

Oracle Database 19c: Advanced PL/SQL

Electronic Presentation
D1101121GC10

Learn more from Oracle University at education.oracle.com



Author
Brent Dayley

Copyright © 2020, Oracle and/or its affiliates.

**Technical Contributor
and Reviewer**
Don Bates

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.

Editors
Aju Kumar
Moushmi Mukherjee

The information contained in this document is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

Publishers
Sujatha Nagendra
Asief Baig

1011172020

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 END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government's use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

Trademark Notice

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

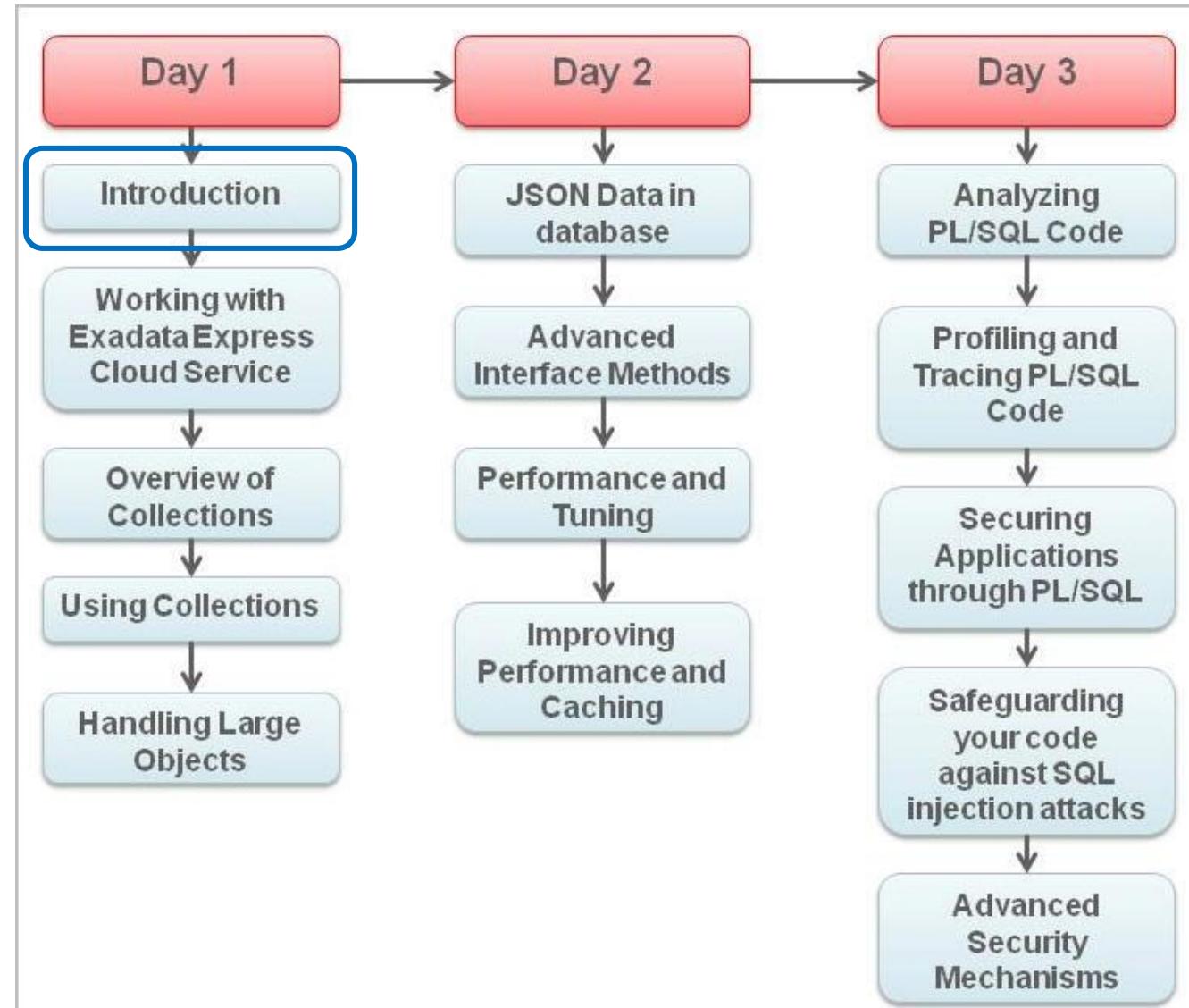
Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

Third-Party Content, Products, and Services Disclaimer

This documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

Introduction

Course Agenda



Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle Database 19c and related products
- Oracle Cloud
- Oracle documentation and additional resources
- New Features



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 such as joins and nested queries.
- You have a 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.

