
3 TAX_AMT	CB57A22DF0C8ED794608BB02E91CD833	FUNCTION
4 PLST0JAVAFAC_FUN	BE4CDC7413B90427E7C38725568BAE5C	FUNCTION
5 N	CFEEA6BC0D1EEB095E8273ACB484AE05	FORMAL IN
6 X	2FC123A1ED101340CFDB1F3E9CAA7F9	FORMAL IN
7 X	A4E010EB5FF197741D14C26674BAF704	FORMAL IN
8 CUST_NO	63E4BF5F621EDC75E7A33B08851560B6	FORMAL IN
9 V_COUNT	BF3EC08187C9F04406CB086125C28E4CA	VARIABLE
10 QUERY_CODE_PKG	F498E0F124850E09220CD470C1566F4C	PACKAGE
11 MANAGE_DEPT_PROJ	1B308A9E47B7B0D122ED09487F091157	PACKAGE
12 DEMO_PACK	2C595FE125A2C5A705CD1B8C65A9E502	PACKAGE
***		



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

You can display the unique identifiers in your schema by querying for all 'DECLARATION' identifier actions. Valid actions are:

- **DECLARATION:** All identifiers have one and only one declaration. Each declaration may also have an associated type.
- **TYPE:** Has the value of either packages, function or procedures, object types, triggers, or exceptions
- **SIGNATURE:** A unique value that distinguishes the identifier from other identifiers with the same name, whether they are defined in the same program unit or different program units

## Using USER\_IDENTIFIERS to Find All Local Variables

Find all local variables:

```
SELECT a.NAME variable_name, b.NAME context_name, a.SIGNATURE
FROM USER_IDENTIFIERS a, USER_IDENTIFIERS b
WHERE a.USAGE_CONTEXT_ID = b.USAGE_ID
  AND a.TYPE = 'VARIABLE'
  AND a.USAGE = 'DECLARATION'
  AND a.OBJECT_NAME = 'CREDIT_CARD_PKG'
  AND a.OBJECT_NAME = b.OBJECT_NAME
  AND a.OBJECT_TYPE = b.OBJECT_TYPE
  AND (b.TYPE = 'FUNCTION' or b.TYPE = 'PROCEDURE')
ORDER BY a.OBJECT_TYPE, a.USAGE_ID;
```

Query Result		
VARIABLE_NAME	CONTEXT_NAME	SIGNATURE
1 V_CARD_INFO	UPDATE_CARD_INFO	5FE3409B23709E61A12314F2667949CA
2 I	UPDATE_CARD_INFO	F500BE5A79542F0BA3ABFF4AFC9B6C1E
3 V_CARD_INFO	DISPLAY_CARD_INFO	5ACFED9813606BB5A8BCBC9063F974E4
4 I	DISPLAY_CARD_INFO	E5E48887F019043F4240B84ECA77E916



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

In the example shown in the slide, all local variables belonging to procedures or functions are found for the CREDIT\_CARD\_PKG package.

## Finding Identifier Actions

Find all usages performed on the local variable:

```
SELECT USAGE, USAGE_ID, OBJECT_NAME, OBJECT_TYPE
FROM USER_IDENTIFIERS
WHERE SIGNATURE='5FE3409B23709E61A12314F2667949CA'
ORDER BY OBJECT_TYPE, USAGE_ID;
```

USAGE	USAGE_ID	OBJECT_NAME	OBJECT_TYPE
1 DECLARATION	9	CREDIT_CARD_PKG	PACKAGE BODY
2 ASSIGNMENT	17	CREDIT_CARD_PKG	PACKAGE BODY
3 REFERENCE	19	CREDIT_CARD_PKG	PACKAGE BODY
4 REFERENCE	21	CREDIT_CARD_PKG	PACKAGE BODY
5 REFERENCE	22	CREDIT_CARD_PKG	PACKAGE BODY
6 ASSIGNMENT	23	CREDIT_CARD_PKG	PACKAGE BODY
7 REFERENCE	32	CREDIT_CARD_PKG	PACKAGE BODY



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

You can find all actions performed on an identifier. In the example in the slide, all actions performed on a variable are found by searching on the signature of the variable. The variable is called V\_CARD\_INFO, and it is used in the DISPLAY\_CARD\_INFO procedure. Variable V\_CARD\_INFO's signature is found in the previous query. It is available to you in the SIGNATURE column of the USER\_IDENTIFIERS dictionary view.

The different types of identifier usage:

- DECLARATION
- DEFINITION
- CALL
- REFERENCE
- ASSIGNMENT

## Finding Identifier Actions

Find out where the assignment to the local identifier `i` occurred:

```
SELECT LINE, COL, OBJECT_NAME, OBJECT_TYPE
FROM USER_IDENTIFIERS
WHERE SIGNATURE='5ACFED9813606BB5A8BCBC9063F974E4'
AND USAGE='ASSIGNMENT';
```

LINE	COL	OBJECT_NAME	OBJECT_TYPE
1	14	7 CREDIT_CARD_PKG	PACKAGE BODY



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

The local `i` identifier is found on line 14 of the `CREDIT_CARD_PKG` body.

You can learn more on using PL/Scope here:

<https://docs.oracle.com/database/122/ADFNS/plscope.htm#ADFNS022>

## Lesson Agenda

- Running reports on source code
- PL/Scope
- Oracle Supplied Packages for Code Analysis



ORACLE®

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

## Oracle Supplied Packages for Code Analysis

- Following are Oracle packages you can use for PL/SQL code analysis:
  - DBMS\_DESCRIBE
  - DBMS\_UTLILITY
  - DBMS\_METADATA
  - UTL\_CALL\_STACK



ORACLE®

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

You can use the Oracle-supplied `DBMS_DESCRIBE` package to obtain information about a PL/SQL object. The package contains the `DESCRIBE PROCEDURE` procedure, which provides a brief description of a PL/SQL stored procedure. It takes the name of a stored procedure and returns information about each parameter of that procedure.

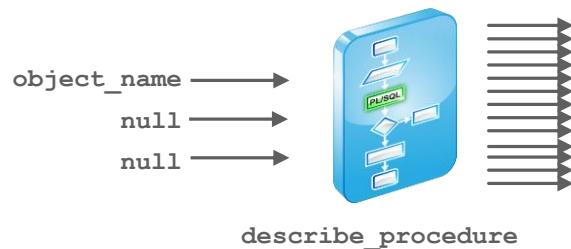
You can use the `DBMS_UTLILITY`-supplied package to follow a call stack and an exception stack.

You can also use `UTL_CALL_STACK` package to understand the execution of the PL/SQL program units.

The `DBMS_METADATA` package provides a way for you to retrieve metadata from the database dictionary as XML or creation DDL and to submit the XML to re-create the object.

## Using DBMS\_DESCRIBE

- Can be used to retrieve information about a PL/SQL object
- Contains one procedure: DESCRIBE\_PROCEDURE
- Includes:
  - Three scalar IN parameters
  - One scalar OUT parameter
  - 12 associative array OUT parameters



**ORACLE**

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

You can use the DBMS\_DESCRIBE package to find information about your procedures. It contains one procedure named DESCRIBE\_PROCEDURE. The DESCRIBE\_PROCEDURE routine accepts the name of the procedure that you are enquiring about. There are two other IN parameters. Both must be either NULL or an empty string. These two parameters are reserved.

The DBMS\_DESCRIBE package has DBMS\_PROCEDURE. The procedure DESCRIBE\_PROCEDURE provides a brief description of a PL/SQL stored procedure. It takes the name of a stored procedure and returns information about each parameter of that procedure.

Following are the parameters:

- **Object name:** name of the procedure being described.
- **Overload:** If overloaded, holds a value for each version of the procedure
- **Position:** Position of the argument in the parameter list. 0 is reserved for the RETURN information of a function.
- **Level:** For composite types only; it holds the level of the data type
- **Argument name:** Name of the argument
- **Data type:** A numerically coded value representing a data type
- **Default value:** 0 for no default value; 1 if the argument has a default value
- **IN\_OUT:** 0 = IN, 1 = OUT, 2 = IN OUT describes the mode of the parameter.
- **Length:** For %rowtype formal arguments, the length constraint is returned, otherwise 0 is returned.

Apart from the listed arguments there are few other parameters which are in the parameter list of the DESCRIBE\_PROCEDURE. The output of the procedure is returned through output parameters.

## Using DBMS\_DESCRIBE

Create a package to call the DBMS\_DESCRIBE.DESCRIBE\_PROCEDURE routine:

The screenshot shows the Oracle SQL Developer interface. At the top, there is a code editor window containing PL/SQL code. Below it, two script output windows show the results of the compilation and execution of the package.

```

CREATE OR REPLACE PACKAGE use_dbms_describe
IS
    PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/

```

**Script Output 1:**

```

Script Output x
Task completed in 0.073 seconds
Package USE_DBMS_DESCRIBE compiled

```

**Script Output 2:**

```

Script Output x
Task completed in 0.229 seconds
Package body USE_DBMS_DESCRIBE compiled

```

**Script Output 3:**

```

Script Output x
Task completed in 0.182 seconds
PL/SQL procedure successfully completed.

Name          Mode Position Datatype
STR           0      1        1

```

**ORACLE** logo and copyright notice: Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

DESCRIBE\_PROCEDURE returns information about your parameters in a set of associative arrays, it is easy to define a package to call and handle the information returned from it.

In the first example in the slide, the specification for the USE\_DBMS\_DESCRIBE package is defined. This package holds one procedure, GET\_DATA. The GET\_DATA routine calls the DBMS\_DESCRIBE.DESCRIBE\_PROCEDURE routine. The implementation of the USE\_DBMS\_DESCRIBE package is shown on the next page. Note that several associative array variables are defined to hold the values returned via the OUT parameters from the DESCRIBE\_PROCEDURE routine. Each of these arrays uses the predefined package types:

```

TYPE VARCHAR_TABLE IS TABLE OF VARCHAR2(30)
INDEX BY BINARY_INTEGER;
TYPE NUMBER_TABLE IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;

```

In the call to the DESCRIBE\_PROCEDURE routine, you need to pass three parameters: the name of the procedure that you are enquiring about and two null values. These null values are reserved for future use.

In the second example in the slide, the results are displayed for the parameters of the query\_code\_pkg.find\_text\_in\_code function. A data type of 1 indicates that it is a VARCHAR2 data type.

**Calling DBMS\_DESCRIBE.DESCRIBE PROCEDURE**

```

CREATE OR REPLACE PACKAGE use_dbms_describe IS
  PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/
CREATE OR REPLACE PACKAGE BODY use_dbms_describe IS
  PROCEDURE get_data (p_obj_name VARCHAR2)
  IS
    v_overload      DBMS_DESCRIBE.NUMBER_TABLE;
    v_position      DBMS_DESCRIBE.NUMBER_TABLE;
    v_level         DBMS_DESCRIBE.NUMBER_TABLE;
    v_arg_name      DBMS_DESCRIBE.VARCHAR2_TABLE;
    v_datatype      DBMS_DESCRIBE.NUMBER_TABLE;
    v_def_value     DBMS_DESCRIBE.NUMBER_TABLE;
    v_in_out        DBMS_DESCRIBE.NUMBER_TABLE;
    v_length        DBMS_DESCRIBE.NUMBER_TABLE;
    v_precision     DBMS_DESCRIBE.NUMBER_TABLE;
    v_scale         DBMS_DESCRIBE.NUMBER_TABLE;
    v_radix         DBMS_DESCRIBE.NUMBER_TABLE;
    v_spare         DBMS_DESCRIBE.NUMBER_TABLE;
  BEGIN
    DBMS_DESCRIBE.DESCRIBE_PROCEDURE
    (p_obj_name, null, null, -- these are the 3 in parameters
     v_overload, v_position, v_level, v_arg_name,
     v_datatype, v_def_value, v_in_out, v_length,
     v_precision, v_scale, v_radix, v_spare, null);
    IF v_in_out.FIRST IS NULL THEN
      DBMS_OUTPUT.PUT_LINE ('No arguments to report.');
    ELSE
      DBMS_OUTPUT.PUT
      ('Name                                Mode');
      DBMS_OUTPUT.PUT_LINE('  Position      Datatype');
      FOR i IN v_arg_name.FIRST .. v_arg_name.LAST LOOP
        IF v_position(i) = 0 THEN
          DBMS_OUTPUT.PUT('This is the RETURN data for
                           the function: ');
        ELSE
          DBMS_OUTPUT.PUT (
            rpad(v_arg_name(i), LENGTH(v_arg_name(i)) +
                  42-LENGTH(v_arg_name(i)), ' '));
        END IF;
        DBMS_OUTPUT.PUT ('      ' ||
                         v_in_out(i) || '      ' || v_position(i) ||
                         '      ' || v_datatype(i));
        DBMS_OUTPUT.NEW_LINE;
      END LOOP;
    END IF;
  END get_data;
END use_dbms_describe;

```

## DBMS.Utility Package

- The DBMS.Utility package provides various utility subprograms.
- You can use these programs for code analysis.
- FORMAT\_CALL\_STACK is a function which can be used on any stored procedure or trigger to access the call stack.
- FORMAT\_ERROR\_BACKTRACE is a function in the package to display the call stack when an exception is raised.
- Accessing the call stack is useful in debugging and function call analysis



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

## Using DBMS\_UTILITY.FORMAT\_CALL\_STACK

Consider the given code:

```
CREATE OR REPLACE PROCEDURE first_one
IS
BEGIN
  dbms_output.put_line(
    substr(dbms_utility.format_call_stack, 1, 255));
END;

CREATE OR REPLACE PROCEDURE second_one
...
CREATE OR REPLACE PROCEDURE third_one
IS
BEGIN
  null;
  null;
  second_one;
END;
```



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

```
SET SERVEROUTPUT ON
CREATE OR REPLACE PROCEDURE first_one
IS
BEGIN
  dbms_output.put_line(
    substr(dbms_utility.format_call_stack, 1, 255));
END;
/

CREATE OR REPLACE PROCEDURE second_one
IS
BEGIN
  null;
  first_one;
END;
/
CREATE OR REPLACE PROCEDURE third_one
IS
BEGIN
  null;
  null;
  second_one;
END;
/
```

## Using DBMS\_UTLILITY.FORMAT\_CALL\_STACK

- This function returns the formatted text string of the current call stack.
- Use it to find the line of code being executed.

```
EXECUTE third_one
```

```
Script Output x | Task completed in 0.058 seconds
PL/SQL procedure successfully completed.

----- PL/SQL Call Stack -----
object      line  object
handle    number  name
0xa3f53850      4  procedure OE.FIRST_ONE
0xb191a0a8      5  procedure OE.SECOND_ONE
0xa3cb9638      6  procedure OE.THIRD_ONE
0xd1bdd7a0      1  anonymous block
```

ORACLE

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

The output from the FORMAT\_CALL\_STACK function shows the object handle number, the line number where a routine is called from, and the routine that is called. Note that the NULL; statements added to the procedures are used to emphasize the line number where the routine is called from.

## Using DBMS\_UTILITY

DBMS\_UTILITY.FORMAT\_ERROR\_BACKTRACE

- Shows you the call stack at the point where an exception is raised
- Returns:
  - The backtrace string
  - A null string if no errors are being handled

DBMS\_UTILITY.FORMAT\_ERROR\_STACK

- This function is used to format the current error stack.



ORACLE

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

You can use the DBMS\_UTILITY.FORMAT\_ERROR\_BACKTRACE function to display the call stack at the point where an exception was raised, even if the procedure is called from an exception handler in an outer scope. The output returned is similar to the output of the SQLERRM function, but not subject to the same size limitation.

You can use the DBMS\_UTILITY.FORMAT\_ERROR\_STACK function to format the current error stack. It can be used in exception handlers to view the full error stack. The function returns the error stack up to 2,000 bytes.

## Using DBMS\_UTLILITY

```
CREATE OR REPLACE PROCEDURE top_with_logging IS
  -- NOTE: SQLERRM in principle gives the same info
  -- as format_error_stack.
  -- But SQLERRM is subject to some length limits,
  -- while format_error_stack is not.
BEGIN
  P5(); -- this procedure, in turn, calls others,
        -- building a stack. P0 contains the exception
EXCEPTION
  WHEN OTHERS THEN
    log_errors ( 'Error Stack...' || CHR(10) ||
      DBMS_UTLILITY.FORMAT_ERROR_STACK() );
    log_errors ( 'Error Backtrace...' || CHR(10) ||
      DBMS_UTLILITY.FORMAT_ERROR_BACKTRACE() );
    DBMS_OUTPUT.PUT_LINE ( '-----' );
END top_with_logging;
/
```



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

To show you the functionality of the FORMAT\_ERROR\_STACK and FORMAT\_ERROR\_BACKTRACE functions, a TOP\_WITH\_LOGGING procedure is created. This procedure calls the LOG\_ERRORS procedure and passes to it the results of the FORMAT\_ERROR\_STACK and FORMAT\_ERROR\_BACKTRACE functions.

The LOG\_ERRORS procedure is shown on the next page.

## Finding Error Information

```

CREATE OR REPLACE PROCEDURE log_errors ( i_buff IN VARCHAR2 ) IS
  g_start_pos PLS_INTEGER := 1;
  g_end_pos   PLS_INTEGER;
  FUNCTION output_one_line RETURN BOOLEAN IS
BEGIN
  g_end_pos := INSTR ( i_buff, CHR(10), g_start_pos );
  CASE g_end_pos > 0
    WHEN TRUE THEN
      DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff,
                                      g_start_pos, g_end_pos-g_start_pos ) );
      g_start_pos := g_end_pos+1;
      RETURN TRUE;
    WHEN FALSE THEN
      DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff, g_start_pos,
                                      (LENGTH(i_buff)-g_start_pos)+1 ) );
      RETURN FALSE;
    END CASE;
  END output_one_line;
BEGIN
  WHILE output_one_line() LOOP NULL;
  END LOOP;
END log_errors;

```



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

### **LOG\_ERRORS: Example**

This procedure takes the return results of the FORMAT\_ERROR\_STACK and FORMAT\_ERROR\_BACKTRACE functions as an IN string parameter, and reports the results back to you using DBMS\_OUTPUT.PUT\_LINE. The LOG\_ERRORS procedure is called twice from the TOP\_WITH\_LOGGING procedure. The first call passes the results of FORMAT\_ERROR\_STACK, and the second procedure passes the results of FORMAT\_ERROR\_BACKTRACE.

**Note:** You can use UTL\_FILE instead of DBMS\_OUTPUT to write and format the results to a file.

Next, several procedures are created and one procedure calls another so that a stack of procedures is built. The P0 procedure raises a zero divide exception when it is invoked. The call stack is:

```
TOP_WITH_LOGGING > P5 > P4 > P3 > P2 > P1 > P0
    SET DOC OFF
    SET FEEDBACK OFF
    SET ECHO OFF

    CREATE OR REPLACE PROCEDURE P0 IS
        e_01476 EXCEPTION;
        pragma exception_init ( e_01476, -1476 );
    BEGIN
        RAISE e_01476; -- this is a zero divide error
    END P0;
    /
    CREATE OR REPLACE PROCEDURE P1 IS
    BEGIN
        P0();
    END P1;
    /
    CREATE OR REPLACE PROCEDURE P2 IS
    BEGIN
        P1();
    END P2;
    /
    CREATE OR REPLACE PROCEDURE P3 IS
    BEGIN
        P2();
    END P3;
    /
    CREATE OR REPLACE PROCEDURE P4 IS
        BEGIN P3();
    END P4;
    /
    CREATE OR REPLACE PROCEDURE P5 IS
        BEGIN P4();
    END P5;
    /
    CREATE OR REPLACE PROCEDURE top IS
    BEGIN
        P5(); -- this procedure is used to show the results
              -- without using the TOP_WITH_LOGGING routine.
    END top;
    /
    SET FEEDBACK ON
```

## Finding Error Information

Results:

The screenshot shows the 'Script Output' window from Oracle SQL Developer. The title bar says 'Script Output x'. Below it are icons for copy, paste, save, and refresh, followed by the text 'Task completed in 0.096 seconds'. The main area displays the following output:

```
PL/SQL procedure successfully completed.

Error_Stack...
ORA-01476: divisor is equal to zero
ORA-06512: at "OE.PO", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2

Error_Backtrace...
ORA-06512: at "OE.PO", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP_WITH_LOGGING", line 7

-----
```

ORACLE

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

The results from executing the `TOP_WITH_LOGGING` procedure is shown in the slide. Note that the error stack displays the exception encountered. The backtrace information traces the flow of the exception to its origin.

If you execute the `TOP` procedure without using the `TOP_WITH_LOGGING` procedure, these are the results:

The screenshot shows the 'Script Output' window from Oracle SQL Developer. The title bar says 'Script Output x'. Below it are icons for copy, paste, save, and refresh, followed by the text 'Task completed in 0.046 seconds'. The main area displays the following output:

```
Error starting at line : 358 in command -
EXECUTE top
Error report -
ORA-01476: divisor is equal to zero
ORA-06512: at "OE.PO", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP", line 3
ORA-06512: at line 1
01476. 00000 - "divisor is equal to zero"
*Cause:
*Action:
```

## DBMS\_METADATA Package

- The DBMS\_METADATA package provides a way for you to retrieve metadata from the database dictionary as XML.
- You can also create DDL or submit the XML code to re-create the object.
- You can use DBMS\_METADATA package to extract DDL of an object, of a user or a role.
- Use DBMS\_METADATA when you need to backup certain objects that are going to be changed or dropped.



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

Over time, as you have used the Oracle database, you may have developed your own code for extracting metadata from the dictionary, manipulating the metadata (adding columns, changing column data types, and so on) and then converting the metadata to DDL so that you could re-create the object on the same or another database. Keeping that code updated to support new dictionary features has probably proven to be challenging.

The DBMS\_METADATA API eliminates the need for you to write and maintain your own code for metadata extraction. It provides a centralized facility for the extraction, manipulation, and re-creation of dictionary metadata. And it supports all dictionary objects at their most current level.