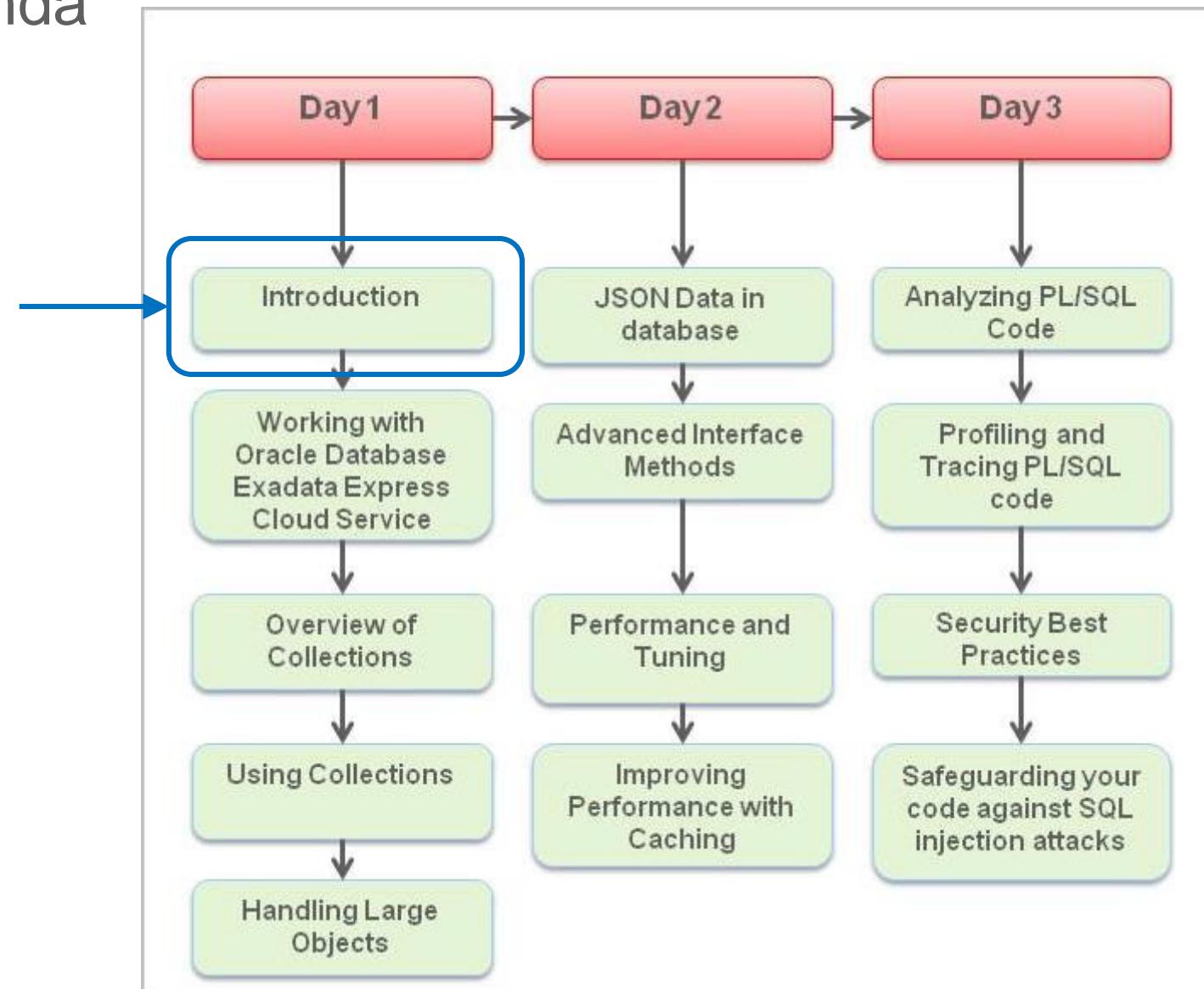
Course Objectives

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
- Identify the security requirements of applications
- Manage PL/SQL code effectively



Course Agenda



Appendices Used in This Course

- Appendix A: Table Descriptions and Data
- Appendix B: Using SQL Developer
- Appendix C: Using 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 19c and related products
- Oracle Cloud
- Oracle documentation and additional resources
- New Features



PL/SQL Development Environments

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

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

Oracle 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.

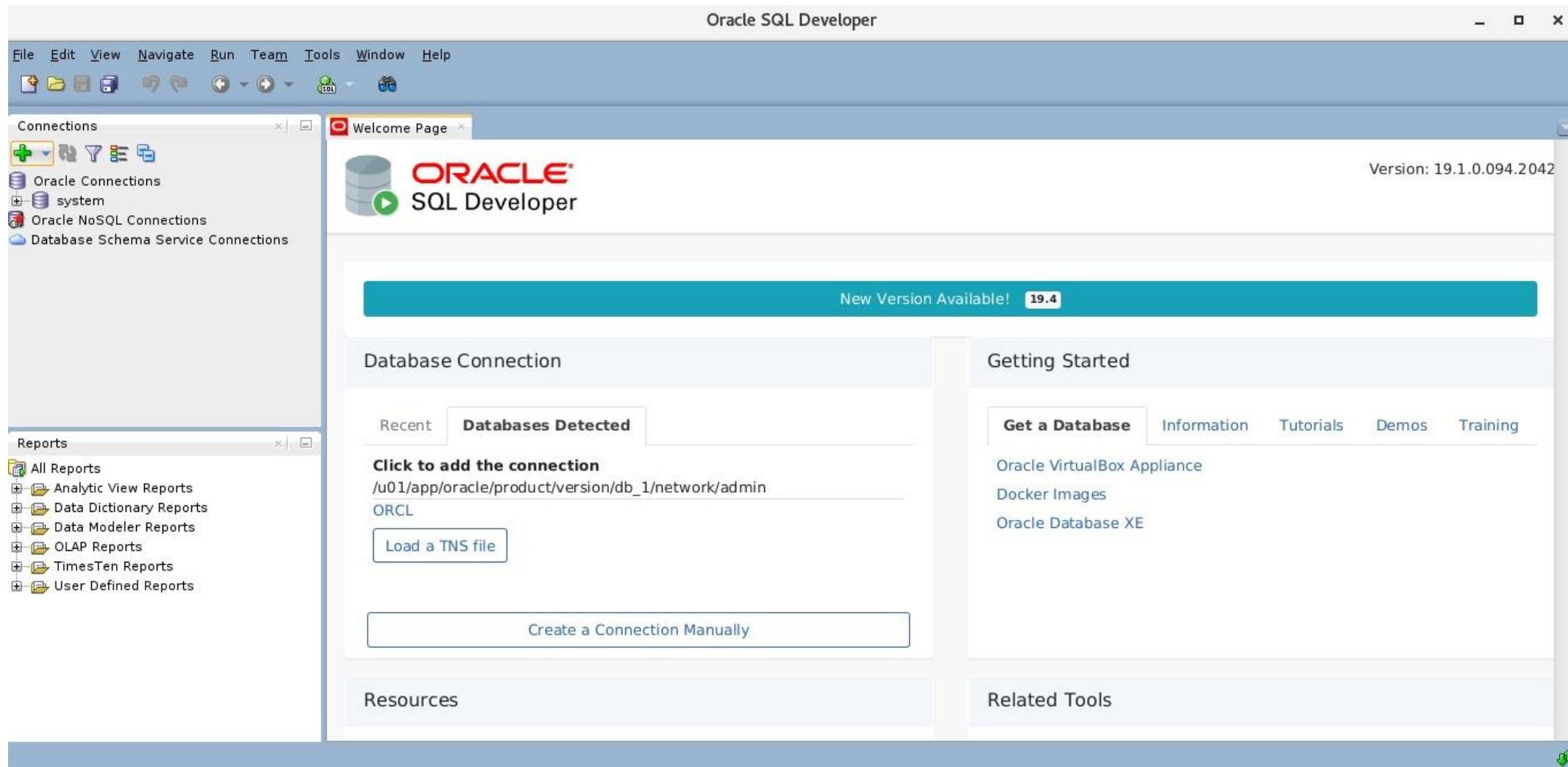


SQL Developer

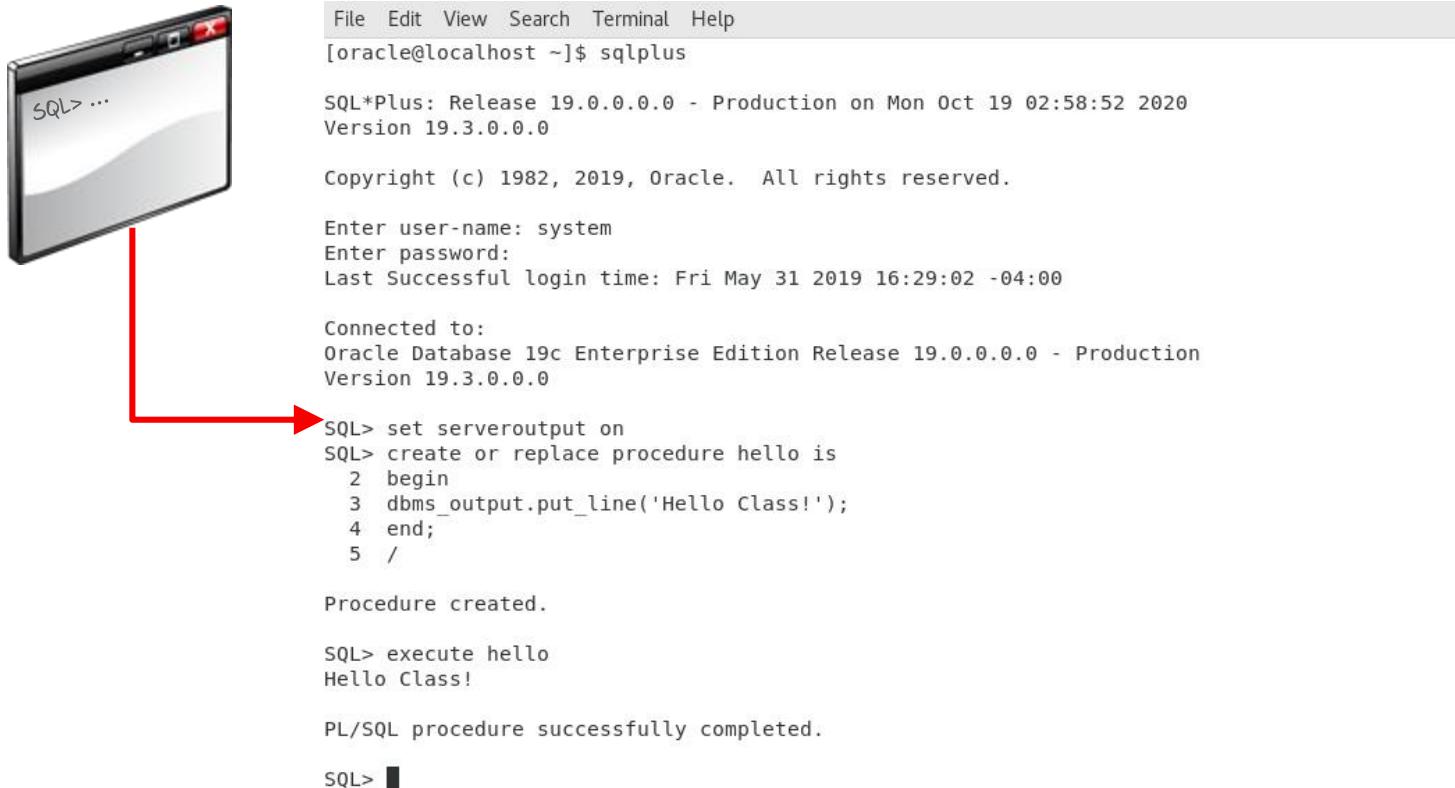
Specifications of SQL Developer

- 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

SQL Developer 19.x.x Interface



Coding PL/SQL in SQL*Plus



A screenshot of the Oracle SQL*Plus command-line interface. The window title bar says "File Edit View Search Terminal Help". The command prompt shows "[oracle@localhost ~]\$ sqlplus". The output pane displays the following session:

```
SQL*Plus: Release 19.0.0.0.0 - Production on Mon Oct 19 02:58:52 2020
Version 19.3.0.0.0

Copyright (c) 1982, 2019, Oracle. All rights reserved.