Although the DBMS\_METADATA API can dramatically decrease the amount of custom code you are writing and maintaining, it does not involve any changes to your normal database procedures.

The DBMS\_METADATA API is installed in the same way as data dictionary views, by running `catproc.sql` to run a SQL script at database installation time. Once it is installed, it is available whenever the instance is operational, even in restricted mode.

The DBMS\_METADATA API does not require you to make any source code changes when you change database releases because it is upwardly compatible across different Oracle releases. XML documents retrieved by one release can be processed by the submit interface on the same or later release. For example, XML documents retrieved by an Oracle9*i* database can be submitted to Oracle Database 10g.

## DBMS\_METADATA Subprograms

- You have sub programs for retrieving multiple objects from the database.
- You also have sub programs for submitting XML to the database.
- The package provides two types of retrieval interfaces for two types of usage:
  - For programmatic use
  - For use in SQL queries and for ad hoc browsing



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

The package provides two types of retrieval interfaces for two types of usage:

- **For programmatic use:** OPEN, SET\_FILTER, SET\_COUNT, GET\_QUERY, SET\_PARSE\_ITEM, ADD\_TRANSFORM, SET\_TRANSFORM\_PARAM, SET\_REMAP\_PARAM, FETCH\_xxx, and CLOSE. These enable a flexible selection criteria and the extraction of a stream of objects.
- **For use in SQL queries and for ad hoc browsing:** The GET\_xxx interfaces (GET\_XML and GET\_DDL) return metadata for a single named object. The GET\_DEPENDENT\_xxx and GET\_GRANTED\_xxx interfaces return metadata for one or more dependent or granted objects. None of these APIs supports heterogeneous object types.

The table in the slide provides an overview of the procedures and functions that are available in the DBMS\_METADATA package.

## Subprograms in DBMS\_METADATA (continued)

Subprogram	Description
ADD_TRANSFORM function	Specifies a transform that <code>FETCH_xxx</code> applies to the XML representation of the retrieved objects
CLOSE Procedure	Invalidates the handle returned by <code>OPEN</code> and cleans up the associated state
CONVERT procedures and functions	Converts an XML document to DDL
FETCH_xxx procedures and functions	Returns metadata for objects meeting the criteria established by <code>OPEN</code> , <code>SET_FILTER</code> , <code>SET_COUNT</code> , <code>ADD_TRANSFORM</code> , and so on
GET_xxx functions	Fetches the metadata for a specified object as XML, SXML, or DDL, using only a single call
GET_QUERY function	Returns the text of the queries that are used by <code>FETCH_xxx</code>
OPEN function	Specifies the type of object to be retrieved, the version of its metadata, and the object model
OPENW function	Opens a write context
PUT function	Submits an XML document to the database
SET_COUNT Procedure	Specifies the maximum number of objects to be retrieved in a single <code>FETCH_xxx</code> call
SET_FILTER Procedure	Specifies restrictions on the objects to be retrieved, for example, the object name or schema
SET_PARSE_ITEM Procedure	Enables output parsing by specifying an object attribute to be parsed and returned
SET_TRANSFORM_PARAM and SET_REMAP_PARAM Procedures	Specifies parameters to the XSLT stylesheets identified by <code>transform_handle</code>

## FETCH\_XXX Subprograms

Name	Description
FETCH_XML	This function returns the XML metadata for an object as an XMLType.
FETCH_DDL	This function returns the DDL (either to create or to drop the object) into a predefined nested table.
FETCH_CLOB	This function returns the objects (transformed or not) as a CLOB.
FETCH_XML_CLOB	This procedure returns the XML metadata for the objects as a CLOB in an IN OUT NOCOPY parameter to avoid expensive LOB copies.

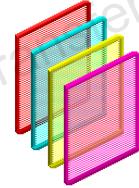


Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

The functions and procedure listed in the slide return metadata for objects meeting the criteria established by the call to the OPEN function that returned the handle, and the subsequent calls to SET\_FILTER, SET\_COUNT, ADD\_TRANSFORM, and so on. Each call to `FETCH_XXX` returns the number of objects specified by `SET_COUNT` (or a smaller number, if fewer objects remain in the current cursor) until all objects are returned.

## Filters on Metadata

- You can define filters to restrict the objects to be retrieved.
- Use `SET_FILTER` procedure to define a filter on the fetched data.
- `SET_FILTER` is an overloaded procedure.



ORACLE

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

The filters defined through `SET_FILTER` procedure are organized into object type categories. Some of the object type categories are:

Named objects, Tables, Objects dependent on tables, Index, Dependent objects, Granted objects, Table data, Index statistics, Constraints, All object types, Database export

## SET\_FILTER Procedure

- Syntax:

```
PROCEDURE set_filter
( handle IN NUMBER,
  name   IN VARCHAR2,
  value  IN VARCHAR2|BOOLEAN|NUMBER,
  object_type_path VARCHAR2
);
```

- Example:

```
...
DBMS_METADATA.SET_FILTER (handle, 'NAME', 'OE');
...
```



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

You use the SET\_FILTER procedure to identify restrictions on the objects that are to be retrieved. For example, you can specify restrictions on an object or schema that is being retrieved. This procedure is overloaded with parameters that have the following meanings:

- handle is the handle returned from the OPEN function.
- name is the name of the filter. For each filter, the object type applies to its name, data type (text or Boolean), and meaning or effect (including its default value, if there is one).
- value is the value of the filter. It can be text, Boolean, or a numeric value.
- object\_type\_path is a path name designating the object types to which the filter applies. By default, the filter applies to the object type of the OPEN handle.

## Examples of Setting Filters

Set the filter to fetch the OE schema objects excluding the object types of functions, procedures, and packages, as well as any views that contain PAYROLL at the start of the view name:

```
DBMS_METADATA.SET_FILTER(handle, 'SCHEMA_EXPR',
    'IN (''PAYROLL'', ''OE''));
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
    '='''FUNCTION'''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
    '='''PROCEDURE'''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_PATH_EXPR',
    '='''PACKAGE'''');
DBMS_METADATA.SET_FILTER(handle, 'EXCLUDE_NAME_EXPR',
    'LIKE ''PAYROLL%'''', 'VIEW');
```



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

The example shown in the slide calls the SET\_FILTER procedure several times to create a WHERE condition that identifies which object types are to be fetched. First, the objects in the PAYROLL and OE schemas are identified as object types to be fetched. Subsequently, the SET\_FILTER procedure identifies certain object types (functions, procedures, and packages) and view object names that are to be excluded.

## Programmatic Use: Example 1

```

CREATE PROCEDURE example_one IS
  v_hdl NUMBER; v_th1 NUMBER; v_th2 NUMBER;
  v_doc sys.ku$_ddls; ← 1
BEGIN
  v_hdl := DBMS_METADATA.OPEN('SCHEMA_EXPORT'); ← 2
  DBMS_METADATA.SET_FILTER(v_hdl, 'SCHEMA', 'OE'); ← 3
  v_th1 := DBMS_METADATA.ADD_TRANSFORM(v_hdl, ←
    'MODIFY', NULL, 'TABLE'); ← 4
  DBMS_METADATA.SET_REMAP_PARAM(v_th1, ←
    'REMAP_TABLESPACE', 'SYSTEM', 'TBS1'); ← 5
  v_th2 := DBMS_METADATA.ADD_TRANSFORM(v_hdl, 'DDL'); ←
  DBMS_METADATA.SET_TRANSFORM_PARAM(v_th2, ←
    'SQLTERMINATOR', TRUE); ← 6
  DBMS_METADATA.SET_TRANSFORM_PARAM(v_th2, ←
    'REF_CONSTRAINTS', FALSE, 'TABLE'); ← 7
  LOOP
    v_doc := DBMS_METADATA.FETCH_DDL(v_hdl); ←
    EXIT WHEN v_doc IS NULL;
  END LOOP;
  DBMS_METADATA CLOSE(v_hdl); ← 8
END;

```

ORACLE

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

In this example, all objects are retrieved from the HR schema as creation DDL. The MODIFY transform is used to change the tablespaces for the tables.

1. The DBMS\_METADATA package has several predefined types that are owned by SYS. The sys.ku\$ddls stand-alone object type is used in the DBMS\_METADATA package. It is a table type that holds the CLOB type of data.
2. You use the OPEN function to specify the type of object to be retrieved, the version of its metadata, and the object model. It returns a context handle for the set of objects. In this example, 'SCHEMA\_EXPORT' is the object type, which indicates all metadata objects in a schema. There are 85 predefined types of objects for the model that you can specify for this parameter. Both the version of metadata and the object model parameters are not identified in this example. The version of the metadata parameter defaults to 'COMPATIBLE'. You can also specify 'LATEST' or a specific database version.
3. The SET\_FILTER procedure identifies restrictions on the objects that are to be retrieved.

4. The ADD\_TRANSFORM function specifies a transform that FETCH\_XXX applies to the XML representation of the retrieved objects. You can have multiple transforms. In the example, two transforms occur, one for each of the th1 and th2 program variables. The ADD\_TRANSFORM function accepts four parameters and returns a number representing the opaque handle to the transform. The parameters are the handle returned from the OPEN statement, the name of the transform (DDL, DROP, or MODIFY), the encoding name (which is the name of the national language support [NLS] character set in which the style sheet pointed to by the name is encoded), and the object type. If the object type is omitted, the transform applies to all objects; otherwise, it applies only to the object type specified. The first transform shown in the program code is the handle returned from the OPEN function. The second transform shown in the code has two parameter values specified. The first parameter is the handle identified from the OPEN function. The second parameter value is the DDL, which means the document is transformed to the DDL that creates the object. The output of this transform is not an XML document. The third and fourth parameters are not specified. Both take the default values for the encoding and object type parameters.
5. The SET\_REMAP\_PARAM procedure identifies the parameters to the XSLT style sheet identified by the transform handle, which is the first parameter passed to the procedure. In the example, the second parameter value 'REMAP\_TABLESPACE' means that the objects have their tablespaces renamed from an old value to a new value. In the ADD\_TRANSFORM function, the choices are DDL, DROP, or MODIFY. For each of these values, the SET\_REMAP\_PARAM identifies the name of the parameter. REMAP\_TABLESPACE means the objects in the document will have their tablespaces renamed from an old value to a new value. The third and fourth parameters identify the old value and the new value. In this example, the old tablespace name is SYSTEM, and the new tablespace name is TBS1.
6. SET\_TRANSFORM\_PARAM works similarly to SET\_REMAP\_PARAM. In the code shown, the first call to SET\_TRANSFORM\_PARAM identifies the parameters for the th2 variable. The SQLTERMINATOR and TRUE parameter values cause the SQL terminator (; or /) to be appended to each DDL statement. The second call to SET\_TRANSFORM\_PARAM identifies more characteristics for the th2 variable. REF\_CONSTRAINTS, FALSE, and TABLE means that the referential constraints on the tables are not copied to the document.
7. The FETCH\_DDL function returns the metadata for objects that meet the criteria established by the OPEN, SET\_FILTER, ADD\_TRANSFORM, SET\_REMAP\_PARAM, and SET\_TRANSFORM\_PARAM subroutines.
8. The CLOSE function invalidates the handle returned by the OPEN function and cleans up the associated state. Use this function to terminate the stream of objects established by the OPEN function.

## Programmatic Use: Example 2

```

CREATE FUNCTION get_table_md RETURN CLOB IS
  v_hdl NUMBER; -- returned by 'OPEN'
  v_th NUMBER; -- returned by 'ADD_TRANSFORM'
  v_doc CLOB;
BEGIN
  -- specify the OBJECT TYPE
  v_hdl := DBMS_METADATA.OPEN('TABLE');
  -- use FILTERS to specify the objects desired
  DBMS_METADATA.SET_FILTER(v_hdl , 'SCHEMA', 'OE');
  DBMS_METADATA.SET_FILTER
    (v_hdl , 'NAME','ORDERS');
  -- request to be TRANSFORMED into creation DDL
  v_th := DBMS_METADATA.ADD_TRANSFORM(v_hdl,'DDL');
  -- FETCH the object
  v_doc := DBMS_METADATA.FETCH_CLOB(v_hdl);
  -- release resources
  DBMS_METADATA CLOSE(v_hdl);
  RETURN v_doc;
END;
/

```



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

The example in this slide returns the metadata for the ORDERS table. The result is:

```

set pagesize 0
set long 1000000
SELECT get_table_md FROM dual;

```

```

CREATE TABLE "OE"."ORDERS"
(
  "ORDER_ID" NUMBER(12,0),
  "ORDER_DATE" TIMESTAMP (6) WITH LOCAL TIME ZONE CONSTRAINT "ORDER_DATE_NN" NOT
NULL ENABLE,
  "ORDER_MODE" VARCHAR2(8),
  "CUSTOMER_ID" NUMBER(6,0) CONSTRAINT "ORDER_CUSTOMER_ID_NN" NOT NULL ENABLE,
  "ORDER_STATUS" NUMBER(2,0),
  "ORDER_TOTAL" NUMBER(8,2),
  "SALES_REP_ID" NUMBER(6,0),
  "PROMOTION_ID" NUMBER(6,0),
  CONSTRAINT "ORDER_MODE_LOV" CHECK (order_mode in ('direct','online')) ENABLE,
  ...
)

```

```
...
CONSTRAINT "ORDER_TOTAL_MIN" CHECK (order_total >= 0) ENABLE,
CONSTRAINT "ORDER_PK" PRIMARY KEY ("ORDER_ID")

USING INDEX PCTFREE 10 INITTRANS 2 MAXTRANS 255 COMPUTE STATISTICS
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
TABLESPACE "USERS"  ENABLE,
CONSTRAINT "ORDERS_SALES_REP_FK" FOREIGN KEY ("SALES REP_ID")
REFERENCES "HR"."EMPLOYEES" ("EMPLOYEE_ID") ON DELETE SET NULL ENABLE,
CONSTRAINT "ORDERS_CUSTOMER_ID_FK" FOREIGN KEY ("CUSTOMER_ID")
REFERENCES "OE"."CUSTOMERS" ("CUSTOMER_ID") ON DELETE SET NULL ENABLE
) SEGMENT CREATION IMMEDIATE
PCTFREE 10 PCTUSED 40 INITTRANS 1 MAXTRANS 255
NOCOMPRESS LOGGING
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
BUFFER_POOL DEFAULT FLASH_CACHE DEFAULT CELL_FLASH_CACHE DEFAULT)
TABLESPACE "USERS"
```

You can accomplish the same effect with the browsing interface:

```
SELECT dbms_metadata.get_ddl
      ('TABLE', 'ORDERS', 'OE')
FROM dual;
```

## Browsing APIs

Name	Description
GET_XXX	The GET_XML and GET_DDL functions return metadata for a single named object.
GET_DEPENDENT_XXX	This function returns metadata for a dependent object.
GET_GRANTED_XXX	This function returns metadata for a granted object.
<b>Where xxx is:</b>	<b>DDL or XML</b>



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

The browsing APIs are designed for use in SQL queries and ad hoc browsing. These functions enable you to fetch metadata for objects with a single call. They encapsulate calls to OPEN, SET\_FILTER, and so on. The function that you use depends on the characteristics of the object type and whether you want XML or DDL.

For some object types, you can use multiple functions. You can use GET\_XXX to fetch an index by name, or GET\_DEPENDENT\_XXX to fetch the same index by specifying the table on which it is defined.

GET\_XXX returns a single object name.

For GET\_DEPENDENT\_XXX and GET\_GRANTED\_XXX, an arbitrary number of granted or dependent objects may match the input criteria. However, you can specify an object count when fetching these objects.

If you invoke these functions from SQL\*Plus, you should use the SET LONG and SET PAGESIZE commands to retrieve the complete, uninterrupted output.

```
SET LONG 2000000
SET PAGESIZE 300
```

## Using the UTL\_CALL\_STACK Package

The UTL\_CALL\_STACK package is introduced newly in Oracle Database 12c. This package:

- Is similar to DBMS.Utility.Format\_Call\_Stack
- Defines a VARRAY type UNIT\_QUALIFIED\_NAME
- Provides subprograms to return the current call stack for a PL/SQL program
- Displays the call stack in a structured format
- Is well suited for programmatic analysis



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

The UTL\_CALL\_STACK package provides an interface for PL/SQL programmers to get information about currently executing programs including the subprogram name from dynamic and lexical stacks and the depths of those stacks.

Individual functions return subprogram names, unit names, owner names, edition names, and line numbers for given dynamic depths.

More functions return error stack information.

Using these information, you can create more revealing error logs and application execution traces.

*To know the complete set of UTL\_CALL\_STACK subprograms, refer to 'Oracle Database PL/SQL Packages and Types Reference 12c Release 2' documentation.*

## DEPRECATE Pragma

- DEPRECATE pragma marks a PL/SQL element as deprecated.
- You can apply it to whole unit or a subprogram within an unit.

Usage:

```
...
PRAGMA DEPRECATE(P1, 'P1 is deprecated, you must use the
new P2') ;
...
```

- When you use a deprecated item, the compiler will return a warning.



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

You might have a new implementation of a functionality but you still cannot do away with the old implementation of it. In such scenario you can mark the older implementation to be deprecated using the DEPRECATE pragma.

For example, the UTL\_CALL\_STACK package, which was new in Oracle Database 12c Release 1 (12.1), provided the functionality that earlier was provided by the FORMAT\_CALL\_STACK(), FORMAT\_ERROR\_STACK() and FORMAT\_ERROR\_BACKTRACE() procedures in the DBMS\_UTILITY package. The same thing happens in PL/SQL code that is developed by customers. The old API cannot be removed as soon as the new API is available because of the quantity of its existing uses; but new code that requires the functionality uses the new API.

## Quiz



Which of the following SQL Developer predefined reports would you use to find the occurrence of a text string or an object name within a PL/SQL coding?

- a. Search Source Code
- b. Program Unit Arguments
- c. Unit Line Counts

ORACLE

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

**Answer: a**

## Quiz



Which of the following would you use to find information about arguments for procedures and functions?

- a. The ALL\_ARGUMENTS view
- b. The DBMS\_DESCRIBE.DESCRIBE\_PROCEDURE routine
- c. SQL Developer predefined report, Program Unit Arguments
- d. None of the above

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

ORACLE



**Answer: b, c**

## Quiz



You can invoke `DBMS_METADATA` to retrieve metadata from the database dictionary as XML or creation DDL, and submit the XML to re-create the object.

- a. True
- b. False

**ORACLE**

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



**Answer: a**

## Summary

In this lesson, you should have learned how to:

- Use the supplied packages and the dictionary views to find coding information
- Determine identifier types and usages with PL/Scope
- Use the `DBMS_METADATA` package to obtain metadata from the data dictionary as XML or creation DDL that can be used to re-create the objects



ORACLE

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

This lesson showed you how to use the dictionary views and supplied PL/SQL packages to analyze your PL/SQL applications.

## Practice 10: Overview

This practice covers the following topics:

- Analyzing PL/SQL code
- Using PL/Scope
- Using DBMS\_METADATA



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

Using the OE application that you have created, write code to analyze your application. You will perform the following:

- Find coding information
- Use PL/Scope
- Use DBMS\_METADATA

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

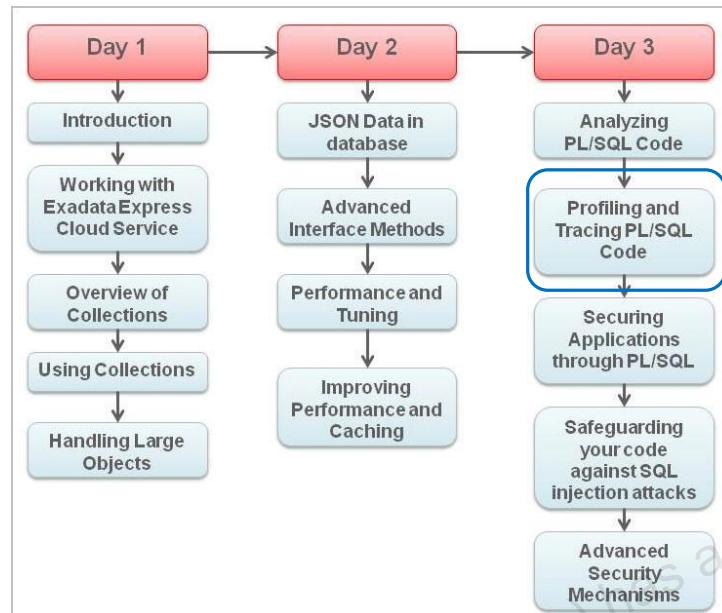
PELISSIER LOIC (loic.pelissier@gmail.com) has a non-transferable  
license to use this Student Guide.

# Profiling and Tracing PL/SQL Code

The Oracle logo, consisting of the word "ORACLE" in white capital letters on a red rectangular background.

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

## Course Agenda



ORACLE

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



In this lesson, you are introduced to Collections. Collections are composite data types, you can store values of data which can in turn have various internal components. In earlier course ( Oracle database 12c: PL/SQL Program Units) you must've learnt about records. Records are also composite data types.

A collection stores a set of values of same data type, whereas a record stores a set of values of different data types. In this lesson we will discuss different variants of collections.

## Objectives

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

- Trace PL/SQL program execution
- Profile PL/SQL applications



ORACLE

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

In this lesson, you will learn how to trace and profile PL/SQL applications.

## Lesson Agenda

- Tracing PL/SQL program execution
- Profiling PL/SQL applications



ORACLE®

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

## Tracing PL/SQL Execution

- You can understand the program execution path of the application by tracing it.
- Use `DBMS_TRACE` package to trace the application.



ORACLE

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

In large and complex PL/SQL applications, it can sometimes become difficult to keep track of subprogram calls when a number of subprograms call each other. By tracing your PL/SQL code, you can get a clearer idea of the paths and order in which your programs execute.

Oracle provides an API for tracing the execution of PL/SQL programs on the server. You can use the Trace API, implemented on the server as the `DBMS_TRACE` package, to trace PL/SQL subprogram code.

**Note:** You cannot use PL/SQL tracing with the multithreaded server (MTS).

## Tracing PL/SQL Execution

The DBMS\_TRACE package contains the following PL/SQL subprograms:

Subprogram	Description
<b>CLEAR_PLSQL_TRACE procedure</b>	Stops trace data dumping in session
<b>GET_PLSQL_TRACE_LEVEL function</b>	Gets the trace level
<b>PLSQL_TRACE_VERSION procedure</b>	Gets the version number of the trace package
<b>SET_PLSQL_TRACE procedure</b>	Starts tracing in the current session



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

### The DBMS\_TRACE Programs

DBMS\_TRACE provides subprograms to start and stop PL/SQL tracing in a session. The trace data is collected as the program executes, and it is written out to data dictionary tables.

A typical trace session involves:

- Enabling specific subprograms for trace data collection (optional)
- Starting the PL/SQL tracing session (`DBMS_TRACE.set_plsql_trace`)
- Running the application that is to be traced
- Stopping the PL/SQL tracing session (`DBMS_TRACE.clear_plsql_trace`)

## Tracing PL/SQL Execution

- `DBMS_TRACE.set_plsql_trace` starts a tracing session.
- You can start the tracing session with various trace call constants.
- You define what units have to be traced with trace call constants.
- The constants `TRACE_PAUSE` and `TRACE_RESUME` are used for trace control.
- `DBMS_TRACE.clear_plsql_trace` stops the tracing session.



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

Name	Type	Value	Description
TRACE_ALL_CALLS	INTEGER	1	Traces calls or returns
TRACE_ENABLED_CALLS	INTEGER	2	
TRACE_ALL_EXCEPTIONS	INTEGER	4	Traces exceptions
TRACE_ENABLED_EXCEPTIONS	INTEGER	8	Traces exceptions and handlers
TRACE_LIMIT	INTEGER	16	Save only the last few records. This allows tracing up to a problem area, without filling the database up with masses of irrelevant information. If event 10940 is set, the limit is 1023*(the value of event 10940). This can be overridden by the use of "TRACE_LIMIT" flag.
TRACE_ALL_SQL	INTEGER	32	Traces SQL statements
TRACE_ENABLED_SQL	INTEGER	64	Traces SQL statements at PL/SQL level. This does not invoke SQL Trace

TRACE_ALL_LINES	INTEGER	128	Traces each line
TRACE_ENABLED_LINES	INTEGER	256	
TRACE_PAUSE	INTEGER	4096	Pauses tracing
TRACE_RESUME	INTEGER	8192	Resume tracing
TRACE_STOP	INTEGER	16384	Stops tracing
NO_TRACE_ADMINISTRATIVE	INTEGER	32768	Prevents tracing of 'administrative events such as •PL/SQL Trace Tool started •Trace flags changed •PL/SQL Virtual Machine started •PL/SQL Virtual Machine stopped
NO_TRACE_HANDLED_EXCEPTIONS	INTEGER	65536	Prevents tracing of handled exceptions

### Specifying a Trace Level

During the trace session, there are two levels that you can specify to trace calls, exceptions, SQL, and lines of code.

#### Trace Calls

- **Level 1:** Trace all calls. This corresponds to the constant `trace_all_calls`.
- **Level 2:** Trace calls only to enabled program units. This corresponds to the constant `trace_enabled_calls`.

#### Trace Exceptions

- **Level 1:** Trace all exceptions. This corresponds to `trace_all_exceptions`.
- **Level 2:** Trace exceptions raised only in enabled program units. This corresponds to `trace_enabled_exceptions`.

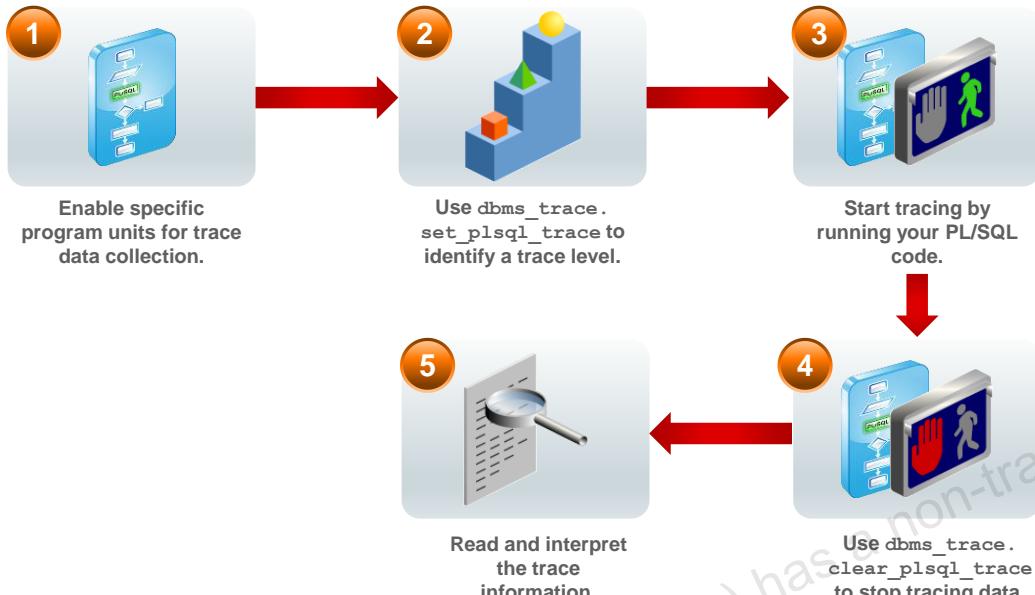
#### Trace SQL

- **Level 1:** Trace all SQL. This corresponds to the constant `trace_all_sql`.
- **Level 2:** Trace SQL only in enabled program units. This corresponds to the constant `trace_enabled_sql`.

#### Trace Lines

- **Level 1:** Trace all lines. This corresponds to the constant `trace_all_lines`.
- **Level 2:** Trace lines only in enabled program units. This corresponds to the constant `trace_enabled_lines`.

## Tracing PL/SQL: Steps



ORACLE

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

To trace PL/SQL code by using the `dbms_trace` package, perform the following steps:

1. Enable specific program units for trace data collection.
2. Use `DBMS_TRACE.set_plsql_trace` to identify a trace level.
3. Run your PL/SQL code.
4. Use `DBMS_TRACE.clear_plsql_trace` to stop tracing data.
5. Read and interpret the trace information.

The following few slides demonstrate the steps to accomplish PL/SQL tracing.

## Step 1: Enable Specific Subprograms

Enable specific subprograms with one of the two methods:

- Enable a subprogram by compiling it with the debug option:

```
ALTER SESSION SET PLSQL_DEBUG=true;
```

```
CREATE OR REPLACE ....
```

- Recompile a specific subprogram with the debug option:

```
ALTER [PROCEDURE | FUNCTION | PACKAGE]
    <subprogram-name> COMPILE DEBUG [BODY];
```



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

Profiling large applications may produce a huge volume of data that can be difficult to manage. Before turning on the trace facility, you have the option to control the volume of data collected by enabling a specific subprogram for trace data collection. You can enable a subprogram by compiling it with the debug option. You can do this in one of two ways:

- Enable a subprogram by compiling it with the `ALTER SESSION` `debug` option, and then compile the program unit by using the `CREATE OR REPLACE` syntax:

```
ALTER SESSION SET PLSQL_DEBUG = true;
CREATE OR REPLACE ...
```

- Recompile a specific subprogram with the debug option:

```
ALTER [PROCEDURE | FUNCTION | PACKAGE]
    <subprogram-name> COMPILE DEBUG [BODY];
ALTER PROCEDURE P5 COMPILE DEBUG;
```

**Note:** The second method cannot be used for anonymous blocks.

Enabling specific subprograms enables you to:

- Limit and control the amount of trace data, especially in large applications
- Obtain additional trace information that is otherwise not available. For example, during the tracing session, if a subprogram calls another subprogram, the name of the called subprogram is included in the trace data if the calling subprogram was enabled by compiling it in debug mode.

## Steps 2 and 3: Identify a Trace Level and Start Tracing

- Specify the trace level by using `dbms_trace.set_plsql_trace`:

```
EXECUTE DBMS_TRACE.set_plsql_trace -  
      (tracelevel1 + tracelevel2 ...)
```

- Execute the code that is to be traced:

```
EXECUTE my_program
```



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

To trace PL/SQL code execution by using `DBMS_TRACE`, perform the following steps:

- Start the trace session by using the syntax shown in the slide. For example:

```
EXECUTE -  
      DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.trace_all_calls)
```

- Execute the PL/SQL code. The trace data is written to the data dictionary views.

To specify additional trace levels in the argument, use the “+” symbol between each trace level value. Following rules apply when you are using multiple trace levels:

- Lower trace levels supersede higher levels when tracing is activated for multiple tracing levels.
- If tracing is requested only for enabled subprograms and if the current subprogram is not enabled, no trace data is written.
- If the current subprogram is enabled, call tracing writes out the subprogram type, name, and stack depth.
- If the current subprogram is not enabled, call tracing writes out the subprogram type, line number, and stack depth.
- Exception tracing writes out the line number. Raising the exception shows information about whether the exception is user-defined or predefined and, in the case of predefined exceptions, the exception number.

## Step 4 and Step 5: Turn Off and Examine the Trace Data

- Remember to turn tracing off by using the DBMS\_TRACE.clear\_plsql\_trace procedure:

```
EXECUTE DBMS_TRACE.clear_plsql_trace
```
- Examine the trace information:
  - Call tracing writes out the program unit type, name, and stack depth.
  - Exception tracing writes out the line number.



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

After tracing the PL/SQL program unit, turn tracing off by executing DBMS\_TRACE.clear\_plsql\_trace. This stops any further writing to the trace tables.

To avoid the overhead of writing the trace information, it is recommended that you turn tracing off when you are not using it.

## plsql\_trace\_runs and plsql\_trace\_events

- Trace information is written to the following dictionary views:
  - plsql\_trace\_runs
  - plsql\_trace\_events
- Run the `tracetab.sql` script to create the dictionary views.
- You need privileges to view the trace information in the dictionary views.



ORACLE®

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

All trace information is written to the dictionary views `plsql_trace_runs` and `plsql_trace_events`. These views are created (typically by a DBA) by running the `tracetab.sql` script. The script is located in the `/u01/app/oracle/product/12.2.0/dbhome_1/rdbms/admin` folder. Run the script as `SYS`. After the script is run, you need the `SELECT` privilege to view information from these dictionary views.

```
--Execute as sys
@/u01/app/oracle/product/12.2.0/dbhome_1/rdbms/admin/tracetab
GRANT SELECT ON plsql_trace_runs TO OE;
GRANT SELECT ON plsql_trace_events TO OE;
```

## plsql\_trace\_runs and plsql\_trace\_events

```
ALTER SESSION SET PLSQL_DEBUG=TRUE;
ALTER PROCEDURE P5 COMPILE DEBUG;

EXECUTE DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.trace_all_calls)
EXECUTE p5
EXECUTE DBMS_TRACE.CLEAR_PLSQL_TRACE

SELECT proc_name, proc_line,
       event_proc_name, event_comment
FROM sys.plsql_trace_events
WHERE event_proc_name = 'P5'
OR PROC_NAME = 'P5';
```

Query Result			
PROC_NAME	PROC_LINE	EVENT_PROC_NAME	EVENT_COMMENT
1 P5	1 (null)		Procedure Call
2 (null)	1 P5		Procedure Call

ORACLE

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

### Query the plsql\_trace\_runs and plsql\_trace\_events Views

Use the `plsql_trace_runs` and `plsql_trace_events` dictionary views to view the trace information generated by using the `DBMS_TRACE` facility. `plsql_trace_runs` holds generic information about traced programs, such as the date, time, owner, and name of the traced stored program. `dbms_trace_events` holds more specific information about the traced subprograms.

## Lesson Agenda

- Tracing PL/SQL program execution
- Profiling PL/SQL applications



ORACLE®

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

## Profiling PL/SQL Code

- Profiling PL/SQL code enables you to identify areas of performance improvement in PL/SQL code.
- Oracle provides built-in packages that can be used for profiling:
  - DBMS\_PROFILER
  - DBMS\_HPROF
- DBMS\_HPROF writes profile information into HTML reports, we will discuss hierarchical profiling further in this lesson.



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

DBMS\_PROFILER provides a profiler interface, which computes the time that your PL/SQL program spends at each line and in each subprogram. You must have CREATE privileges on the units to be profiled. Saves runtime statistics in database tables, which you can query.

DBMS\_HPROF provides a hierarchical profiler interface, which reports the dynamic execution program profile of your PL/SQL program, organized by subprogram invocations. It requires no special source or compile-time preparation. Generates reports in HTML. Provides the option of storing results in relational format in database tables for custom report generation.

## Hierarchical Profiling

- Used to identify hotspots and performance tuning opportunities in PL/SQL applications
- Reports the dynamic execution profile of a PL/SQL program organized by function calls
- Reports SQL and PL/SQL execution times separately
- Provides function-level summaries.



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

The hierarchical profiler is available to help you identify the hotspots and performance tuning opportunities in your PL/SQL applications. This feature enables you to view reports of how a PL/SQL program is executed, organized by function call, as well as SQL and PL/SQL execution times.

You can view function-level summaries that include the:

- Number of calls to a function
- Time spent in the function itself
- Time spent in the entire subtree under the function
- Detailed parent-children information for each function

You can use this information to tune your PL/SQL applications, and understand the structure, flow, and control of complex programs (especially those written by someone else).

## Hierarchical Profiling Concepts

The PL/SQL hierarchical profiler consists of the:

- Data collection component
  - `start_profiling` procedure
  - `Stop_profiling` procedure
- Analyzer component
  - `analyze` function



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

The PL/SQL hierarchical profiler consists of two subcomponents:

- The data collection component is an intrinsic part of the PL/SQL Virtual Machine. It comprises of `start_profiling` procedure and `stop_profiling` procedure. The raw profiler output is written to a file.
- The analyzer component, which is also exposed by the `DBMS_HPROF` package, is used to process the raw profiler output and upload the results of the profiling into database tables that can then be queried. The `analyze` function in the package performs the analysis. You can use the `plshprof` command-line utility to generate simple HTML reports directly from the raw profiler data.

## Using the PL/SQL Profiler

Using the PL/SQL profiler, you can find information such as:

- The number of calls to a function
- The function time, not including descendants
- The subtree time, including descendants
- Parent-children information for each function such as:
  - Who were the callers of a given function?
  - What functions were called from a particular function?
  - How much time was spent in function X when called from function Y?
  - How many calls to function X came from function Y?
  - How many times did X call Y?



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

Using the hierarchical PL/SQL profiler, you can find both function-level execution summary information, as well as detailed parent-children information for each function.

When profiling is turned on, function entry and exit operations are logged to a file along with time information. Fine-grained elapsed information is collected.

You don't need to recompile PL/SQL modules to use the hierarchical profiler. You can analyze both interpreted and natively compiled PL/SQL modules.

## Using the PL/SQL Profiler

Sample data for profiling:

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
        ...
    END update_card_info;
    PROCEDURE display_card_info
        (p_cust_id NUMBER)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
        ...
    END display_card_info;
END credit_card_pkg; -- package body
```



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

CREDIT\_CARD\_PKG that was created earlier is used to demonstrate hierarchical profiling. The full code is shown on the following pages.

```

CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2);

    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

```

```

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                    WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;

```

```
PROCEDURE display_card_info
    (p_cust_id NUMBER)
IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
BEGIN
    SELECT credit_cards
        INTO v_card_info
        FROM customers
        WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
        FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
            DBMS_OUTPUT.PUT('Card Type: ' ||
v_card_info(idx).card_type
                           || ',');
            DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
v_card_info(idx).card_num );
        END LOOP;
    ELSE
        DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
    END display_card_info;
END credit_card_pkg; -- package body
/
```

## Using the PL/SQL Profiler

```

BEGIN
  -- start profiling
  DBMS_HPROF.START_PROFILING('PROFILE_DATA', 'pd_cc_pkg.txt');
END;

```

1

```

DECLARE
  v_card_info typ_cr_card_nst;
BEGIN
  -- run application
  credit_card_pkg.update_card_info
    (154, 'Discover', '123456789');
END;

```

2

```

BEGIN
  DBMS_HPROF.STOP_PROFILING;
END;

```

3



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

You use the new DBMS\_HPROF package to hierarchically profile PL/SQL code. You must be granted the privilege to execute the routines in the DBMS\_HPROF package:

```
GRANT EXECUTE ON sys.dbms_hprof TO OE;
```

You also need to identify the location of the profiler files. Create a DIRECTORY object to identify this information:

```
CREATE OR REPLACE DIRECTORY profile_data AS
  '/home/oracle/labs/labs';
```

1. The first step is to turn profiling on. Use the DBMS\_HPROF.START\_PROFILING procedure. When calling this procedure, pass the following two parameters:
  - **Directory object:** An alias for a file system path name. You need to have WRITE privileges to this location.
  - **File name:** The name of the file to which you want the output written
  - You can optionally pass a third parameter, max\_depth. When max\_depth value is NULL (the default), profile information is gathered for all functions irrespective of their call depth. When a non-NULL value is specified, the profiler collects data only for functions up to a call depth level of max\_depth.
2. The second step is to run the code that you want profiled.
3. The third step is to turn off profiling. Use the DBMS\_HPROF.STOP\_PROFILING procedure to stop the profiling:

```
EXECUTE DBMS_HPROF.STOP_PROFILING
```

## Understanding Raw Profiler Data

The pd\_cc\_pkg.txt file has the raw profiler data in it.

```
pd_cc_pkg.txt X
P#V PLSHPROF Internal Version 1.0
P#! PL/SQL Timer Started
P#C PLSQL.""."._plsql_vm"
P#X 4
P#C PLSQL.""."._anonymous_block"
P#X 2671
P#C PLSQL."OE"."CREDIT_CARD_PKG":11."UPDATE_CARD_INFO">#3a2b06cfaf1322e42 #3
P#X 69
P#C SQL."OE"."CREDIT_CARD_PKG":11."_static_sql_exec_line9" #9."3zjmncbj06t6"
P#! SELECT CREDIT_CARDS FROM CUSTOMERS WHERE CUSTOMER_
P#X 17651
P#R
P#X 1944
P#C SQL."OE"."CREDIT_CARD_PKG":11."_static_sql_exec_line21" #21."82nnmry1yp117"
P#! UPDATE CUSTOMERS SET CREDIT_CARDS = TYP_CR_CARD_NS
P#X 117888
P#R
P#X 35
P#R
P#X 2
P#R
P#X 5
P#R
P#C PLSQL.""."._plsql_vm"
P#X 3
P#C PLSQL.""."._anonymous_block"
P#X 72
P#C PLSQL."SYS"."DBMS_HPROF":11."STOP_PROFILING"#980980e97e42f8ec #453
P#R
P#R
P#! PL/SQL Timer Stopped
```



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

You can examine the contents of the generated raw profiler text file. However, it is easier to understand the data after using the analyzer component of hierarchical profiling. The text shown in the slide displays the contents of the pd\_cc\_pkg.txt file. Each line starts with a file indicator. These are the meanings:

- P#V – PLSHPROF banner with version number
- P#C – Call to a subprogram (call event)
- P#R – Return from a subprogram (call event)
- P#X – Elapsed time between preceding and following events
- P#! – Comment

As you will see in the section “Using DBMS\_HPROF.analyze,” the DBMS\_HPROF.analyze function generates easier-to-decipher data and saves the data in tables.

## Using the Hierarchical Profiler Tables

- Upload the raw profiler data into the database tables.
- Run the `dbmshptab.sql` script that is located in the `ORACLE_HOME/rdbms/admin` folder to set up the profiler tables.