Enter user-name: system
Enter password:
Last Successful login time: Fri May 31 2019 16:29:02 -04:00

Connected to:
Oracle Database 19c Enterprise Edition Release 19.0.0.0.0 - Production
Version 19.3.0.0.0

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>
```

A red arrow points from the left margin of the slide towards the "execute hello" command in the SQL*Plus session.

Lesson Agenda

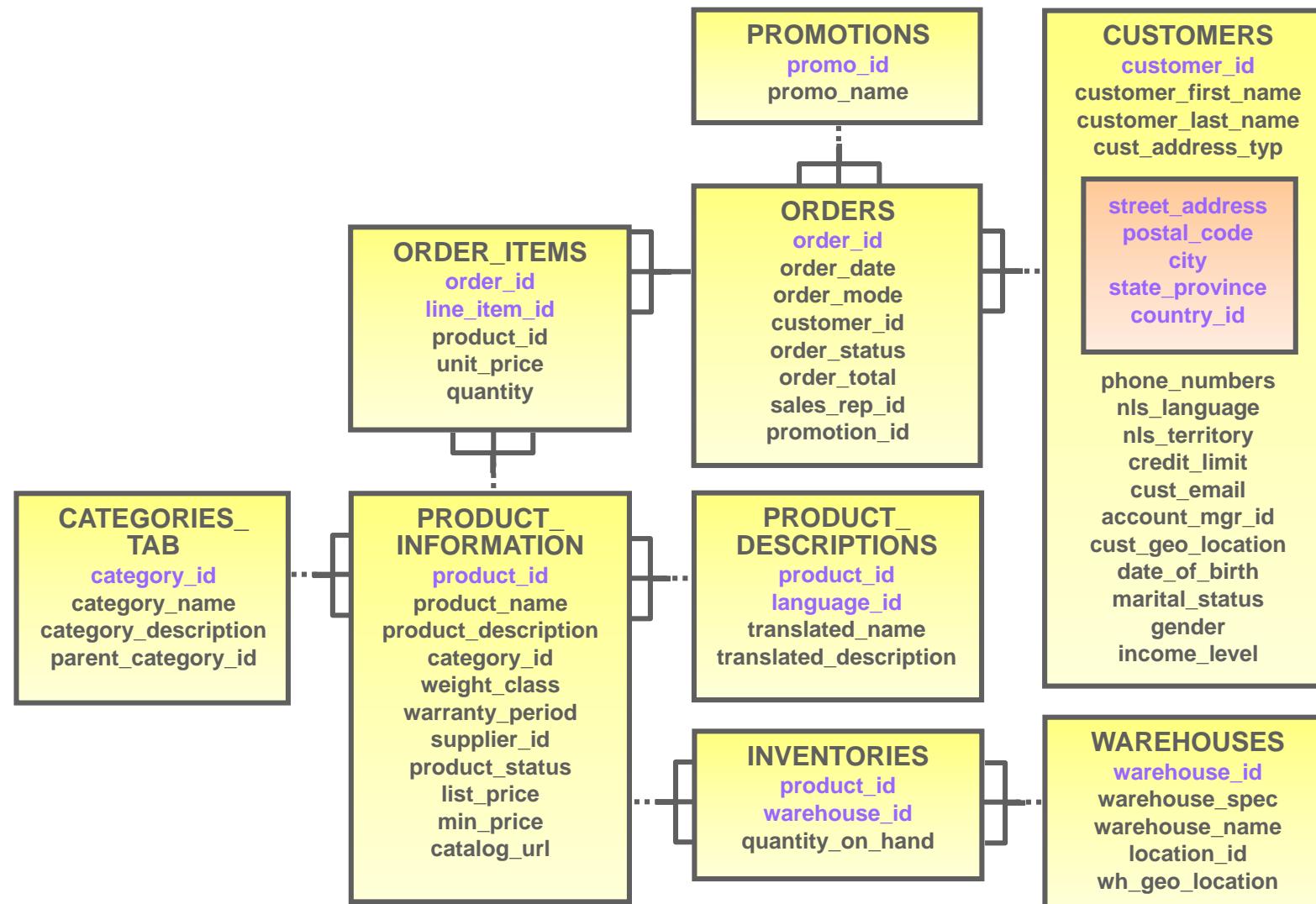
- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle Database 19c and related products
- Oracle Cloud
- Oracle documentation and additional resources
- New Features