```
-- run this only once per schema  
-- under the schema where you want the profiler tables located  
@u01/app/oracle/product/12.2.0/dbhome_1/rdbms/admin/dbmshptab.sql
```

- Creates these tables:

Table	Description
DBMSHP_RUNS	Contains top-level information for each run command
DBMSHP_FUNCTION_INFO	Contains information on each function profiled
DBMSHP_PARENT_CHILD_INF	Contains parent-child profiler information



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

Before uploading the profiler data into the database tables, you must create the hierarchical profiler database tables. Run the `dbmshptab.sql` script under the schema where you want the profiling tables to create the hierarchical profiling tables. This script is located in your `$ORACLE_HOME/rdbms/admin/` folder.

The script creates three tables and other data structures that are required for persistently storing the profiler data.

**Note:** Running the script a second time drops any previously created hierarchical profiler tables.

## Using DBMS\_HPROF.ANALYZE

DBMS\_HPROF.analyze:

- Analyzes the raw profiler data
- Generates hierarchical profiler information in the profiler database tables
- Definition:

```
DBMS_HPROF.analyze(
    location      IN VARCHAR2,
    filename      IN VARCHAR2,
    summary_mode  IN BOOLEAN      DEFAULT FALSE,
    trace         IN VARCHAR2    DEFAULT NULL,
    skip          IN PLS_INTEGER DEFAULT 0,
    collect        IN PLS_INTEGER DEFAULT NULL,
    run_comment   IN VARCHAR2    DEFAULT NULL)
RETURN NUMBER;
```



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

Use the DBMS\_HPROF.ANALYZE function to analyze the raw profiler output and produce the hierarchical profiler information in the database.

This function accepts the following parameters:

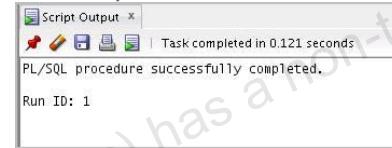
- **Location:** The name of the directory object that identifies the location from which to read
- **Filename:** The file name of the raw profiler data to be analyzed
- **summary\_mode:** A Boolean that identifies whether to generate only top-level summary information into the database tables (TRUE) or to provide detailed analysis (FALSE). The default is false.
- **Trace:** Identifies whether to analyze only the subtrees rooted at the specified trace entry or perform the analysis for the entire run. This parameter is specified in a special, qualified format within quotation marks. It includes the schema name, module name, and function name (for example "SCOTT"."PKG"."FOO").
- **Skip:** Analyzes only the subtrees rooted at the specified trace, but ignores the first “skip” invocations to trace. The default value for “skip” is 0. Used only when trace is specified.
- **Collect:** Analyzes “collect” member of invocations of traces (starting from “skip” + 1). By default, only one invocation is collected. It is used only when trace is specified.
- **run\_comment:** A comment for your run

## Using DBMS\_HPROF.ANALYZE to Write to Hierarchical Profiler Tables

- Use the DBMS\_HPROF.analyze function to upload the raw profiler results into the database tables:

```
DECLARE
  v_runid NUMBER;
BEGIN
  v_runid := DBMS_HPROF.analyze (LOCATION => 'PROFILE_DATA',
                                    FILENAME => 'pd_cc_pkg.txt');
  DBMS_OUTPUT.PUT_LINE('Run ID: ' || v_runid);
END;
```

- This function returns a unique run identifier for the run. You can use this identifier to look up results corresponding to this run from the hierarchical profiler tables.



ORACLE

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

In the example shown in the slide, the DBMS\_HPROF.ANALYZE function is used to take the raw data from the text file named pd\_cc\_pkg.txt and place the resulting analyzed data in the profiler tables. The function returns a unique identifier for the run.

## Analyzer Output from the DBMSHP\_RUNS Table

Query the DBMSHP\_RUNS table to find top-level information for each run:

```
SELECT runid, run_timestamp, total_elapsed_time
FROM dbmshp_runs
WHERE runid = 1;
```

RUNID	RUN_TIMESTAMP	TOTAL_ELAPSED_TIME
1	12-MAY-17 09.49.44.192857000 AM	140344



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

The DBMS\_HPROF.analyze PL/SQL API is used to analyze the raw profiler output and upload the results into the database tables. This enables you to query the data for custom report generation using tools, such as SQL Developer or other third-party tools.

In the example shown in the slide, the DBMSHP\_RUNS table is queried after the DBMS\_HPROF.analyze is run. This table contains top-level information for each run of the DBMS\_HPROF.analyze command. From the output, you can see the run ID, the timestamp of the run, and the total elapsed time in milliseconds. If you provided a comment for the run, you can retrieve that information from the RUN\_COMMENT column (not shown).

## Analyzer Output from the DBMSHP\_FUNCTION\_INFO Table

Query the DBMSHP\_FUNCTION\_INFO table to find information about each function profiled:

```
SELECT owner, module, type, function_line#, namespace,
       calls, function_elapsed_time
  FROM dbmshp_function_info
 WHERE runid = 1;
```

The screenshot shows the Oracle SQL Developer interface with a query result window. The title bar says 'Query Result X'. Below it, there are icons for refresh, save, and execute. The status bar indicates 'All Rows Fetched: 6 in 0.022 seconds'. The main area is a table with the following data:

OWNER	MODULE	TYPE	LINE#	NAMESPACE	CALLS	FUNCTION_ELAPSED_TIME
1 (null)	(null)	(null)	_anonymous_block	PLSQL	2	2745
2 (null)	(null)	(null)	_plsql_vm	PLSQL	2	12
3 OE	CREDIT_CARD_PKG PACKAGE BODY UPDATE_CARD_INFO			PLSQL	1	2048
4 SYS	DBMS_HPROF PACKAGE BODY STOP_PROFILING			PLSQL	1	0
5 OE	CREDIT_CARD_PKG PACKAGE BODY __static_sql_exec_line21	SQL			1	117888
6 OE	CREDIT_CARD_PKG PACKAGE BODY __static_sql_exec_line9	SQL			1	17651



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

You can query the DBMSHP\_FUNCTION\_INFO table to find summary information for each function profiled in a run of the analyzer.

In the query shown in the slide, the following information is retrieved:

- **OWNER:** Module owner of the function
- **MODULE:** The module name
- **TYPE:** The module type (such as package, package body, or procedure)
- **LINE#:** The line number in the module at which the function is defined. This line number helps to identify the source location of the function in the module and can be used by the integrated development environment (IDE) tools to navigate to the appropriate location in the source where the function is defined. The line number can also be used to distinguish between overloaded routines.
- **NAMESPACE:** The language information. At this time, SQL and PL/SQL are supported.
- **CALLS:** The number of calls to the function
- **FUNCTION\_ELAPSED\_TIME:** The time in microseconds, not including the time spent in descendant functions

## plshprof: A Simple HTML Report Generator

- `plshprof` is a command-line utility.
- You can use `plshprof` to generate simple HTML reports directly from the raw profiler data.
- The HTML reports can be browsed in any browser.
- The navigational capabilities combined with the links provide a means for you to analyze the performance of large applications.



ORACLE®

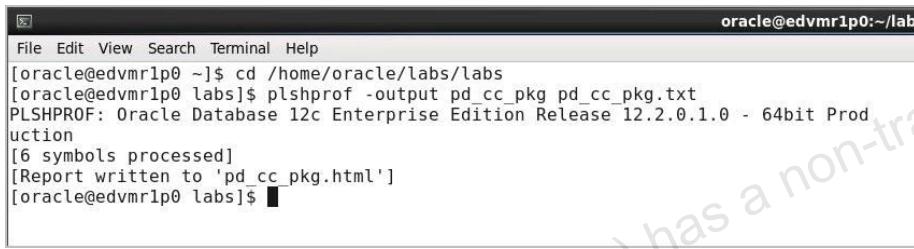
Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

`plshprof` is a command-line utility that you can use to generate HTML reports based on the raw profiler data generated by the data collection component after running the `DBMS_HPROF.ANALYZE` PL/SQL API.

## Using plshprof

After generating the raw profiler output file:

1. Change to the directory where you want the HTML output placed
2. Run `plshprof`
  - Syntax:  
`plshprof [option...] output_filename_1 output_filename_2`
  - Example:



```
oracle@edvmr1p0:~/lab
File Edit View Search Terminal Help
[oracle@edvmr1p0 ~]$ cd /home/oracle/labs/labs
[oracle@edvmr1p0 labs]$ plshprof -output pd_cc_pkg pd_cc_pkg.txt
PLSHPROF: Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production
[6 symbols processed]
[Report written to 'pd_cc_pkg.html']
[oracle@edvmr1p0 labs]$
```



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

After you have the raw profiler output file, you can generate HTML reports for that run.

The `plshprof` utility has the following options:

- **`-skip count`**: Skips the first count calls. Use only with the `-trace symbol`. The default is 0.
- **`-collect count`**: Collects information for count calls. Use only with the `-trace symbol`. The default is 1.
- **`-output filename`**: Specifies the name of the output file `symbol.html` or `tracefile1.html`
- **`-summary`**: Prints only elapsed time; no default
- **`-trace symbol`**: Specifies the function name of the tree root

To generate the HTML reports, perform the following steps:

1. Change to the directory of the raw profiler output file. In the example shown in the slide, the raw profiler file `pd_dd_pkg.txt` is in the `/home/oracle/labs/labs` folder.
2. Run the `plshprof` utility. This utility accepts line arguments. The example in the slide identifies that the output file should be named `pd_cc_pkg.html` and the original raw profiler file is named `pd_dd_pkg.txt`.

## Using plshprof

Generated files:

Name	Size	Type	Date Modified
lab_09.sql	930 bytes	SQL code	Wed 19 Apr 2017 03:21:45 AM GMT
lab_10.sql	3.6 KB	SQL code	Wed 19 Apr 2017 03:21:45 AM GMT
lab_11.sql	1.8 KB	SQL code	Wed 19 Apr 2017 03:21:47 AM GMT
lab_12.sql	635 bytes	SQL code	Wed 19 Apr 2017 03:21:47 AM GMT
lab_13.sql	1.3 KB	SQL code	Wed 19 Apr 2017 03:21:48 AM GMT
labs cleanup script.sql	356 bytes	SQL code	Wed 19 Apr 2017 03:21:41 AM GMT
<b>pd_cc_pkg.html</b>	54.2 KB	HTML document	Fri 12 May 2017 09:59:25 AM GMT
<b>pd_cc_pkg.txt</b>	750 bytes	plain text document	Fri 12 May 2017 09:49:07 AM GMT
pd_cc_pkg_zc.html	1.6 KB	HTML document	Wed 19 Apr 2017 03:21:50 AM GMT
pd_cc_pkg_2f.html	1.6 KB	HTML document	Wed 19 Apr 2017 03:21:51 AM GMT
pd_cc_pkg_2n.html	1.3 KB	HTML document	Wed 19 Apr 2017 03:21:51 AM GMT
pd_cc_pkg_fn.html	4.6 KB	HTML document	Wed 19 Apr 2017 03:21:53 AM GMT
pd_cc_pkg_md.html	5.6 KB	HTML document	Wed 19 Apr 2017 03:21:53 AM GMT
pd_cc_pkg_mf.html	5.6 KB	HTML document	Wed 19 Apr 2017 03:21:54 AM GMT
pd_cc_pkg_ms.html	5.6 KB	HTML document	Wed 19 Apr 2017 03:21:54 AM GMT
pd_cc_pkg_nc.html	1.0 KB	HTML document	Wed 19 Apr 2017 03:21:55 AM GMT



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

Running the `plshprof` utility generates a set of HTML files. `filename.html` is the root page where you start browsing. The other files with the same file name prefix are hyperlinks from the base `filename.html` file. In the example in the slide, the root page is named `pd_cc_pkg.html`.

## Using plshprof

3. Open pd\_cc\_pkg.html in a browser:

The screenshot shows a Mozilla Firefox window with the title "PL/SQL Elapsed Time (microsecs) Analysis for 'pd\_cc\_pkg' - Mozilla Firefox". The address bar displays "file:///home/oracle/abs/abs/pd\_cc\_pkg.html". The main content area is titled "PL/SQL Elapsed Time (microsecs) Analysis" and contains the following text:  
140344 microsecs (elapsed time) & 8 function calls  
The PL/SQL Hierarchical Profiler produces a collection of reports that present information derived from the profiler's output log in a variety of formats. The following reports have been found to be the most generally useful as starting points for browsing:

- Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)
- SQL ID Elapsed Time (microsecs) Data sorted by SQL ID

In addition, the following reports are also available:

- Function Elapsed Time (microsecs) Data sorted by Function Name
- Function Elapsed Time (microsecs) Data sorted by Total Descendants Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Function Elapsed Time (microsecs) Data sorted by Mean Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Mean Function Elapsed Time (microsecs)



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

3. You can open the start page file in any browser. In this example, the pd\_cc\_pkg.html file is opened. This is an example of a single-run report. It contains some overall summary information and hyperlinks to additional information.

## Using the HTML Reports

derived from the profiler's output log in a variety of formats. The following reports have been found to be the most generally useful as starting points for browsing:

- Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)
- Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)
- SQL ID Elapsed Time (microsecs) Data sorted by SQL ID

The screenshot shows a Mozilla Firefox browser window with the title bar "PL/SQL Elapsed Time (microsecs) Analysis for 'pd\_cc\_pkg' - Mozilla Firefox". The address bar contains the URL "file:///home/oracle/labs/labs/pd\_cc\_pkg.html#hprof\_ts". The main content area displays a table titled "Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)". The table has the following columns: Subtree, Ind%, Function, Ind%, Descendants, Ind%, Calls, Ind%, Function Name, SQL ID, and SQL TEXT. The data in the table is as follows:

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name	SQL ID	SQL TEXT
140344	100%		12	0.0%	140332	100%	2	25.0%	<a href="#">plsql_vm</a>	
140332	100%	2745	2.0%		137587	98.0%	2	25.0%	<a href="#">anonymous_block</a>	
137587	98.0%	2048	1.5%		135539	96.6%	1	12.5%	<a href="#">OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO (Line 3)</a>	
117888	84.0%	117888	84.0%		0	0.0%	1	12.5%	<a href="#">OE.CREDIT_CARD_PKG._static_sql_exec_line21 (Line 21)</a>	82nnmrylyp117 UPDATE CUSTOMERS SET CREDIT CARDS = TYP CR CARD NS
17651	12.6%	17651	12.6%		0	0.0%	1	12.5%	<a href="#">OE.CREDIT_CARD_PKG._static_sql_exec_line9 (Line 9)</a>	3zjmmcb1j06t6 SELECT CREDIT CARDS FROM CUSTOMERS WHERE CUSTOMER
	0	0.0%	0	0.0%	0	0.0%	1	12.5%	<a href="#">SYS.DBMS_HPROF.STOP_PROFILING (Line 453)</a>	



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

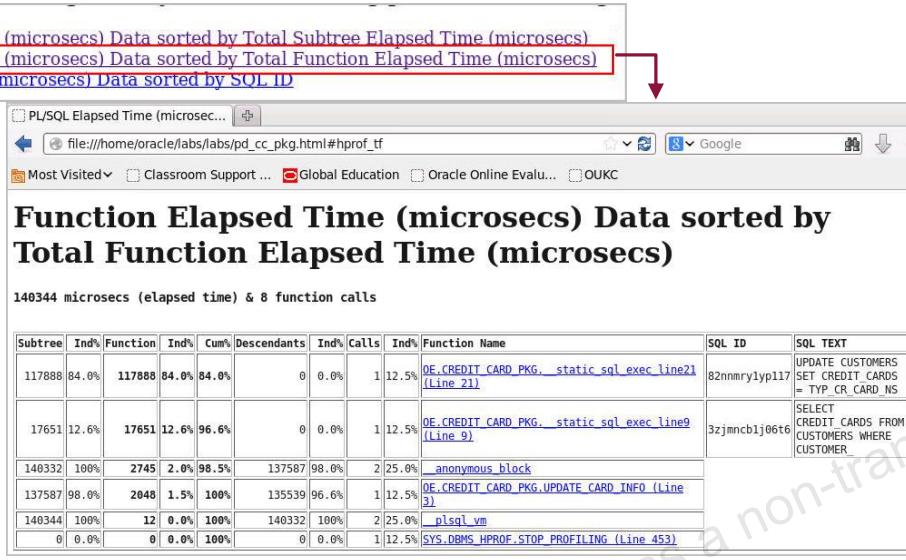
The function-level summary reports provide you with a flat view of the profile information. The reports list the total number of calls, self time, subtree time, and descendants for each function. Each report is sorted on a particular attribute.

Note that in the example in the slide, each function name is hyperlinked to its corresponding parents and children report. The column that is in bold identifies how the report is sorted.

If a subprogram is nested, the profiler reports display the fully qualified name, such as OE.P.A.B; that is, procedure B is nested within procedure A of package OE.P.

## Using the HTML Reports

• Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)  
 • Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)  
 • SQL ID Elapsed Time (microsecs) Data sorted by SQL ID



**Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)**

140344 microsecs (elapsed time) & 8 function calls

Subtree	Ind%	Function	Ind%	Cum%	Descendants	Ind%	Calls	Ind%	Function Name	SQL ID	SQL TEXT
117888	84.0%	117888	84.0%	84.0%	0	0.0%	1	12.5%	OE.CREDIT_CARD_PKG..static_sql_exec_line21 (Line 21)	82nnmry1ypl17	UPDATE CUSTOMERS SET CREDIT_CARDS = TYP CR CARD NS
17651	12.6%	17651	12.6%	96.6%	0	0.0%	1	12.5%	OE.CREDIT_CARD_PKG..static_sql_exec_line9 (Line 9)	3zjmncbjj06t6	SELECT CREDIT_CARDS FROM CUSTOMERS WHERE CUSTOMER
140332	100%	2745	2.0%	98.5%	137587	98.0%	2	25.0%	_anonymous_block		
137587	98.0%	2048	1.5%	100%	135539	96.6%	1	12.5%	OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO (Line 31)		
140344	100%	12	0.0%	100%	140332	100%	2	25.0%	plsql_vm		
0	0.0%	0	0.0%	100%	0	0.0%	1	12.5%	SYS.DBMS_HPROF_STOP_PROFILING (Line 453)		



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

You can use the module-level summary report to find information about each PL/SQL module, including the total time spent in the module and the total number of calls to functions in the module. The total time spent in a module is calculated by adding the self times for all functions in that module.

## Using the HTML Reports

The screenshot shows a web browser displaying an HTML report titled "Namespace Elapsed Time (microsecs) Data sorted by Namespace". The report lists three options in a menu bar:

- Namespace Elapsed Time (microsecs) Data sorted by Namespace
- Namespace Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Parents and Children Elapsed Time (microsecs) Data

A red arrow points from the second option down to the main content area of the report. The content area includes the title, a summary of 140344 microseconds and 8 function calls, and a table showing the distribution of calls by namespace:

Function	Ind%	Calls	Ind%	Namespace
4805	3.4%	6	75.0%	PLSQL
135539	96.6%	2	25.0%	SQL

Each function tracked by the profiler is associated with a namespace. PL/SQL functions, procedures, triggers, and packages fall under the category “PLSQL” namespace. Operations corresponding to SQL statement execution from PL/SQL, such as UPDATE, SELECT, FETCH, and EXECUTE IMMEDIATE, fall under the “SQL” namespace. The namespace-level summary report provides information about the total time spent in that namespace and the total number of calls to functions in that namespace.

## Quiz



Select the correct order of the five steps to trace PL/SQL code using the `dbms_trace` package:

- A. Enable specific program units for trace data collection.
  - B. Use `DBMS_TRACE.clear_plsql_trace` to stop tracing data.
  - C. Run your PL/SQL code.
  - D. Read and interpret the trace information.
  - E. Use `DBMS_TRACE.set_plsql_trace` to identify a trace level.
- a. A, E, C, B, D
  - b. A, B, C, D, E
  - c. A, C, E, B, D
  - d. A, E, C, D, B



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

**Answer: a**

## Quiz



You can view the hierarchical profiler function level summaries that include:

- a. Number of calls to a function
- b. Time spent in the function itself
- c. Time spent in the entire subtree under the function
- d. Detailed parent-children information for each function



ORACLE

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

**Answer: a, b, c, d**

## Quiz



Use the \_\_\_\_\_ and the \_\_\_\_\_ to find information about each function profiled by the PL/SQL profiler.

- a. DBMS\_HPROF.ANALYZE table
- b. DBMSHP\_RUNS table
- c. DBMSHP\_FUNCTION\_INFO table
- d. plshprof command-line utility

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

ORACLE



**Answer: a, d**

## Summary

In this lesson, you should have learned how to:

- Trace PL/SQL program execution
- Profile PL/SQL applications



ORACLE

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

This lesson showed you how to trace PL/SQL program execution by using the `DBMS_TRACE` package, how to profile your PL/SQL application, and, with profiling, how to set up the profiler tables, generate raw data, and use `DBMS_HPROF` to analyze raw data in the profiler tables.

## Practice 11: Overview

This practice covers hierarchical profiling of PL/SQL code.



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

Using the OE application that you have created, write code to profile components in your application.  
Use the OE schema for this practice.

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

PELISSIER LOIC (loic.pelissier@gmail.com) has a non-transferable  
license to use this Student Guide.

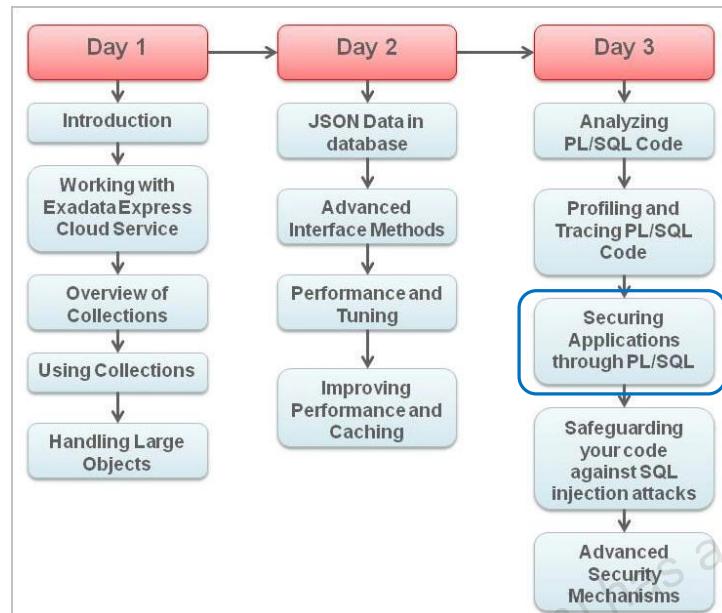
12

# Securing Applications through PL/SQL

ORACLE

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

## Course Agenda



ORACLE

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



In this lesson, you are introduced to Collections. Collections are composite data types, you can store values of data which can in turn have various internal components. In earlier course ( Oracle database 12c: PL/SQL Program Units) you must've learnt about records. Records are also composite data types.

A collection stores a set of values of same data type, whereas a record stores a set of values of different data types. In this lesson we will discuss different variants of collections.

## Objectives

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

- Implement invoker's rights on program units.
- Understand various access control mechanisms.
- Define security policies on applications.
- Implement Virtual Private Databases.



ORACLE

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

In this lesson, you learn about the security features in the Oracle Database from an application developer's standpoint.

For more information about these features, refer to *Oracle Supplied PL/SQL Packages and Types Reference*, *Oracle Label Security Administrator's Guide*, *Oracle Single Sign-On Application Developer's Guide*, and *Oracle Security Overview*.

## Lesson Agenda

- Invoker's rights and Definer's rights on program units
- Controlling access to program units
- Application Security Policy
- Application Context
- Virtual Private Database



ORACLE®

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

## Invoker's Rights and Definer's Rights

Who is an invoker?

An user trying to execute a program unit created by some other user.



Who is a definer?

An user who created the program unit.



What are invoker's rights?

Access privileges of an user who is invoking a program unit created by other users.

What are definer's rights?

Access privileges of an user who has created the program unit.

ORACLE®

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

The owner of a procedure, called the definer, must have the necessary object privileges for objects that the procedure references.

The user of a procedure who is not its owner/definer is called the *invoker*. The invoker of a procedure should have access privileges to the objects referenced in the procedure.

When you define a procedure to execute with definer's rights, the invoker executing the procedure need not request for access to the objects referenced in the procedure. If you define a procedure to be executed with invoker's rights, then the invoker requires additional privileges on referenced objects.

## Why Invoker's Rights?

- Invoker's rights lets you reuse code and centralize application logic.
- You can restrict access to sensitive data.
- You can create one instance of the procedure, and many users can call it to access their own data.
- You can use the same procedure to execute on different schemas.



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

Definer's rights and invoker's rights are used to control access to the privileges necessary to run a user-created procedure, or program unit.

In a definer's rights procedure, the procedure executes with the privileges of the owner. An invoker's rights procedure executes with the privileges of the current user, that is, the user who invokes the procedure.

An user of an invoker's rights procedure must have privileges (granted to the user either directly or through a role) on objects that the procedure accesses through external references that are resolved in the schema of the invoker.

By default, stored procedures and SQL methods execute with the privileges of their owner, not their current user. Such definer's rights subprograms are bound to the schema in which they reside, allowing you to refer to objects in the same schema without qualifying their names. For example, if schemas `HR` and `OE` both have a table called `departments`, a procedure owned by `HR` can refer to `departments` rather than `HR.departments`. If user `OE` calls `HR`'s procedure, the procedure still accesses the `departments` table owned by `HR`.

You can create one instance of the procedure, and many users can call it to access their own data provided they have the required privileges. Invoker's rights subprograms are not bound to a particular schema.

## AUTHID clause

You can define a procedure to be executed as invoker's rights or definer's rights using AUTHID clause:

```
CREATE OR REPLACE PROCEDURE print_customers
AUTHID CURRENT_USER
AS
. . .
```



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

To implement invoker's rights, use the AUTHID clause. AUTHID clause specifies whether a subprogram executes with the privileges of its owner or its current user. It also specifies whether external references (that is, references to objects outside the subprogram) are resolved in the schema of the owner or the current user.

The AUTHID clause is allowed only in the header of a standalone subprogram, a package spec, or an object type spec. In the CREATE FUNCTION, CREATE PROCEDURE, CREATE PACKAGE, or CREATE TYPE statement, you can include either AUTHID CURRENT\_USER or AUTHID DEFINER immediately before the IS or AS keyword that begins the declaration section.

DEFINER is the default option. In a package or object type, the AUTHID clause applies to all subprograms.

Most supplied PL/SQL packages (such as DBMS\_LOB, DBMS\_PIPE, DBMS\_ROWID, DBMS\_SQL and UTL\_REF) are invoker's rights packages.

## Lesson Agenda

- Invoker's rights and Definer's rights on program units
- Controlling access to program units
- Application Security Policy
- Application Context
- Virtual Private Database

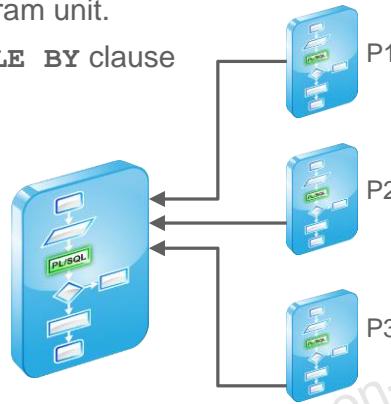


ORACLE®

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

## White Lists

- A list of program units that can access given program unit.
- You specify a white list while creating the program unit.
- The list of program units follow the **ACCESSIBLE BY** clause while creating the program unit.



ORACLE

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

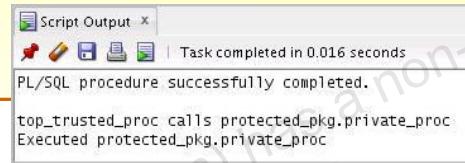
Prior to 12c, a PL/SQL program unit could be invoked by all other program units of the schema. In 12c, you have white lists which restrict the access of a program unit. A program unit in the schema cannot access every other program unit in the given schema. While defining a program unit, you specify the white list of program units that can access or invoke the current program unit.

You define the white list using **ACCESSIBLE BY** clause.

## ACCESSIBLE BY Clause

- You can specify the white list of a program unit with **ACCESSIBLE BY** clause.

```
CREATE OR REPLACE PROCEDURE top_protected_proc ACCESSIBLE BY (PROCEDURE
  top_trusted_proc) AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('Executed top_protected_proc.');
END;
CREATE OR REPLACE PROCEDURE top_trusted_proc AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('top_trusted_proc calls top_protected_proc');
  top_protected_proc;
END;
EXECUTE top_trusted_proc;
/
```



ORACLE

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

You can include a clause in the header of your program unit that specifies a “whitelist” of other PL/SQL units that can access the PL/SQL unit you are creating.

The code in the slide shows two procedures `top_protected_proc` and `top_trusted_proc`. The `protected_proc` procedure is defined to be accessed by only `top_trusted_proc`. This implies that you cannot access `top_protected_proc` directly.

You can see the output, when you execute `top_trusted_proc` in the slide.

## Using ACCESSIBLE BY Clause in Packages

```
CREATE OR REPLACE PACKAGE protected_pkg
AS
  PROCEDURE public_proc;
  PROCEDURE private_proc ACCESSIBLE BY (PROCEDURE top_trusted_proc);
END;
CREATE OR REPLACE PACKAGE BODY protected_pkg
AS
  PROCEDURE public_proc AS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Executed protected_pkg.public_proc');
  END;
  PROCEDURE private_proc ACCESSIBLE BY (PROCEDURE top_trusted_proc) AS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Executed protected_pkg.private_proc');
  END;
END;
```



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

The code in the slide shows a procedure `private_proc` in the package `protected_pkg` is defined to be accessed by the procedure `top_trusted_proc`, which is defined outside the package.

Except for the `private_proc` procedure other procedures of the package can be accessed by program units accessing the package.

```
CREATE OR REPLACE PROCEDURE top_trusted_proc
AS
BEGIN
  DBMS_OUTPUT.PUT_LINE('top_trusted_proc calls
                      protected_pkg.private_proc ');
  protected_pkg.private_proc;
END;

. . .

EXECUTE protected_pkg.private_proc
```

## Lesson Agenda

- Invoker's rights and Definer's rights on program units
- Controlling access to program units
- Application Security Policy
- Application Context
- Virtual Private Database



ORACLE®

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

## What Is an Application Security Policy?

- Applications have a list of security requirements and rules to regulate user access to database objects.
- These rules are generally a result of the context in which the application is deployed.

An application can be accessed by various types of users such as developers, marketing executives, target audience and some malicious users. You have to define an application security policy, that can allow access to the right users and deny access to intruders.



ORACLE

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

An application security policy is a list of application security requirements and rules that regulate user access to database objects.

Creating an application security policy is the first step to create a secure database application. You should draft security policies for each database application. For example, each database application should have one or more database roles that provide different levels of security when executing the application. You then can grant the database roles to other roles or directly to specific users.

Applications that can potentially allow unrestricted SQL statement processing (through tools such as SQL\*Plus or SQL Developer) also need security policies that prevent malicious access to confidential or important schema objects.

## Implementing Application Security Policy

- Define roles in the application with appropriate access privileges.
- Secure passwords in application design.
- Secure external procedures through authentication.
- Use DBMS\_RLS package to add, modify and remove security policies.



ORACLE

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

Create a role for each application and grant that role all the privileges a user must run the application. This will simplify privilege management. An application can have several roles, each granted a specific subset of privileges that allow greater or lesser capabilities while running the application. You can determine which users have privileges on which applications by querying the DBA\_ROLE\_PRIVS data dictionary view.

You have to secure the passwords used for authentication in the application against platform specific threats by using password encryption and other such techniques.

All these security mechanisms can primarily be implemented through application administration. Apart from defining security roles and implementing password management, you can define policies for database object access.

## DBMS\_RLS package

- DBMS\_RLS package provides fine-grained access control administrative interface for developers.
- Implements security rules through dynamic predicates.
- The policy function generates the predicates based on the session environment variables available during the function call.
- The session environment variables are available in the form of application contexts.



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

DBMS\_RLS package has a set of PL/SQL program units, which enable developers to define, modify security policies. It provides an interface that developers can use to perform administrative tasks.

The functionality to support fine-grained access control is based on dynamic predicates, where security rules are not embedded in views, but are acquired at the statement parse time, when the base table or view is referenced in a DML statement.

The policy function may generate the predicates based on the session environment variables available during the function call. These variables usually appear in the form of application contexts. We will discuss application contexts further in the lesson.

## Defining a Policy

- To define a policy for a specific user, you should have administration privileges on that user.
  - To define a policy for OE user, you have to be an user such as SYS.
- You should define a policy function which details the functionality of the policy.
- Define the policy using the `DBMS_RLS.ADD_POLICY` procedure.



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

You can define security policies for a certain user only when you have administration privileges on that user. If you want to define a policy for the `OE` user, then you've to a `SYS` user or someone with admin privileges on `OE`.

The implementation of the security policy is defined through a function.

Use the `DBMS_RLS.ADD_POLICY` procedure to define the policy. You can use other procedures of `DBMS_RLS` package to perform various operations on the security policies.

## Defining a Policy Function

- Let's define a security policy on OE user through this function.

```
CREATE OR REPLACE FUNCTION auth_orders(
    schema_var IN VARCHAR2,
    table_var IN VARCHAR2 )
RETURN VARCHAR2
IS
    return_val VARCHAR2 (400);
BEGIN
    return_val := 'SALES REP ID = 159';
    RETURN return_val;
END auth_orders;
/
```



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

A security policy defined will have an associated function, which has the details of the implementation of the policy. The slide shows a policy function that limits access to orders created by Sales Representative 159 in the OE.ORDERS table.

## Defining a Policy

- Define a policy using DBMS\_RLS.ADD\_POLICY procedure.

```
BEGIN DBMS_RLS.ADD_POLICY (
object_schema => 'oe',
object_name => 'orders',
policy_name => 'orders_policy',
function_schema => 'sys',
policy_function => 'auth_orders',
statement_types => 'select' );
END;
/
```



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

The code in the slide defines a policy on the schema object ORDERS in the schema OE.

The policy is named to be `orders_policy` and is defined by the SYS schema through `auth_orders` function.

The policy is defined for only SELECT statements. As a result of this policy, whenever you execute a SELECT statement on ORDERS table as 'OE' user, a dynamic predicate WHERE sales\_rep\_id = 159 will be added to the SELECT statement.

To see the policy in action, you login as 'OE' user and execute:

```
SELECT COUNT(*) FROM ORDERS;
```

The output of this query will be equivalent to the output of

```
SELECT COUNT(*) FROM ORDERS WHERE SALES REP_ID = 159;
```

## Lesson Agenda

- Invoker's rights and Definer's rights on program units
- Controlling access to program units
- Application Security Policy
- Application Context
- Virtual Private Database

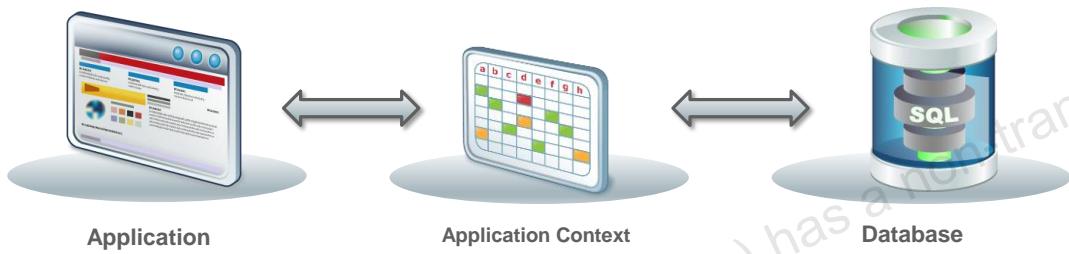


ORACLE®

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

## Application Context - Concept

- Consider a situation where a customer is accessing the OE application and intends to access the orders placed.
- The application is expected to show only the orders placed by the particular customer, instead of all the orders.
- You can define an application context which holds the customer\_id and use it while retrieving data from the database during the user session.



ORACLE®

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

Consider the order entry application, when you use a table containing the columns ORDER\_ID and CUSTOMER\_ID, you can use the values in these columns as security attributes to restrict access by a customer to his or her own orders, based on the ID of that customer. If a customer with certain customer\_id logs into the application, the value of the customer\_id is saved in the application context and used for the entire application session.

You have to filter the orders based on the customer\_id of the customer.

Instead of writing a new SQL query for each and every customer accessing the application, you can pick the details from the application context.

## Application Context - Implementation

- The application context is a set of **name-value** pairs that Oracle Database stores in memory.
- It's an associative array, the **name** points to a location in memory that holds the **value**.
- An application can use the application context to access session information about a user.
- You can then use this information to either permit or prevent the user from accessing data through the application.



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

The application context is a set of name-value pairs that Oracle Database stores in memory. The application context has a label called a namespace (for example, `empno_ctx` for an application context that retrieves employee IDs).

Inside the context are the name-value pairs (an associative array): the name points to a location in memory that holds the value.

An application can use the application context to access session information about a user, such as the user ID or other user-specific information, or a client ID, and then securely pass this data to the database.

You can then use this information to either permit or prevent the user from accessing data through the application. You can use application contexts to authenticate both database and non-database users.

Most applications contain information about the basis on which access is to be limited. In an order entry application, for example, you limit the customers' access to their own orders (`ORDER_ID`) and customer number (`CUSTOMER_ID`). Or, you may limit account managers (`ACCOUNT_MGR_ID`) to view only their own customers. These values can be used as security attributes. Your application can use a context to set values that are accessed within your code and used to generate `WHERE` clause predicates for fine-grained access control.

An application context is owned by `SYS`. Users cannot change their application's context.

## USERENV Application Context

- USERENV is a namespace which holds the data of the current session.
- USERENV holds the context of the current session.

```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER')
FROM DUAL;
```

Query Result X	
SYS_CONTEXT('USERENV','SESSION_USER')	
1	OE

```
SELECT SYS_CONTEXT ('USERENV', 'DB_NAME')
FROM DUAL;
```

Query Result X	
SYS_CONTEXT('USERENV','DB_NAME')	
1	PDBORCL



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

USERENV is the namespace which holds the context information of the current session. You can use the SYS\_CONTEXT function to access the context information of a session.

SYS\_CONTEXT returns the value of parameter associated with the context namespace at the current instant. You can use this function in both SQL and PL/SQL statements.

## Creating an Application Context

```
CREATE [OR REPLACE] CONTEXT namespace
USING [schema.]plsql_package
```

- Requires the CREATE ANY CONTEXT system privilege
- Parameters:
  - *namespace* is the name of the context.
  - *schema* is the name of the schema owning the PL/SQL package.
  - *plsql\_package* is the name of the package used to set or modify the attributes of the context. (It does not need to exist at the time of context creation.)

```
CREATE OR REPLACE CONTEXT order_ctx USING oe.orders_app_pkg;
```



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

For fine-grained access where you want account managers to view only their customers, customers to view only their information, and sales representatives to view only their orders, you can create a context called ORDER\_CTX and define for it the ACCOUNT\_MGR, CUST\_ID and SALE\_REP attributes, respectively.

Because a context is associated with a PL/SQL package, you need to name the package that you are associating with the context. This package does not need to exist at the time of context creation.

## Setting a Context

- Use the supplied package procedure DBMS\_SESSION.SET\_CONTEXT to set a value for an attribute within a context.

```
DBMS_SESSION.SET_CONTEXT('context_name',
                          'attribute_name',
                          'attribute_value')
```

- Set the attribute value in the package that is associated with the context.

```
CREATE OR REPLACE PACKAGE orders_app_pkg
...
BEGIN
    DBMS_SESSION.SET_CONTEXT('ORDER_CTX',
                            'ACCOUNT_MGR',
                            v_user)
...

```



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

When a context is defined, you can use the DBMS\_SESSION.SET\_CONTEXT procedure to set a value for an attribute within a context. The attribute is set in the package that is associated with the context.

```
CREATE OR REPLACE PACKAGE orders_app_pkg
IS
    PROCEDURE set_app_context;
END;
/
CREATE OR REPLACE PACKAGE BODY orders_app_pkg
IS
    c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';
    PROCEDURE set_app_context
    IS
        v_user VARCHAR2(30);
    BEGIN
        SELECT user INTO v_user FROM dual;
        DBMS_SESSION.SET_CONTEXT
            (c_context, 'ACCOUNT_MGR', v_user);
    END;
END;
/
```

In the example on the previous page, the ORDER\_CTX context has the ACCOUNT\_MGR attribute set to the current user logged (determined by the USER function).

For this example, assume that users AM145, AM147, AM148, and AM149 exist. As each user logs on and the DBMS\_SESSION.SET\_CONTEXT is invoked, the attribute value for that ACCOUNT\_MGR is set to the user ID.

```
GRANT EXECUTE ON oe.orders_app_pkg  
TO AM145, AM147, AM148, AM149;
```

```
Enter user-name: AM147/oracle@pdborcl  
Last Successful login time: Fri Jun 16 2017 06:49:14 +00:00  
  
Connected to:  
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production  
  
SQL> EXECUTE oe.orders_app_pkg.set_app_context;  
  
PL/SQL procedure successfully completed.  
  
SQL> SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;  
  
SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')  
-----  
AM147  
  
SQL> ■
```

If you switch the user ID, the attribute value is also changed to reflect the current user.

```
Enter user-name: AM145/oracle@pdborcl  
  
Connected to:  
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production  
  
SQL> EXECUTE oe.orders_app_pkg.set_app_context;  
  
PL/SQL procedure successfully completed.  
  
SQL> SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;  
  
SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')  
-----  
AM145  
  
SQL> ■
```

## Lesson Agenda

- Invoker's rights and Definer's rights on program units
- Controlling access to program units
- Application Security Policy
- Application Context
- Virtual Private Database



ORACLE®

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

## Virtual Private Database

Consider a scenario where you are a sales representative with `sales_rep_id 159`:

- All the orders you manage are stored in the database.
- You are given an user id and password credentials to login to the database and have a look at your orders.
- You can view only view your orders when you log into the database.
- Your perspective is that the database has only your orders, however the database actually has all the orders which includes orders managed by other sales representatives.
- That's **Virtual Private Database**.



ORACLE

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

## Virtual Private Database

- A Virtual Private Database is a subset of a large database retrieved after applying database security policies.
- Using the concepts of application context and application security policy you can filter relevant data from the database.



ORACLE®

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

## Implementing a Virtual Private Database

Following are the steps of implementing a VPD through a package:

1. Setting up a driving application context.
2. Creating a package.
3. Defining a security policy.
4. Setting up a logon trigger.
5. Testing the security policy.



ORACLE

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

You can create VPD using a simple security policy as we did earlier or you can implement a security policy through a package.

1. You have to setup an application context first. While doing this, you define the user environment say the user roles and user credentials. The access to certain data is granted based on this user environment.
2. While defining an application policy, you can define a policy function or a package.
3. Define security policy, say which user can access what data is determined through the security policy.
4. When a user logs in with his/her credentials, then the application context has to be set with values. Based on the application context appropriate security policies have to be enforced. To track this event of user login and apply appropriate security policies you can use a logon trigger.
5. You login as a user on which the security policy is defined and test whether the security policy is applied or not.

## Setting Up a Context

Set up a driving context.

```
CREATE OR REPLACE CONTEXT order_ctx  
USING orders_app_pkg;
```



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

## Creating the Package

Create the package associated with the context that you have defined. In the package:

- a. Set the context
- b. Define the predicate

```
CREATE OR REPLACE PACKAGE orders_app_pkg
IS
  PROCEDURE show_app_context;
  PROCEDURE set_app_context;
  FUNCTION the_predicate
    (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2;
END orders_app_pkg;      -- package spec
/
```



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

In the OE schema, the ORDERS\_APP\_PKG is created. This package contains three routines:

- **show\_app\_context:** For learning and testing purposes, this procedure displays a context attribute and value.
- **set\_app\_context:** This procedure sets a context attribute to a specific value.
- **the\_predicate:** This function builds the predicate (the WHERE clause) that controls the rows that are visible in the CUSTOMERS table to a user. (Note that this function requires two input parameters. An error occurs when the policy is implemented if you exclude these two parameters.)

```

CREATE OR REPLACE PACKAGE BODY orders_app_pkg
IS
    c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';
    c_attrib  CONSTANT VARCHAR2(30) := 'ACCOUNT_MGR';

PROCEDURE show_app_context
IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('Type: ' || c_attrib ||
        ' - ' || SYS_CONTEXT(c_context, c_attrib));
END show_app_context;

PROCEDURE set_app_context
IS
    v_user VARCHAR2(30);
BEGIN
    SELECT user INTO v_user FROM dual;
    DBMS_SESSION.SET_CONTEXT
        (c_context, c_attrib, v_user);
END set_app_context;

FUNCTION the_predicate
(p_schema VARCHAR2, p_name VARCHAR2)
RETURN VARCHAR2
IS
    v_context_value VARCHAR2(100) := 
        SYS_CONTEXT(c_context, c_attrib);
    v_restriction VARCHAR2(2000);
BEGIN
    IF v_context_value LIKE 'AM%' THEN
        v_restriction :=
            'ACCOUNT_MGR_ID =
                SUBSTR(''' || v_context_value || ''', 3, 3)';
    ELSE
        v_restriction := null;
    END IF;
    RETURN v_restriction;
END the_predicate;

END orders_app_pkg; -- package body
/

```

Note that the THE\_PREDICATE function builds the WHERE clause and stores it in the V\_RESTRICTION variable. If the SYS\_CONTEXT function returns an attribute value that starts with AM, the WHERE clause is built with ACCOUNT\_MGR\_ID = *the last three characters of the attribute value*. If the user is AM145, the WHERE clause will be:

```
WHERE account_mgr_id = 145
```

## Define the Security Policy

- To define a security policy use the ADD\_POLICY procedure of DBMS\_RLS package.