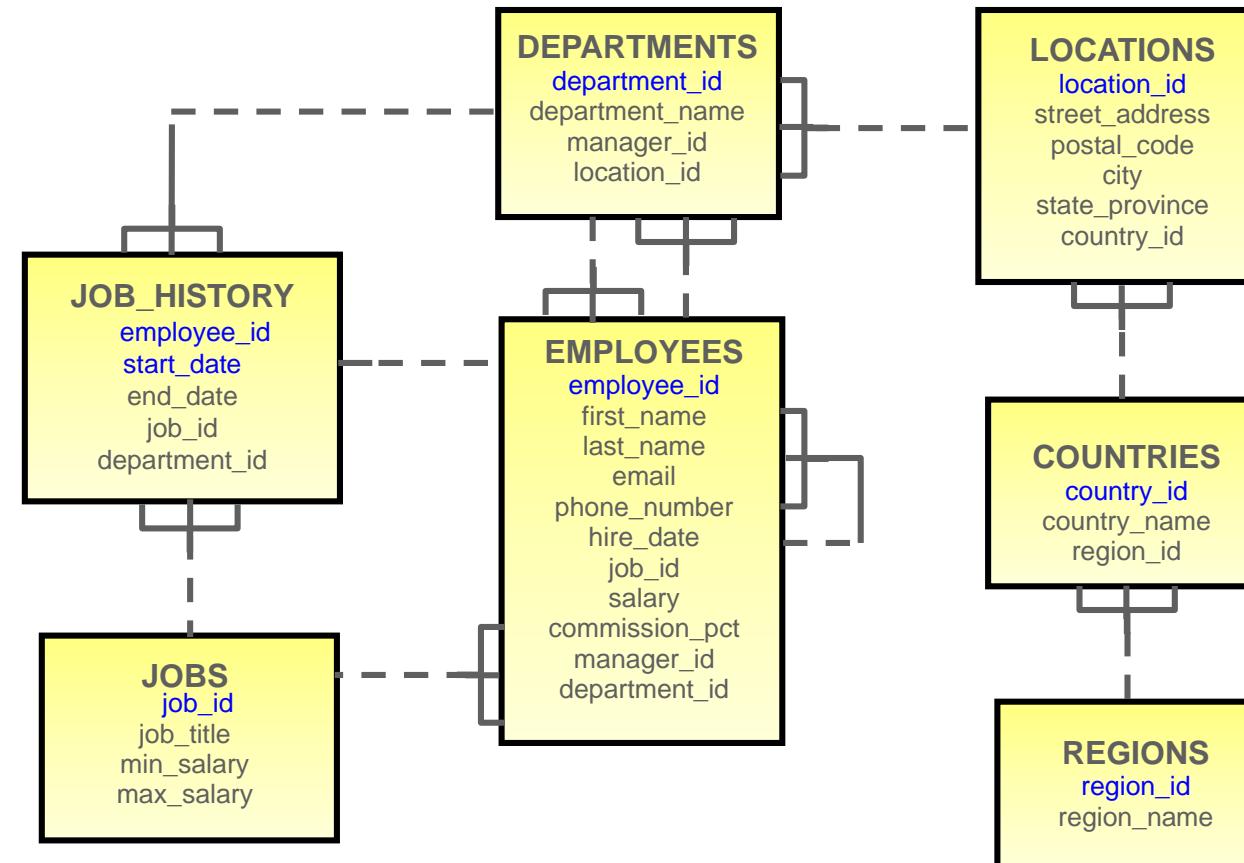
Tables Used in This Course

- 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.

Order Entry Schema



Human Resources Schema



Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle Database 19c and related products
- Oracle Cloud
- Oracle documentation and additional resources
- New Features



Oracle Database 19c: Focus Areas

Information
Management



Oracle Cloud



Application
Development



Infrastructure
Grids

Oracle Database 19c



High Availability



Performance



Security



Manageability



Information
Integration

Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle Database 19c and related products
- Oracle Cloud
- Oracle documentation and additional resources
- New Features

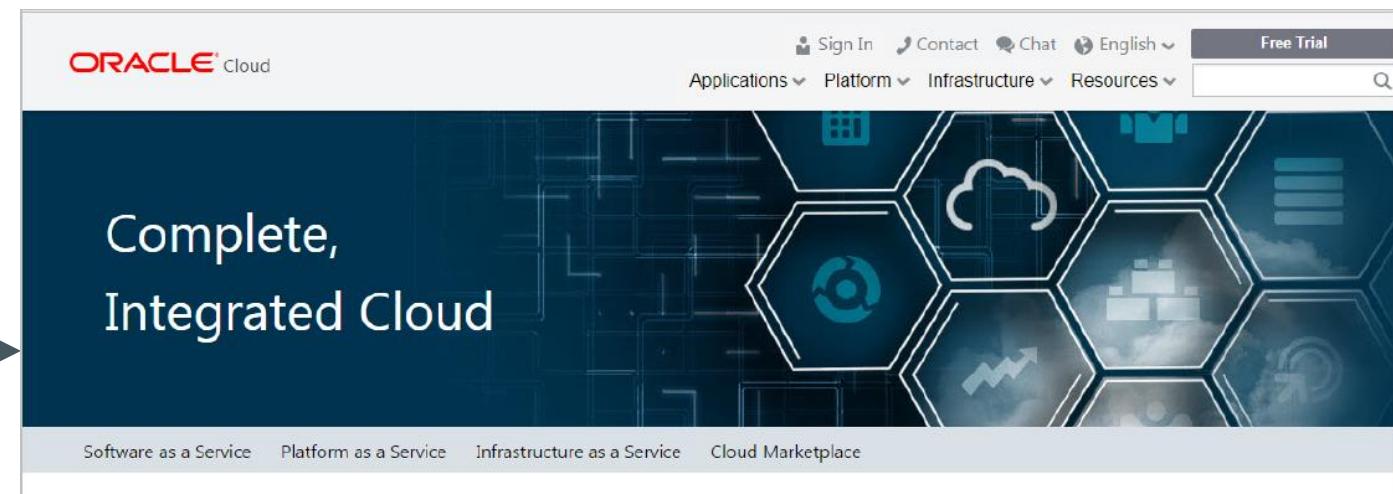


Introduction to Oracle Cloud

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



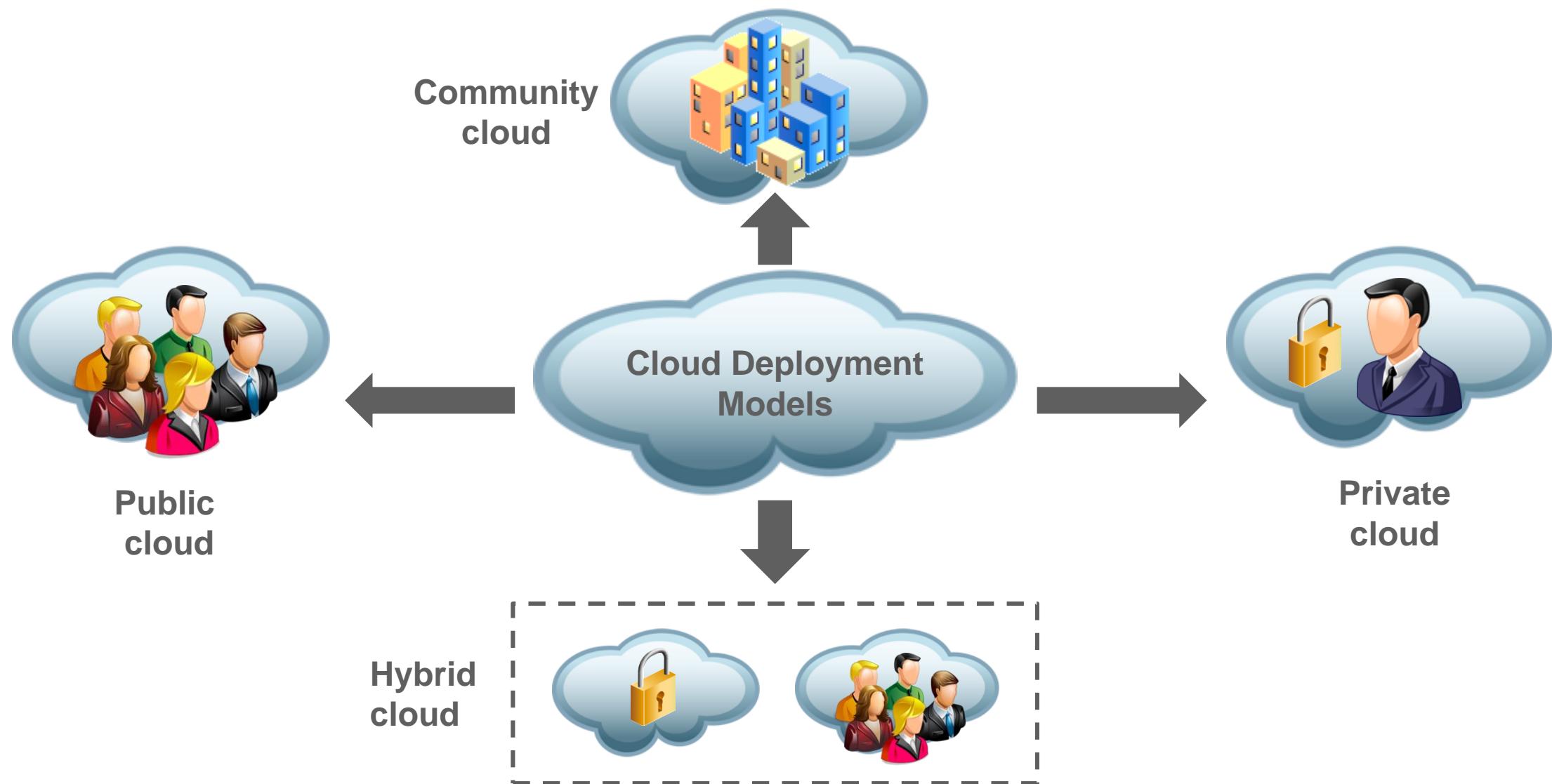
Oracle Cloud Services

Oracle Cloud provides the following three types of services:

- Software as a Service (SaaS)
- Platform as a Service (PaaS)
- Infrastructure as a Service (IaaS)



Cloud Deployment Models



Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle Database 19c and related products
- Oracle Cloud
- Oracle documentation and additional resources
- New Features