```

DECLARE
BEGIN
    DBMS_RLS.ADD_POLICY (
        'OE',          Object schema
        'CUSTOMERS',   Table name
        'OE_ACCESS_POLICY', Policy name
        'SYS',          Function schema
        'ORDERS_APP_PKG.THE_PREDICATE', Policy function
        'SELECT, UPDATE, DELETE', Statement types
        FALSE,          Update check
        TRUE);          Enabled
    END;
/

```



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

The OE\_ACCESS\_POLICY security policy is created and added with the DBMS\_RLS.ADD\_POLICY procedure. The predicate function that defines how the policy is to be implemented is associated with the policy being added.

Here is the description of the parameters of

Parameter	Description
OBJECT_SCHEMA	Schema containing the table or view (logon user, if NULL)
OBJECT_NAME	Name of the table or view to which the policy is added
POLICY_NAME	Name of the policy to be added. For any table or view, each POLICY_NAME must be unique.
FUNCTION_SCHEMA	Schema of the policy function (logon user, if NULL)
POLICY_FUNCTION	Name of the function that generates a predicate for the policy. If the function is defined within a package, the name of the package must be present.
STATEMENT_TYPES	Statement types that the policy will apply. It can be any combination of SELECT, INSERT, UPDATE, and DELETE. The default is to apply all these statement types to the policy.
UPDATE_CHECK	Optional argument for the INSERT or UPDATE statement types. The default is FALSE. Setting update_check to TRUE causes the server to also check the policy against the value after INSERT or UPDATE.
ENABLE	Indicates whether the policy is enabled when it is added. The default is TRUE.
SEC_RELEVANT_COLS SEC_RELEVANT_COLS_OPT	Enable getting all rows from the table. However, for the relevant columns users can see only their own data; in other columns they can see NULL values.

## Setting Up the Logon Trigger

Create a database trigger that executes whenever anyone logs on to the database:

```
CREATE OR REPLACE TRIGGER set_id_on_logon
AFTER logon on DATABASE
BEGIN
  oe.orders_app_pkg.set_app_context;
END;
/
```



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

The logon trigger sets the context for the user.

## Policy in Action

To see the policy in action

```
-- Execute as SYS user
SELECT COUNT(*), account_mgr_id
FROM customers
GROUP BY account_mgr_id;
```

COUNT(*)	ACCOUNT_MGR_ID
1	76
2	74
3	58
4	111

```
-- Execute as AM148 in sqlplus
SELECT COUNT(*) from oe.customers;
```

COUNT(*)
-----
58

ORACLE

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

```
[oracle@edvmr1p0 ~]$ sqlplus

SQL*Plus: Release 12.2.0.1.0 Production on Tue Jun 20 04:10:26 2017

Copyright (c) 1982, 2016, Oracle. All rights reserved.

Enter user-name: AM148/oracle@pdborcl
Last Successful login time: Tue Jun 20 2017 04:00:33 +00:00

Connected to:
Oracle Database 12c Enterprise Edition Release 12.2.0.1.0 - 64bit Production

SQL> select count(*) from oe.customers;

COUNT(*)
-----
58
```

## Data Dictionary Views

View	Description
USER_POLICIES	All policies owned by the current schema
ALL_POLICIES	All policies owned or accessible by the current schema
DBA_POLICIES	All policies in the database (Its columns are the same as those in ALL_POLICIES.)
ALL_CONTEXT	All active context namespaces defined in the session
DBA_CONTEXT	All context namespace information (active and inactive)



ORACLE®

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

You can query the data dictionary views to find information about the policies available in your schema.

## Using the ALL\_CONTEXT Dictionary View

Use ALL\_CONTEXT to see the active context namespaces defined in your session:

```
CONNECT AS AM148  
  
SELECT *  
FROM all_context;
```

NAMESPACE	SCHEMA	PACKAGE
ORDER_CTX	OE	ORDERS_APP_PKG

ORACLE

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

You can use the ALL\_CONTEXT dictionary view to view information about the contexts to which you have access. In the slide, the NAMESPACE column is equivalent to the context name.

## Using the ALL\_CONTEXT Dictionary View

Use ALL\_POLICIES to view information about the policies to which you have access:

```
SELECT object_name, policy_name, pf_owner,
       package, function, sel, ins, upd, del
  FROM ALL_POLICIES;
```

OBJECT_NAME	POLICY_NAME	PF_OWNER	PACKAGE	FUNCTION	SEL	INS	UPD	DEL
CUSTOMERS	OE_ACCESS_POLICY	SYS	ORDERS_APP_PKG	THE_PREDICATE	YES	NO	YES	YES

ORACLE

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

You can use the ALL\_POLICIES dictionary view to view information about the policies to which you have access. In the example in the slide, information is shown about the OE\_ACCESS\_POLICY policy.

## Policy Groups

- A set of security policies defined with respect to an application.
- A policy group is set up by the administrator by defining an application context.
- DBMS\_RLS package has various procedures which can be used to create policy groups.
- SYS\_DEFAULT is the default policy group:



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

You may want to enforce different security policies depending on the application that is accessing data. Consider a situation in which two applications, Order Entry and Inventory, both access the orders table. You may want to have the Inventory application use a policy that limits access based on type of product. At the same time, you may want to have the Order Entry application use a policy that limits access based on customer number.

In this case, you must partition the use of fine-grained access by application. Otherwise, both policies would be automatically concatenated together, which may not be the result that you want. You can specify two or more policy groups, and a driving application context that determines which policy group is in effect for a given transaction. You can also designate default policies that always apply to data access.

The process of defining a policy group is similar to that of creating a policy, by using DBMS\_RLS.ADD\_GROUP\_POLICY procedure.

You use policy groups when you want to a set of policies to compositely implement fine grained access.

## Quiz



Which of the following statements is *not* true about fine-grained access control?

- a. Fine-grained access control enables you to enforce security through a low level of granularity.
- b. Fine-grained access control restricts users to viewing only “their” information.
- c. Fine-grained access control is implemented through a security policy attached to tables.
- d. To implement fine-grained access control, hard code the security policy into the user code.

**ORACLE**

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



**Answer: d**

## Quiz



Application context for a user is fixed and does not change with change of session.

- a. True
- b. False

ORACLE

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

**Answer: b**

## Quiz



Arrange the following steps used to implement a security policy:

- A. Define the policy.
  - B. Create the package associated with the context.
  - C. Set up a driving context.
  - D. Set up a logon trigger to call the package at logon time and set the context.
- a. A, B, C D
  - b. C, A, B, D
  - c. B, A, C, D
  - d. D, A, B, C



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

**Answer: b**

## Summary

In this lesson, you should have learnt about:

- Invoker's rights and their implementation.
- Implementation of White lists.
- Defining Security policies.
- Implementation of Virtual Private Databases.



ORACLE®

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

## Practice 12: Overview

This practice covers the following topics:

- Creating an application context
- Creating a policy
- Creating a logon trigger
- Implementing a virtual private database
- Testing the virtual private database



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

In this practice, you implement and test fine-grained access control.

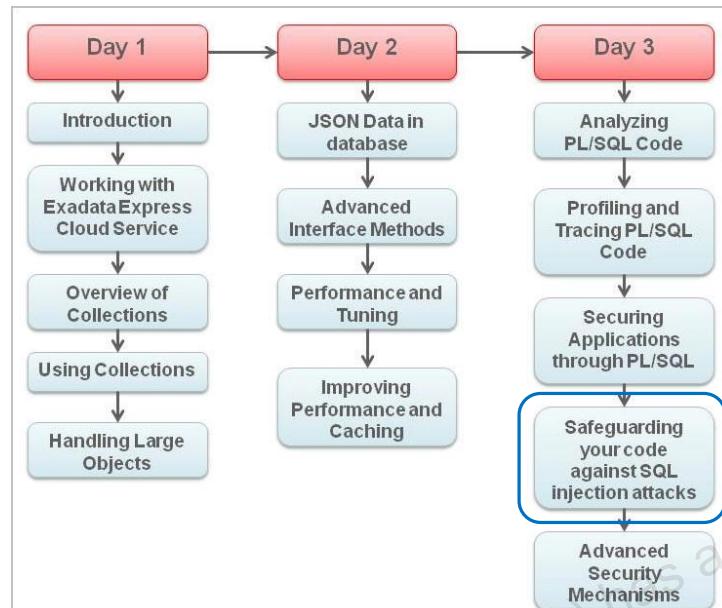
13

# Safeguarding Your Code Against SQL Injection Attacks

ORACLE

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

## Course Agenda



ORACLE

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



## Objectives

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

- Describe SQL injection
- Reduce attack surfaces
- Use DBMS\_ASSERT



ORACLE

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

In this lesson, you learn how to use the techniques and tools to strengthen your code and applications against SQL injection attacks.

## Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Static SQL vs Dynamic SQL
- Using bind arguments
- Filtering input with DBMS\_ASSERT



ORACLE®

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

## SQL injection

- SQL injection refers to an attack on the database, where an unauthorized user can gain access to sensitive data by mutating SQL statements into undesirable form.
- Can happen when PL/SQL units accept user input and construct SQL statements based on the user input.
- The attacker's statement is injected into programmer's intended statement, resulting in a valid SQL statement, hence the name **SQL injection**.
- The legal SQL statement generated through SQL injection will execute without detection and produce unintended result.



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

## SQL Injection: Example

```
CREATE OR REPLACE PROCEDURE GET_EMAIL
  (p_last_name VARCHAR2 DEFAULT NULL)
AS
  TYPE cv_custtyp IS REF CURSOR;
  cv_cv_custtyp;
  v_email customers.cust_email%TYPE;
  v_stmt VARCHAR2(400);
BEGIN
  v_stmt := 'SELECT cust_email FROM customers
            WHERE cust_last_name = '''|| p_last_name || '''';
  DBMS_OUTPUT.PUT_LINE('SQL statement: ' || v_stmt);
  OPEN cv FOR v_stmt;
  LOOP
    FETCH cv INTO v_email;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Email: '||v_email);
  END LOOP;
  CLOSE cv;
EXCEPTION WHEN OTHERS THEN
  dbms_output.PUT_LINE(sqlerrm);
  dbms_output.PUT_LINE('SQL statement: ' || v_stmt);
END;
```

Possibility of SQL injection

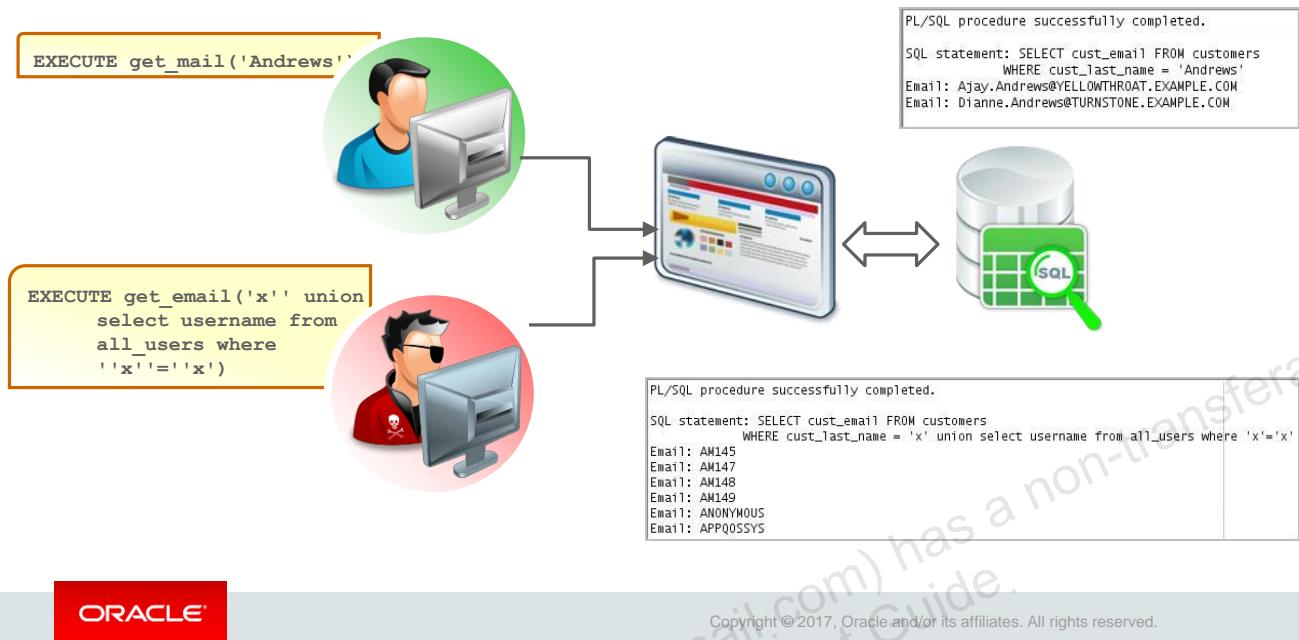


Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

You generally write PL/SQL program units which will accept user input to construct SQL statements. In the procedure `get_email`, you have a SQL statement which accepts the last name of the user and returns the email ids of the users with the given last name.

When you execute the procedure you pass a string value as parameter and execute the procedure based on that.

## Scenario



The example shown in the slide demonstrates a procedure with dynamic SQL constructed via concatenation of input value. This is vulnerable to SQL injection.

```
EXECUTE get_email('Andrews')
SQL statement: SELECT cust_email FROM customers WHERE
cust_last_name = 'Andrews'
Email: Ajay.Andrews@YELLOWTHROAT.COM
Email: Dianne.Andrews@TURNSTONE.COM

PL/SQL procedure successfully completed.

EXECUTE get_email('x' union select username from all_users where
'x'='x')
SQL statement: SELECT cust_email FROM customers WHERE
cust_last_name = 'x' union
select username from all_users where 'x'='x'
Email: AM145
Email: AM147
Email: AM148
Email: AM149
Email: ANONYMOUS
Email: APPQOSSYS
...
```

You can see that the second query could retrieve the email ids of all the users. The output might vary based on the data in your schema.

## Types of SQL Injection

- First Order attack
  - The injected text comes from the parameters of a PL/SQL program unit.
- Second Order attack
  - The injected text comes indirectly from a trusted source such as a table, where the attacker has contrived to insert a bad value.



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

Category	Description
First-order attack	The attacker can simply enter a malicious string and cause the modified code to be executed immediately.
Second-order attack	The attacker injects into persistent storage (such as a table row), which is deemed a trusted source. An attack is subsequently executed by another activity.

PL/SQL code which dynamically constructs SQL statements is vulnerable to SQL injection attacks. It is not a design bug, but malicious practice. You must adopt best practices while writing PL/SQL code to remove scope of PL/SQL injection attacks

## Avoidance Strategies Against SQL Injection

Strategy	Description
Reduce the attack surface	Ensure that all excess database privileges are revoked and that only those routines that are intended for end-user access are exposed. Though this does not entirely eliminate SQL injection vulnerabilities, it does mitigate the impact of the attacks.
Avoid dynamic SQL with concatenated input	Dynamic SQL built with concatenated input values presents the easiest entry point for SQL injections. Avoid constructing dynamic SQL this way.
Use bind arguments	Parameterize queries using bind arguments. Not only do bind arguments eliminate the possibility of SQL injections, they also enhance performance.
Filter and sanitize input	The Oracle-supplied DBMS_ASSERT package contains a number of functions that can be used to sanitize user input and to guard against SQL injection in applications that use dynamic SQL built with concatenated input values. If your filtering requirements cannot be satisfied by the DBMS_ASSERT package, create your own filter.



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

You can use several avoidance strategies to safeguard against, or mitigate the impact of, SQL injection attacks. The slide lists high-level descriptions of each of the strategies that are examined in more detail on subsequent pages.

The available and best methods for eliminating SQL injection vulnerability may depend on the vulnerability itself. Not all methods are available for addressing every vulnerability.

### Methods

- Use static SQL, if all Oracle identifiers (for example, column, table, view, trigger, program unit, or schema names) are known at code compilation time.
 

**Note:** Static SQL automatically binds arguments. Data definition language (DDL) statements cannot be executed with static SQL.
- Use dynamic SQL with bind arguments, if any WHERE clause values, VALUES clause values, or SET clause values are unknown, and any Oracle identifiers are unknown at code compilation time.
 

**Note:** Use bind arguments for the values (the literals). Use string concatenation for the validated and sanitized Oracle identifiers.
- Validate and sanitize input, if concatenating any strings and if bind arguments require additional filtering.

## Protecting Against SQL Injection: Example

```
CREATE OR REPLACE PROCEDURE GET_EMAIL
  (p_last_name VARCHAR2 DEFAULT NULL)
AS
BEGIN
  FOR i IN
    (SELECT cust_email
     FROM customers
     WHERE cust_last_name = p_last_name)
  LOOP
    DBMS_OUTPUT.PUT_LINE('Email: '||i.cust_email);
  END LOOP;
END;
```

This example avoids dynamic SQL with concatenated input values.

```
EXECUTE get_email('Andrews');
```

PL/SQL procedure successfully completed.  
Email: Ajay.Andrews@YELLOWTHROAT.EXAMPLE.COM  
Email: Dianne.Andrews@TURNSTONE.EXAMPLE.COM

```
EXECUTE get_email('x' union select
username from all_users where
'x'='x');
```

PL/SQL procedure successfully completed.



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

The example in the slide with static SQL is protected against SQL injection. You can see in the procedure get\_email, when you use a string variable instead of a concatenated string as input parameter unintended data is not exposed.

When you execute with the parameter 'Andrews' you see the output is as intended and when you use an arbitrary string as parameter no data is returned as the query didn't find any match.

## Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Static SQL vs Dynamic SQL
- Using bind arguments
- Filtering input with `DBMS_ASSERT`



ORACLE®

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

## Reducing the Attack Surface

- To avoid widespread damage because of a SQL injection attack, you have to reduce the possible attack surface.
- Following are some of the programming practices you can implement to reduce the attack surface.
  - Expose database manipulation only through a PL/SQL API.
  - Use invoker's rights access to the PL/SQL program units.
  - Reduce arbitrary inputs.
  - Strengthen database security.



ORACLE

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

### Expose the database to clients only via a PL/SQL API

Carefully control privileges so that the client has no direct access to the application's other kinds of objects, especially tables and views.

If an interface is not available to an attacker, it is clearly not available to be abused. Thus the first, and arguably the most important, line of defense is to reduce the exposed interfaces to only those absolutely required.

### Use invoker's rights

Stored program units and SQL methods execute with a set of privileges. By default, the privileges are those of the schema owner, also known as the definer. The definer's rights not only dictate the privileges, they are also used to resolve object references.

If a program unit doesn't need to be executed with the escalated privileges of the definer, you should specify that the program unit executes with the privileges of the caller, also known as the invoker. Invoker's rights can *mitigate* the risk of SQL injection. However, using invoker's rights don't guarantee complete elimination of SQL injection risk.

### Reduce Arbitrary input

Because a SQL injection attack is really possible only if user input is allowed, you can prevent the attack by limiting user input.

First, you must reduce the end-user interfaces to only those actually needed. For example:

- In a Web application, restrict the users access to specified Web pages.
- In a PL/SQL API, expose only the routines intended for customer use.

Second, where user input is required, make use of language features to ensure that only data of the intended type is specified. For example:

- Don't specify a `VARCHAR2` parameter when the parameter will be used as a number.
- Don't use `NUMBER` if you need only positive integers, use `NATURAL` instead.

## Expose the Database Only Via PL/SQL API

- Expose the database to clients only via a PL/SQL API.
- When you design a PL/SQL package that accesses the database, use the following paradigm:
  - Establish a database user as the *only* one to which a client may connect. Hypothetically, let us call this user `myuser`.
  - `myuser` may own only synonyms and these synonyms may denote *only* PL/SQL units owned by other users.
  - Grant the `Execute` privilege on only the denoted PL/SQL units to `myuser`.



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

Expose the database to clients only via a PL/SQL API. Carefully control privileges so that the client has no direct access to the application's other kinds of objects, especially tables and views.

## Using Invoker's Rights

- Using invoker's rights help to:
  - Limit the privileges
  - Minimize the security exposure
- The following example doesn't use invoker's rights:

```
CREATE OR REPLACE
PROCEDURE drop_user(p_username VARCHAR2 DEFAULT NULL)
IS
  v_sql_stmt VARCHAR2(500);
BEGIN
  v_sql_stmt := 'DROP USER'||p_username;
  EXECUTE IMMEDIATE v_sql_stmt;
END drop_user;
```

1

Note the use of dynamic SQL with concatenated input values.

```
GRANT EXECUTE ON drop_user to OE, HR, SH;
```

2



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

Stored program units and SQL methods execute with a set of privileges. By default, the privileges are those of a schema owner, also known as the definer. The definer's rights not only dictate the privileges, they are also used to resolve object references. If a program unit does not need to be executed with the escalated privileges of the definer, you should specify that the program unit executes with the privileges of a caller, also known as the invoker.

1. The example shown in the slide uses definer's rights. The `DROP_USER` procedure is created under the `SYS` schema. It accepts two parameters and uses them in the `ALTER USER` statement.
2. `SYS` grants `OE`, `HR`, and `SH` the ability to execute the `DROP_USER` procedure. This implies that the users `OE`, `HR` and `SH` now can drop any user created by `SYS` user because the `DROP_USER` procedure is created with definer's rights.

## Using Invoker's Rights

- OE is successful at changing the SYS password, because, by default, CHANGE\_PASSWORD executes with SYS privileges:

```
EXECUTE sys.drop_user('AM147');
```

PL/SQL procedure successfully completed.

- Add the AUTHID to change the privileges to the invokers:

```
CREATE OR REPLACE
PROCEDURE drop_user(p_username VARCHAR2 DEFAULT NULL)
AUTHID CURRENT_USER IS
    v_sql_stmt VARCHAR2(500);
BEGIN
    v_sql_stmt := 'DROP USER '||p_username;
    EXECUTE IMMEDIATE v_sql_stmt;
END drop_user;
/
EXECUTE sys.drop_user('AM148');
```

Error starting at line : 142 in command -  
 EXECUTE sys.drop\_user ('AM148')  
 Error report -  
 ORA-01031: insufficient privileges  
 ORA-06512: at "SYS.DROP\_USER", line 8  
 ORA-06512: at line 1  
 01031. 00000 - "insufficient privileges"  
 \*Cause: An attempt was made to perform a database operation without  
 the necessary privileges.  
 \*Action: Ask your database administrator or designated security  
 administrator to grant you the necessary privileges

ORACLE

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

When OE executes the DROP\_USER procedure, it is executed under SYS privileges (the definer of the procedure), and with the code shown in the slide, OE can drop any user. Obviously, this is an unacceptable outcome.

To prevent another schema from dropping any user that does not belong to the schema, redefine the procedure with the invoker's rights. This is done with the AUTHID CURRENT\_USER option.

Now OE can no longer drop any user created by SYS.

Although using invoker's rights does not guarantee the elimination of SQL injection risks, it can help mitigate the exposure.

## Strengthen Database Security

- Encrypt sensitive data so that it cannot be viewed.
- Avoid:
  - Using PUBLIC privileges
  - Using EXECUTE ANY PROCEDURE privilege
  - Granting privileges the WITH ADMIN option
- Don't allow wide access to any standard Oracle packages that can operate on the operating system.
- Enforce password management.



ORACLE

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

Oracle Database contains inherent security features that help in protecting it from many types of attacks, including SQL injection. The following is a list of some of the practices to observe when you secure the Oracle database:

- Encrypt sensitive data so that it cannot be viewed.
- Evaluate all PUBLIC privileges and revoke them where possible.
- Do not widely grant EXECUTE ANY PROCEDURE.
- Avoid granting privileges the WITH ADMIN option.
- Do not allow wide access to any standard Oracle packages that can operate on the operating system. These packages include UTL\_HTTP, UTL\_SMTP, UTL\_TCP, DBMS\_PIPE, UTL\_MAIL, and UTL\_FTP.
- Certain Oracle packages, such as UTL\_FILE and DBMS\_LOB, are governed by the privilege model of the Oracle DIRECTORY object. Protect Oracle DIRECTORY objects.
- Lock the database default accounts and configure to expire the default passwords.
- Remove example scripts and programs from the Oracle directory.
- Run the database listener as a non privileged user.
- Apply basic password management rules, such as password length, history, and complexity, to all user passwords. Mandate that all users change their passwords regularly.
- Lock and expire the default user accounts and change the default user password.

## Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Static SQL Vs Dynamic SQL
- Using bind arguments
- Filtering input with DBMS\_ASSERT



ORACLE®

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

## Using Static SQL

- Eliminates SQL injection vulnerability
- Creates schema object dependencies upon successful compilation
- Can improve performance, when compared with DBMS\_SQL



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

Because SQL injection is a feature of SQL statements that are dynamically constructed via user inputs, it follows that designing your application to be based on static SQL reduces the scope for attack.

There are two common situations where developers often use dynamic SQL, when static SQL would serve the purpose and be more secure:

- Handling a varying number of IN-list values in the query condition
- Handling the LIKE comparison operator in the query condition

## Using Static SQL

```

CREATE OR REPLACE PROCEDURE list_products_dynamic
(p_product_name VARCHAR2 DEFAULT NULL)
AS
    TYPE cv_prodtyp IS REF CURSOR;
    cv_cv_prodtyp;
    v_prodname product_information.product_name%TYPE;
    v_minprice product_information.min_price%TYPE;
    v_listprice product_information.list_price%TYPE;
    v_stmt VARCHAR2(400);
BEGIN
    v_stmt := 'SELECT product_name, min_price, list_price FROM product_information
              WHERE product_name LIKE ''%'||p_product_name||'%''';
OPEN cv FOR v_stmt;
dbms_output.put_line(v_stmt);
LOOP
    FETCH cv INTO v_prodname, v_minprice, v_listprice;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Product Info: '||v_prodname||', '|| 
                           v_minprice||', '|| v_listprice);
END LOOP;
CLOSE cv;
END;

```

You can convert this statement to static SQL.



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

By converting the dynamic static SQL shown in the slide, you can decrease your attack vulnerability. The code in the slide uses dynamic SQL to handle the `LIKE` operator in the query condition. Notice that string concatenation is used to build the SQL statement.

Examine the execution of this injection:

```

EXECUTE list_products_dynamic('' and 1=0 union select
cast(username as varchar2(100)), null, null
from all_users --')

```

PL/SQL procedure successfully completed.

```

SELECT product_name, min_price, list_price
      FROM product_information WHERE product_name LIKE
      '%' and 1=0 union select cast(username as varchar2(100)), null, null from all_users --%
Product Info: AM145, ,
Product Info: AM147, ,
Product Info: AM148, ,
Product Info: AM149, ,
Product Info: ANONYMOUS, ,
Product Info: APPQOSSYS, ,

```

Notice that the injection succeeded through the concatenation of the `UNION` set operator to the dynamic SQL statement.

## Using Static SQL

- To use static SQL, accept the user input, and then concatenate the necessary string to a local variable.
- Pass the local variable to the static SQL statement.

```

CREATE OR REPLACE PROCEDURE list_products_static
  (p_product_name VARCHAR2 DEFAULT NULL)
AS
  v_bind  VARCHAR2(400);
BEGIN
  v_bind := '%'||p_product_name||'%';
  FOR i IN
    (SELECT product_name, min_price, list_price
     FROM product_information
     WHERE product_name like v_bind)
  LOOP
    DBMS_OUTPUT.PUT_LINE('Product Info: '||i.product_name ||
      '|| i.min_price ||', '|| i.list_price);
  END LOOP;
END list_products_static;

```



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

In the example shown in the slide, you can avoid SQL injection by concatenating the user input and placing it into a local variable, and then using the local variable within the static SQL.

Examine the results:

```
EXECUTE list_products_static('Laptop');
```

```

PL/SQL procedure successfully completed.

Product Info: Laptop 128/12/56/v90/110,
              2606, 3219
Product Info: Laptop 16/8/110,
              800, 999
Product Info: Laptop 32/10/56,
              1542, 1749
Product Info: Laptop 48/10/56/110,
              2073, 2556
Product Info: Laptop 64/10/56/220,
              2275, 2768

```

```
-- this example attempts injection
EXECUTE list_products_static(''' and 1=0 union select
cast(username as varchar2(100)), null, null from all_users --')
```

```

PL/SQL procedure successfully completed.


```

## Using Dynamic SQL

- Dynamic SQL may be unavoidable in the following types of situations:
  - You do not know the full text of the SQL statements that must be executed in a PL/SQL procedure.
  - You want to execute DDL statements and other SQL statements that are not supported in purely static SQL programs.
  - You want to write a program that can handle changes in data definitions without the need to recompile.
- If you must use dynamic SQL, try not to construct it through concatenation of input values. Instead, use bind arguments.



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

- Dynamic SQL may be unavoidable in the following types of situations:
  - You do not know the full text of the SQL statements that must be executed in a PL/SQL procedure—for example, a `SELECT` statement that includes an identifier (such as table name) that is unknown at compile time or a `WHERE` clause in which the number of subclauses is unknown at compile time.
  - You want to execute DDL statements and other SQL statements that are not supported in purely static SQL programs.
  - You want to write a program that can handle changes in data definitions without the need to recompile.
- If you must use dynamic SQL, try not to construct it through concatenation of input values. Instead, use bind arguments.

If you cannot avoid input concatenation, you must validate the input values, and also consider constraining user input to a predefined list of values, preferably numeric values. Input filtering and sanitizing are covered in more detail later in this lesson.

## Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Static SQL vs Dynamic SQL
- Using bind arguments
- Filtering input with DBMS\_ASSERT



ORACLE®

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

## Using Bind Arguments with Dynamic SQL

You can rewrite the following statement

```
v_stmt :=  
  'SELECT |||filter(p_column_list)||| FROM customers |||  
  'WHERE account_mgr_id = ''||| p_sales_rep_id |||';  
  
EXECUTE IMMEDIATE v_stmt;
```

as the following dynamic SQL with a placeholder (:1) by using a bind argument (p\_sales\_rep\_id):

```
v_stmt :=  
  'SELECT |||filter(p_column_list)||| FROM customers |||  
  'WHERE account_mgr_id = :1';  
  
EXECUTE IMMEDIATE v_stmt USING p_sales_rep_id;
```



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

If you must use dynamic SQL, using bind arguments will be the next best protection against SQL injection attacks. Using bind arguments also enables cursor sharing, and thus improves application performance.

You can use bind arguments in the WHERE clause, the VALUES clause, or the SET clause of any SQL statement, as long as the bind arguments are not used as Oracle identifiers (such as column names or table names) or keywords.

Developers often use dynamic SQL to handle a varying number of IN-list values or LIKE comparison operators in the query condition. While using dynamic SQL use bind variables instead of concatenated strings to avoid SQL injection attacks.

## Using Bind Arguments with Dynamic PL/SQL

If you must use dynamic PL/SQL, try to use bind arguments. For example, you can rewrite the following dynamic PL/SQL with concatenated string values

```
v_stmt :=  
'BEGIN  
    get_phone ('' || p_fname || '', '' || p_lname || ''); END;'  
  
EXECUTE IMMEDIATE v_stmt;
```

as the following dynamic PL/SQL with placeholders (:1, :2) by using bind arguments (p\_fname, p\_lname):

```
v_stmt :=  
'BEGIN  
    get_phone(:1, :2); END;'  
  
EXECUTE IMMEDIATE v_stmt USING p_fname, p_lname;
```



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

As with dynamic SQL, you should avoid constructing dynamic PL/SQL with string concatenation. The impact of SQL injection vulnerabilities in dynamic PL/SQL is more serious than in dynamic SQL, because, with dynamic PL/SQL, multiple statements (such as `DELETE` or `DROP`) can be batched together and injected.

## What if You Cannot Use Bind Arguments?

- Bind arguments cannot be used with:
  - DDL statements
  - Oracle identifiers
- If bind arguments cannot be used with the dynamic SQL or PL/SQL, you must filter and sanitize all input concatenated to the dynamic statement.



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

Although you should strive to use bind arguments with all dynamic SQL and PL/SQL statements, there are instances where bind arguments cannot be used:

- DDL statements (such as CREATE, DROP, and ALTER)
- Oracle identifiers (such as names of columns, tables, schemas, database links, packages, procedures, and functions)

If bind arguments cannot be used with the dynamic SQL or PL/SQL, you must filter and sanitize all input concatenated to the dynamic statement. In the following slides, you learn how to use the Oracle-supplied DBMS\_ASSERT package functions to filter and sanitize input values.

## Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Static SQL vs Dynamic SQL
- Using bind arguments
- Filtering input with DBMS\_ASSERT



ORACLE®

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

## DBMS\_ASSERT Package

- Provides an interface to validate properties of the input value.
- You can use **DBMS\_ASSERT** to sanitize and filter input to dynamic SQL statements.
- You can write assertions to check the input.
- If the assertion fails then the execution would stop returning an error.



ORACLE

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

## Understanding DBMS\_ASSERT

DBMS\_ASSERT functions:

Function	Description
<b>ENQUOTE_LITERAL</b>	Encloses string literal in single quotes
<b>ENQUOTE_NAME Function</b>	Ensures that a string is enclosed by quotation marks, then checks that the result is a valid SQL identifier.
<b>NOOP Functions</b>	Returns the value without any checking
<b>QUALIFIED_SQL_NAME Function</b>	Verifies that the input string is a qualified SQL name
<b>SCHEMA_NAME Function</b>	Verifies that the input string is an existing schema name
<b>SIMPLE_SQL_NAME</b>	Verifies that the string is a simple SQL name
<b>SQL_OBJECT_NAME Function</b>	Verifies that the input parameter string is a qualified SQL identifier of an existing SQL object



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

To guard against SQL injection in applications that do not use bind arguments with dynamic SQL, you must filter and sanitize concatenated strings. The primary use case for dynamic SQL with string concatenation is when an Oracle identifier (such as a table name) is unknown at code compilation time.

DBMS\_ASSERT is an Oracle-supplied PL/SQL package containing seven functions that can be used to filter and sanitize input strings, particularly those that are meant to be used as Oracle identifiers. When using the DBMS\_ASSERT package, always specify the SYS schema rather than relying on a public synonym.

## Oracle Identifiers

- To use DBMS\_ASSERT effectively, you must understand how Oracle identifiers can be specified and used.
- In a SQL statement, you specify the name of an object with an unquoted or quoted identifier.
  - The object name used as an identifier:

```
SELECT count(*) records FROM orders;
```

- The object name used as a literal:

```
SELECT num_rows FROM user_tables WHERE table_name = 'ORDERS';
```



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

To use DBMS\_ASSERT effectively, you must understand how Oracle identifiers can be specified and used.

In a SQL statement, you specify the name of an object with an unquoted or quoted identifier.

- An unquoted (or internal format) identifier is not surrounded by punctuation. It must begin with a letter, and may be followed by letters, numbers, or a small set of special characters. This is how identifiers are most often specified, and how object names are stored in data dictionary tables.
- A quoted (or normal format) identifier begins and ends with double quotation marks. The identifier can include almost any character within the double quotes. This format is user supplied.

## Oracle Identifiers

- The object name used as a quoted (normal format) identifier:
  - The "orders" table referenced is a different table compared to the orders table
  - Such identifiers are vulnerable to SQL injection.

```
SELECT count(*) records FROM "orders";
```



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

The statement in the slide uses a quoted (normal format) identifier and references a different table. SQL injection attacks can use the quoted method to attempt to subvert code that is written to expect only the unquoted method.

## Working with Identifiers in Dynamic SQL

- For your identifiers, determine:
  - Where will the input come from: user or data dictionary?
  - What verification is required?
  - How will the result be used, as an identifier or a literal value?
- Based on these three factors, you have to decide on:
  - What preprocessing is required (if any) prior to calling the verification functions
  - Which DBMS\_ASSERT verification function is required
  - What post-processing is required before the identifier can actually be used



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

When working with identifiers in dynamic SQL statements, you must first determine where the data is coming from, either user data or data dictionary data. For verification, you must determine whether the object must exist and the type of identifier. For types of identifiers, you have:

- **SQL Literal:** A SQL literal is a constant value, written at compile time and is read-only at run time. There are three kinds of SQL literal: text, datetime, and numeric.
- **Simple:** A simple SQL name is a string that conforms to the following basic characteristics:
  - The first character of the name is alphabetic.
  - The name contains only alphanumeric characters or the \_, \$, and # characters.
  - Quoted names must be enclosed within double quotation marks and may contain any characters, including quotation marks, provided they are represented by two quotation marks in a row.
- **Qualified SQL name:** A qualified SQL name is one or more simple SQL names that may be followed by a database link.

Finally, determine how the result will be used, as an identifier or a literal value.

Answering these questions will impact your preprocessing, post-processing, and the use of DBMS\_ASSERT.

## Choosing a Verification Route

Based on the type of identifier, you can choose an appropriate verification routine of `DBMS_ASSERT`:

- SQL literal - Verify whether the literal is a well-formed SQL literal by using `DBMS_ASSERT.ENQUOTE_LITERAL`.
- Simple SQL name - Verify that the input string conforms to the basic characteristics of a simple SQL name by using `DBMS_ASSERT.SIMPLE_SQL_NAME`.
- Qualified SQL name:
  - Step 1 - Decompose the qualified SQL name into its simple SQL names by using `DBMS.Utility.Name_Tokenize()`.
  - Step 2 - Verify each of the simple SQL names by using `DBMS_ASSERT.SIMPLE_SQL_NAME`.



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

After determining the type of identifier that you must verify, follow the guidelines in the slide to select the appropriate verification routine.

For example, you must write a procedure that allows a user to change his or her password. To choose the correct `DBMS_ASSERT` function to use, you must determine:

- Where the input will come from, the user or the data dictionary
- What verification is required (Does the object need to exist and if so, what type of identifier is it?)
- How the identifier will be used, as an identifier or as a literal

## Validate Input Using DBMS\_ASSERT

- Use DBMS\_ASSERT package provides an interface to validate properties of the input value.
- You can validate input using the subprograms defined in DBMS\_ASSERT package.
- If the input doesn't uphold the assertion then the execution exits returning an error.
- Malicious input thus fails to execute.
- DBMS\_ASSERT is a SYS schema package.



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

Subprogram	Description
ENQUOTE_LITERAL Function	Enquotes a string literal
ENQUOTE_NAME Function	Ensures that a string is enclosed by quotation marks, then checks that the result is a valid SQL identifier.
NOOP Functions	Returns the value without any checking
QUALIFIED_SQL_NAME Function	Verifies that the input string is a qualified SQL name
SCHEMA_NAME Function	Verifies that the input string is an existing schema name
SIMPLE_SQL_NAME Function	Verifies that the input string is a simple SQL name
SQL_OBJECT_NAME Function	Verifies that the input parameter string is a qualified SQL identifier of an existing SQL object

## Avoiding Injection by Using DBMS\_ASSERT.SIMPLE\_SQL\_NAME

```

CREATE OR REPLACE PROCEDURE show_col2 (p_colname varchar2, p tablename  varchar2)
AS
type t is varray(200) of varchar2(25);
Results t;
Stmt CONSTANT VARCHAR2(4000) :=
    'SELECT'||dbms_assert.simple_sql_name( p_colname )||' FROM'|||
dbms_assert.simple_sql_name( p tablename ) ;

BEGIN
    DBMS_Output.Put_Line ('SQL Stmt: '|| Stmt);
    EXECUTE IMMEDIATE Stmt bulk collect into Results;
for j in 1..Results.Count() loop
DBMS_Output.Put_Line(Results(j));
end loop;
END show_col2;

```

Verify that the input string conforms to the basic characteristics of a simple SQL name.



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

### The Complete Code

```

CONN hr/hr
SET SERVEROUTPUT ON
CREATE OR REPLACE
PROCEDURE show_col (p_colname varchar2, p tablename  varchar2)
AS
type t is varray(200) of varchar2(25);
Results t;
Stmt CONSTANT VARCHAR2(4000) :=
    'SELECT'|| p_colname ||' FROM'|| p tablename ;
BEGIN
    DBMS_Output.Put_Line ('SQL Stmt: '|| Stmt);
    EXECUTE IMMEDIATE Stmt bulk collect into Results;
for j in 1..Results.Count() loop
DBMS_Output.Put_Line(Results(j));
end loop;
--EXCEPTION WHEN OTHERS THEN
--Raise_Application_Error(-20000, 'Wrong table name');
END show_col;
/

```

```

execute show_col('Email','EMPLOYEES');
-- SQL injection, the query will retrieve all the USERs of
the schema instead of email_ids of the employees.
execute show_col('Email','EMPLOYEES where 1=2 union select
Username c1 from All_Users --');

CREATE OR REPLACE
PROCEDURE show_col2 (p_colname varchar2, p_tablename
varchar2)
AS
type t is varray(200) of varchar2(25);
Results t;
Stmt CONSTANT VARCHAR2(4000) :=
  'SELECT '||dbms_assert.simple_sql_name( p_colname ) || '
FROM '|| dbms_assert.simple_sql_name( p_tablename ) ;

BEGIN
  DBMS_Output.Put_Line ('SQL Stmt: ' || Stmt);
  EXECUTE IMMEDIATE Stmt bulk collect into Results;
for j in 1..Results.Count() loop
  DBMS_Output.Put_Line(Results(j));
end loop;
--EXCEPTION WHEN OTHERS THEN
--Raise_Application_Error(-20000, 'Wrong table name');
END show_col2;
/

```

```

execute show_col2('Email','EMPLOYEES');
execute show_col2('Email','EMPLOYEES where 1=2 union select
Username c1 from All_Users --');

```

## DBMS\_ASSERT Guidelines

- Do not perform unnecessary uppercase conversions on identifiers.

```
--Bad:  
SAFE_SCHEMA := sys.dbms_assert.SIMPLE_SQL_NAME(UPPER(MY_SCHEMA)) ;  
--Good:  
SAFE_SCHEMA := sys.dbms_assert.SIMPLE_SQL_NAME(MY_SCHEMA) ;  
--Best:  
SAFE_SCHEMA := sys.dbms_assert.ENQUOTE_NAME(  
SAFE_SCHEMA := sys.dbms_assert.ENQUOTE_LITERAL(  
    sys.dbms_assert.SIMPLE_SQL_NAME(MY_SCHEMA)) ;
```

- When using ENQUOTE\_LITERAL, do not add unnecessary double quotation marks around identifiers.