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*

Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Identifying the tables and schemas used in this course
- Overview of Oracle Database 19c and related products
- Oracle Cloud
- Oracle documentation and additional resources
- New Features



New Features in 18c and 19c



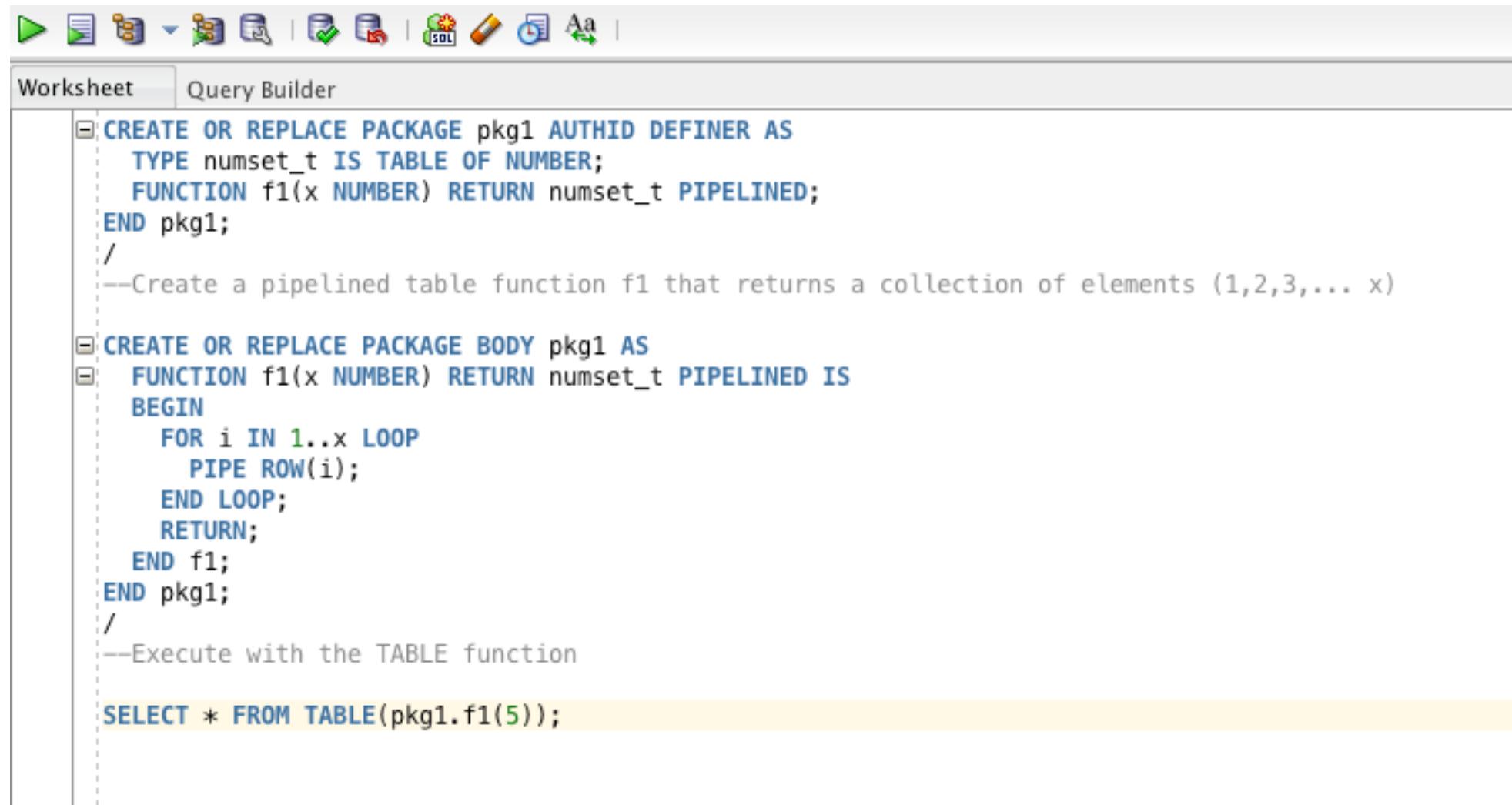
There are several new features in 18c and 19c:

- Polymorphic Table Functions
- Qualified Expressions
- SQL Macros
- Non-Persistence support for Object Types

Polymorphic Table Functions

- A polymorphic table function (PTF) is a new type of table function whose return type is determined by the arguments passed into the PTF.
- A table function is a user-defined PL/SQL function that returns a collection of rows (an associative array, nested table, or varray) that can be called from the `FROM` clause of a SQL query block.
- A PTF is useful when SQL developers and database administrators want to provide generic extensions which work for arbitrary input tables or queries.