```
--Bad:  
my_trace_routine(''||sys.dbms_assert.ENQUOTE_LITERAL(  
my_procedure_name)||');'||...  
--Good:  
my_trace_routine(''||sys.dbms_assert.ENQUOTE_LITERAL(  
replace(my_procedure_name, '''', '''))||');'||...
```



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

Choosing the correct DBMS\_ASSERT verification routines is important, and using them correctly is just as important.

### Limitations

DBMS\_ASSERT is not a panacea for all sorts of PL/SQL evils. It is essentially a collection of pattern-matching routines that confirm whether the supplied string matches expected patterns. It can be used to protect against certain kinds of malicious input, but it cannot comprehensively defend against all such inputs.

Here are some instances where DBMS\_ASSERT may not help:

- It contains no routines to validate TNS connect strings, for example, “((description =...”.
- It is not designed nor is it intended to be a defense against cross-site scripting attacks.
- It does not check for input string lengths, and therefore, cannot be used as any kind of defense against a buffer overflow attack.
- It does not guarantee that a SQL name is, in fact, a parseable SQL name.
- It does not protect against parsing as the wrong user or other security risks due to inappropriate privilege management.

## DBMS\_ASSERT Guidelines

- Check and reject NULL or empty return results from DBMS\_ASSERT (test for NULL, '' , and '""' ).
- Prefix all calls to DBMS\_ASSERT with the owning schema, SYS.
- Protect all injectable parameters and code paths.



ORACLE®

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

### Protect all injectable parameters and code paths.

- **Bad**

```
FUNCTION name_elided
  (LAYER VARCHAR2, OWNER VARCHAR2, FIELD VARCHAR2)
  RETURN BOOLEAN IS
  CRS INTEGER;
  BEGIN
  CRS := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQLPARSE(CRS, 'select |||FIELD ||| from |||OWNER|||.'|||
    sys.dbms_assert.QUALIFIED_SQL_NAME(LAYER) ||| '_elided',
  DBMS_SQL.NATIVE);
```

You are not writing assertions to the FIELD and OWNER values which leaves the code vulnerable to SQL injection attacks.

- **Good**

```
FUNCTION name_elided
(LAYER VARCHAR2, OWNER VARCHAR2, FIELD VARCHAR2)
RETURN BOOLEAN IS
CRS INTEGER;
BEGIN
CRS := DBMS_SQL.OPEN_CURSOR;
DBMS_SQLPARSE(CRS, 'select
'||sys.dbms_assert.SIMPLE_SQL_NAME(FIELD)||' from
'||sys.dbms_assert.SIMPLE_SQL_NAME(OWNER)||'.'
||sys.dbms_assert.SIMPLE_SQL_NAME(LAYER)||'_elided',
DBMS_SQL.NATIVE);
```

## DBMS\_ASSERT Guidelines

- If DBMS\_ASSERT exceptions are raised from a number of input strings, define and raise exceptions explicitly to ease debugging during application development.

```
-- Bad
CREATE OR REPLACE PROCEDURE change_password3
  (username VARCHAR2, password VARCHAR2)
AS
BEGIN
  ...
EXCEPTION WHEN OTHERS THEN
  RAISE;
END;
```



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

You can use PL/SQL conditional compilation to manage self-tracing code. For production deployment, the debug messages can be turned off by setting the PLSQL\_CCFLAGS parameter for tracing to FALSE. You must ensure that error messages displayed in production deployment do not reveal information that is useful to hackers.

```
ALTER SESSION SET Plsql_CCFlags = 'Tracing:true';

CREATE OR REPLACE PROCEDURE change_password3
  (p_username VARCHAR2, p_password VARCHAR2) AS
BEGIN
  ...
EXCEPTION
  WHEN sys.dbms_assert.INVALID_SCHEMA_NAME THEN
    $if $$Tracing $then dbms_output.put_line('Invalid user.');
    $else dbms_output.put_line('Authentication failed.'); $end
  WHEN sys.dbms_assert.INVALID_SQL_NAME THEN
    $if $$Tracing $then dbms_output.put_line('Invalid pw.');
    $else dbms_output.put_line('Authentication failed.'); $end
  WHEN OTHERS THEN
    dbms_output.put_line('Something else went wrong');
END;
```

## Quiz



Code that is most vulnerable to SQL Injection attack contains:

- a. Input parameters
- b. Dynamic SQL with bind arguments
- c. Dynamic SQL with concatenated input values
- d. Calls to external functions

ORACLE

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

**Answer: c**

## Quiz



By default, a stored procedure executes with the privileges of its owner (definer's rights).

- a. True
- b. False

ORACLE

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

**Answer: a**

## Quiz



If you must use dynamic SQL, avoid using input concatenation to build the dynamic SQL.

- a. True
- b. False

**ORACLE**

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

**Answer: a**

## Quiz



In the statement `SELECT total FROM orders WHERE ord_id=p_ord_id`, the table name `orders` is being used as which of the following ?

- a. A literal
- b. An identifier
- c. A placeholder
- d. An argument

ORACLE

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

**Answer: b**

## Summary

In this lesson, you should have learned how to:

- Detect SQL injection vulnerabilities
- Reduce attack surfaces
- Use DBMS\_ASSERT



ORACLE

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

This lesson showed you techniques and tools to strengthen your code and applications against SQL injection attacks.

For more information, see “Appendix D, Designing and Testing Your Code to Avoid SQL Injection Attacks.”

## Practice 13: Overview

This practice covers the following topics:

- Testing your knowledge of SQL injection
- Rewriting code to protect against SQL injection



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

Using the OE, you examine PL/SQL code, test it for SQL injection, and rewrite it to protect against SQL injection vulnerabilities.

Use the OE schema for this practice.

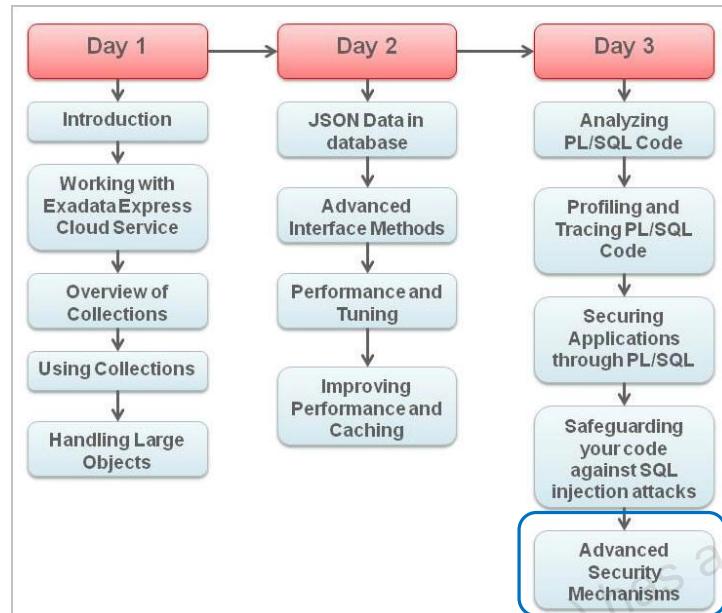
**Note:** For more information about examining, testing, and rewriting a PL/SQL code, see “Appendix D: Designing and Testing Your Code to Avoid SQL Injection Attacks.”

# Advanced Security Mechanisms

ORACLE

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

## Course Agenda



ORACLE

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

In this lesson, you are introduced to Collections. Collections are composite data types, you can store values of data which can in turn have various internal components. In earlier course (Oracle Database 12c: PL/SQL Program Units) you must've learnt about records. Records are also composite data types.

A collection stores a set of values of same data type, whereas a record stores a set of values of different data types. In this lesson we will discuss different variants of collections.

## Objectives

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

- Understand Real Application Security.
- Understand Transparent Data Encryption mechanism.
- Understand Oracle Data Redaction mechanism.



ORACLE

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

In this lesson, you learn about the security features in the Oracle Database from an application developer's standpoint.

For more information about these features, refer to *Oracle Supplied PL/SQL Packages and Types Reference*, *Oracle Label Security Administrator's Guide*, *Oracle Single Sign-On Application Developer's Guide*, and *Oracle Security Overview*.

## Lesson Agenda

- Real Application Security
- Transparent Data Encryption
- Data Redaction



ORACLE®

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

## Real Application Security

- Real Application Security is implemented to ensure security the modern day three-tier applications.
- RAS provides an application access control framework through users, privileges and policies.
- RAS is a policy based authorization model that recognizes application level users, privileges and roles within the database.

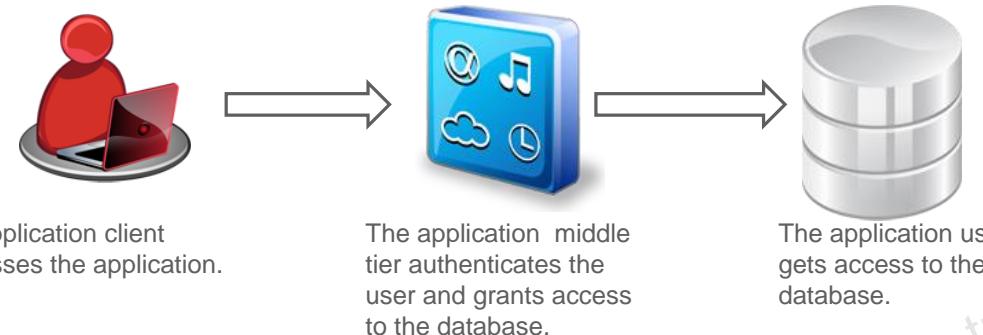


Copyright © 2017, Oracle and/or its affiliates. All rights reserved.

A typical 3-tier application running in the application server, connects to the database as a privileged schema user, and then enforces appropriate access rights based upon the application users and privileges. Application-level access control requirements are not on database tables or views. The data for a target business object can span across many tables or views of different database schemas. Oracle Database 12c Real Application Security (RAS) provides a rich, declarative access control model that natively supports application-specific authorization primitives — users, roles, privileges — within the database.

Real Application Security is an application security framework which enables developers to declaratively define, provision, and enforce their security requirements. Oracle RAS introduces a policy-based authorization model that recognizes application-level users, privileges, and roles within the database, and then controls access on both static and dynamic collections of records representing business objects. The application can also use RAS to find the relevant access control policies on application operations, and enforce them in the middle-tier.

## How It Works Without RAS?



ORACLE

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

An application user is authenticated as a privileged schema user, and then enforces appropriate access rights based upon the application users and privileges. As database connections are expensive to maintain at a per-user level in multi-tier applications, applications share their database connections for all application logic.

To mitigate access control and accountability concerns, some applications store the end-user security context in database session using application variables, but this requires frequent switching of the security context, affecting performance and scalability.

Implementing Virtual Private Databases can protect applications by applying row-level access control rules on the queries. Depending on the application this technique might become very expensive. Apart from this the application has to build infrastructure to provide information on application users/roles, session state and so on.

Real Application Security provides a security framework, where application users privileges are applied in the database as well.

## How It Works with RAS?



ORACLE

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

Real Application Security enables these security tasks, which improve database security and performance:

- Three-tier and two-tier applications can declaratively define, provide, and enforce access control requirements at the database layer.
- The database can provide a uniform security model across all tiers and support multiple application user stores, including the associated roles, authentication credentials, database attributes, and application-defined attributes. This model enables application users to have a single unique global identity across an Oracle enterprise.
- An Oracle database can natively support the application security context. The database supports integrated policy specification and enforcement for both the application and the database, so the application does not need to do this through application code. Because the database stores the application security context information, this also reduces network traffic.
- Developers can use Real Application Security to control application user access to data in an Oracle database throughout all components of an Oracle enterprise in a common manner.

## Real Application Security - Components

Oracle RAS model introduces the following components within the Oracle database:

- Application Users
- Application privileges
- Application Roles
- Data Realms
- Session Namespace Attributes
- Application Sessions
- Access Control Lists
- Data Security Policy
- Authorization Service



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

- **Application Users:** Application end-user identities that are unified across the application and the database tiers. These users are unaware of the database schema and don't own any database objects or resources.
- **Application Privileges:** Named privileges to control execution of application-level operations. The operations can be on rows or columns of a database table or application artifacts such as UI artifacts representing workflow tasks or buttons and pages in a Web application.
- **Application Roles:** Groups of application privileges or other application or database roles. During user provisioning, these roles are assigned to application users.
- **Data Realms:** A collection of a logical set of rows in a table or in a group of related application tables. A Data Realm is the primary construct to specify data security policy. It represents an application-level resource or business object and is defined using a SQL predicate.
- **Session Namespace Attributes:** Collections of attribute-value pairs that can be used in the SQL predicate to define a data realm. Each collection is managed under an application namespace with associated access control policy.
- **Application Sessions:** Application users' sessions that correspond to application users' security contexts - roles and namespace attributes - in the database. These end-user sessions are created through the application tier and are natively supported in the database.
- **Access Control Lists (ACL):** A named list of privilege grants to users or roles. Oracle RAS ACL allows various constraints on the privilege grants such as ordered negative grants.
- **Data Security Policy:** Protects Data Realms by associating them with ACLs.
- **Authorization Service:** Checks if a privilege is granted to the user in the current session.

## Implementing a RAS Data Security Policy



ORACLE

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

Application users within the database represent various application users according to the domain. In the database you map these end users as application users with appropriate access privileges and user roles.

Data Realms refer to the subset of data in the database that can be accessed by these application users.

The end users of the application are mapped onto the application users in the database and granted access to appropriate data.

The RAS data security policy associates the application users with appropriate data realms based on the Access Control List(ACL).

Every application user is mapped onto certain role and each role has certain data access privileges. When an application user tries to access certain piece of data in the database, based on the role the user holds the user is mapped onto a data realm. The mapping of roles and their respective data realm is defined in the ACL(Access Control List).

## Application Sessions in RAS

- Application sessions are created when an end user uses the application through a client.
- Each application session is associated with a RAS protected database session.
- RAS Application Sessions represent the end users and their security context within the database in a secure and efficient manner.



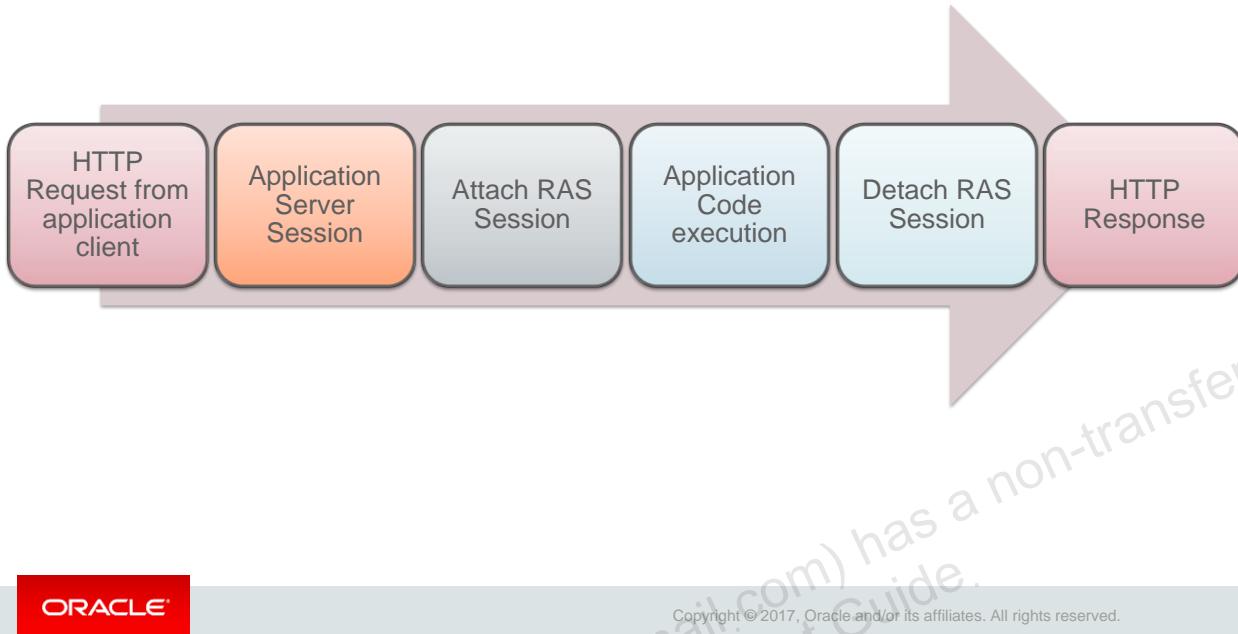
ORACLE

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

When an application client accesses the application an application session starts, traditionally multiple application sessions would share database connection to access the database.

While sharing the database connection, the security context of the application session might become obscure. To avoid it Each application session is mapped onto a RAS session in the database.

## RAS Sessions



ORACLE

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

On first login after authentication typically through a web client a HTTP request is sent to the application server. The application server in turn connects to the database and a RAS application session is created for the user encapsulating the user identity and associated roles.

While processing a user request, a database connection is acquired from the pool, and the RAS user application session is attached to the database connection. Oracle RAS introduces the concept of attach and detach of a RAS Application Session to a database session ensuring that only the application user's security context is used for all relevant database operations. Each session is associated with a security policy, which defines the privileges of the user associated with the session.

The application code is executed in the security context of the RAS session and the response is returned to the application server and then to the client as a HTTP response.

## Lesson Agenda

- Real Application Security
- Transparent Data Encryption
- Data Redaction



ORACLE®

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

## Transparent Data Encryption

- Transparent Data Encryption (TDE) enables you to encrypt data stored in tables and tablespaces.
- TDE automatically encrypts data when it is written to disk and decrypts when applications access it.
- TDE provides in-built key management to manage and secure encryption keys.
- You can encrypt a table column or an entire tablespace using TDE.



ORACLE

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

Oracle Database uses authentication, authorization, and auditing mechanisms to secure data in the database, but not in the operating system data files where data is stored. To protect these data files, Oracle Database provides Transparent Data Encryption (TDE). TDE encrypts sensitive data stored in data files. To prevent unauthorized decryption, TDE stores the encryption keys in a security module external to the database, called a keystore.

The TDE master encryption key is stored in an external security module, which can be an Oracle software keystore or hardware keystore. This TDE master encryption key encrypts and decrypts the TDE table key, which in turn encrypts and decrypts data in the table column.

## Encrypting a Table Column Using TDE

- Uses a two-tiered key-based architecture to encrypt and decrypt sensitive table columns.
- A master encryption key is used for encryption and decryption.
- The encryption key is stored in an external security module.
- The external security module can be accessed only by a user with appropriate privileges.
- TDE uses a single TDE table key regardless of the number of encrypted columns.



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

## Encrypting a Tablespace Using TDE

- All the objects created in an encrypted tablespace are automatically encrypted.
- The encryption key for the tablespace is stored in an external module.
- Allows index range scans on data in encrypted tablespaces unlike TDE column encryption.



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

## Keystores in TDE

- Keystore is responsible for storing the encryption keys of both columns and tablespaces.
- Oracle Database provides a key management framework for TDE that stores and manages keys and credentials.
- Oracle Database supports both software and hardware keystores.



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

Oracle Database supports software keystores and hardware (HSM-based) keystores.

You can configure the following types of software keystores:

- **Password-based software keystores:** Password-based software keystores are protected by using a password that you create. You must open this type of keystore before the keys can be retrieved or used.
- **Auto-login software keystores:** Auto-login software keystores are protected by a system-generated password, and don't need to be explicitly opened by a security administrator. Auto-login software keystores are automatically opened when accessed. Auto-login software keystores can be used across different systems. If your environment doesn't require the extra security provided by a keystore that must be explicitly opened for use, then you can use an auto-login software keystore. Auto-login software keystores are ideal for unattended scenarios.
- **Local auto-login software keystores:** Local auto-login software keystores are auto-login software keystores that are local to the computer on which they are created. Local auto-login keystores can't be opened on any computer other than the one on which they are created. This type of keystore is typically used for scenarios where additional security is required (that is, to limit the use of the auto-login for that computer) while supporting an unattended operation.

Hardware Security Modules are physical devices that provide secure storage for encryption keys, in hardware keystores. HSMs also provide secure computational space (memory) to perform encryption and decryption operations.

## Lesson Agenda

- Real Application Security
- Transparent Data Encryption
- Data Redaction



ORACLE®

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

## Oracle Data Redaction

- It is the ability to redact or mask sensitive data in real time.
- You can mask data that is returned from queries issued by applications.
- You use data redaction when you must disguise sensitive data.



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

Use Oracle Data Redaction when you must disguise sensitive data that your applications and application users must access.

Data Redaction enables you to easily disguise the data using several different redaction styles.

Oracle Data Redaction is ideal for situations in which you must redact specific characters out of the result set of queries of Personally Identifiable Information (PII) returned to certain application users. For example, you may want to present a U.S. Social Security number that ends with the numbers 4320 as \*\*\*-\*\*-4320.

Oracle Data Redaction is particularly suited for call center applications and other applications that are read-only. Take care when using Oracle Data Redaction with applications that perform updates back to the database, because redacted data can be written back to this database.

Oracle Database applies the redaction at runtime, when users access the data (that is, at query-execution time). This solution works well in a production system. During the time that the data is being redacted, all of the data processing is performed normally, and the back-end referential integrity constraints are preserved.

## Data Redaction Methods

You can implement data redaction using one of the following methods:

- Full Redaction
- Partial Redaction
- Regular expressions
- Random redaction
- No redaction



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

**Full redaction:** You redact all of the contents of the column data. The redacted value returned to the querying application user depends on the data type of the column. For example, columns of the NUMBER data type are redacted with a zero (0), and character data types are redacted with a single space.

**Partial redaction:** You redact a portion of the column data. For example, you can redact a Social Security number with asterisks (\*), except for the last 4 digits.

**Regular expressions:** You can use regular expressions to look for patterns of data to redact. For example, you can use regular expressions to redact email addresses, which can have varying character lengths. It is designed for use with character data only.

**Random redaction:** The redacted data presented to the querying application user appears as randomly generated values each time it is displayed, depending on the data type of the column.

**No redaction:** The None redaction type option enables you to test the internal operation of your redaction policies, with no effect on the results of queries against tables with policies defined on them. You can use this option to test the redaction policy definitions before applying them to a production environment.

## Benefits of Data Redaction

- Data Redaction is well suited to environments in which data is constantly changing.
- You can create the Data Redaction policies in one central location and easily manage them from there.
- The Data Redaction policies enable you to create a wide variety of function conditions based on user input.



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

Even though Oracle Data Redaction is not designed to prevent data exposure to database users who run ad hoc queries directly against the database, it can provide an additional layer to reduce the chances of accidental data exposure. Because such users may have rights to change data, alter the database schema, and circumvent the SQL query interface entirely, it is possible for a malicious user to bypass Data Redaction policies in certain circumstances.

## Summary

- Oracle Database provides various security mechanisms to secure data in the database.
- Real Application Security extends application session security to database sessions.
- Transparent Data Encryption secures data on the disk by using encryption techniques.
- Data Redaction protects sensitive data by obscuring it from unintended users.



ORACLE

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

Unauthorized reproduction or distribution prohibited. Copyright© 2019, Oracle and/or its affiliates.

PELISSIER LOIC (loic.pelissier@gmail.com) has a non-transferable  
license to use this Student Guide.