Polymorphic Table Functions



The screenshot shows the Oracle SQL Developer interface with the 'Worksheet' tab selected. The code area contains the following PL/SQL package definition:

```
CREATE OR REPLACE PACKAGE pkg1 AUTHID DEFINER AS
  TYPE numset_t IS TABLE OF NUMBER;
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED;
END pkg1;
/
--Create a pipelined table function f1 that returns a collection of elements (1,2,3,... x)

CREATE OR REPLACE PACKAGE BODY pkg1 AS
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED IS
    BEGIN
      FOR i IN 1..x LOOP
        PIPE ROW(i);
      END LOOP;
      RETURN;
    END f1;
  END pkg1;
/
--Execute with the TABLE function

SELECT * FROM TABLE(pkg1.f1(5));
```

Qualified Expressions

- Starting with Oracle Database Release 18c, any PL/SQL value can be provided by an expression (for example for a record or for an associative array), like a constructor provides an abstract datatype value.
- In PL/SQL, we use the terms "qualified expression" and "aggregate" rather than the SQL term "type constructor"; however, the functionality is the same.
- Qualified expressions use an explicit type indication to provide the type of the qualified item. This explicit indication is known as a typemark.
- The syntax for qualified expressions is:

```
qualified_expression ::= typemark ( aggregate )
aggregate ::= [ positional_choice_list ] [ explicit_choice_list ]
positional_choice_list ::= (expr )+
explicit_choice_list ::= named_choice_list | indexed_choice_list
named_choice_list ::= identifier => expr [,] +
indexed_choice_list ::= expr => expr [,] +
```

Qualified Expressions

This example uses a function to display the values of a table of BOOLEAN:

```
CREATE FUNCTION print_bool (v IN BOOLEAN)
    RETURN VARCHAR2
IS
    v_rtn VARCHAR2(10);
BEGIN
    CASE v
    WHEN TRUE THEN
        v_rtn := 'TRUE';
    WHEN FALSE THEN
        v_rtn := 'FALSE';
    ELSE
        v_rtn := 'NULL';
    END CASE;
    RETURN v_rtn;
END print_bool;
```

The `v_aa1` variable is initialized using index key-value pairs:

```
DECLARE
    TYPE t_aa IS TABLE OF BOOLEAN INDEX BY PLS_INTEGER;
    v_aa1 t_aa := t_aa(1=>FALSE,
                        2=>TRUE,
                        3=>NULL);
BEGIN
    DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(1)));
    DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(2)));
    DBMS_OUTPUT.PUT_LINE(print_bool(v_aa1(3)));
END;

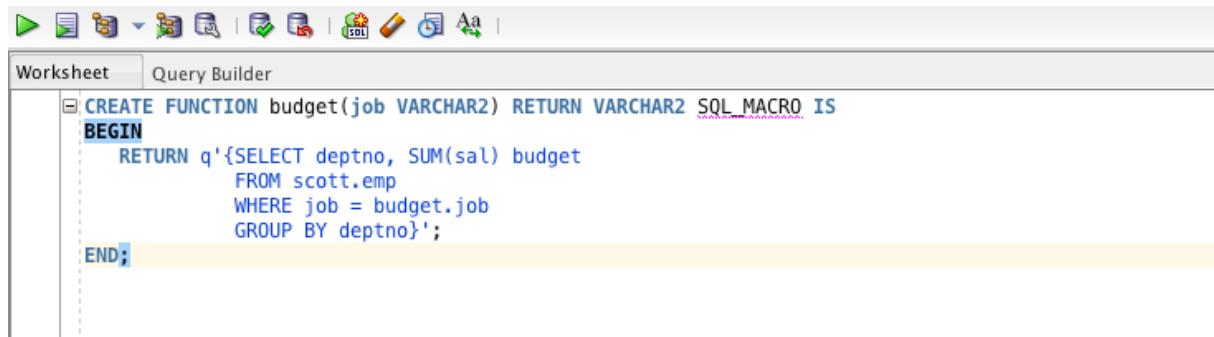
FALSE
TRUE
NULL
```

SQL Macros

- Starting in 19c, you can create SQL macros (SQM) to factor out common SQL expressions and statements into reusable, parameterized constructs that can be used in other SQL statements.
- SQL table macros increase developer productivity, simplify collaborative development, and improve code quality.
- The AUTHID property cannot be specified.
- A SQL macro can appear only in the `FROM` clause of a query table expression.
- A SQL macro cannot appear in a virtual column expression, functional index, editioning view, or materialized view.
- Type methods cannot be annotated with `SQL_MACRO`.

SQL Macros

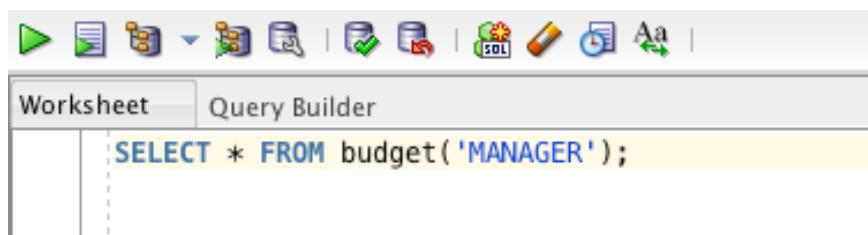
Create the Function:



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

```
CREATE FUNCTION budget(job VARCHAR2) RETURN VARCHAR2 SQL_MACRO IS
BEGIN
    RETURN q'{SELECT deptno, SUM(sal) budget
      FROM scott.emp
     WHERE job = budget.job
   GROUP BY deptno}';
END;
```

Execute the Function:



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

```
SELECT * FROM budget('MANAGER');
```

DEPTNO	BUDGET
20	2975
30	2850
10	2450

Non-Persistence Support for Object Types

- This feature enables you to mark abstract data types used in programs for processing only, for storage only, or for mixed use.
- Instances of non-persistent types cannot persist on disk.
- You cannot specify the [NOT] PERSISTABLE clause in a subtype definition.

Summary

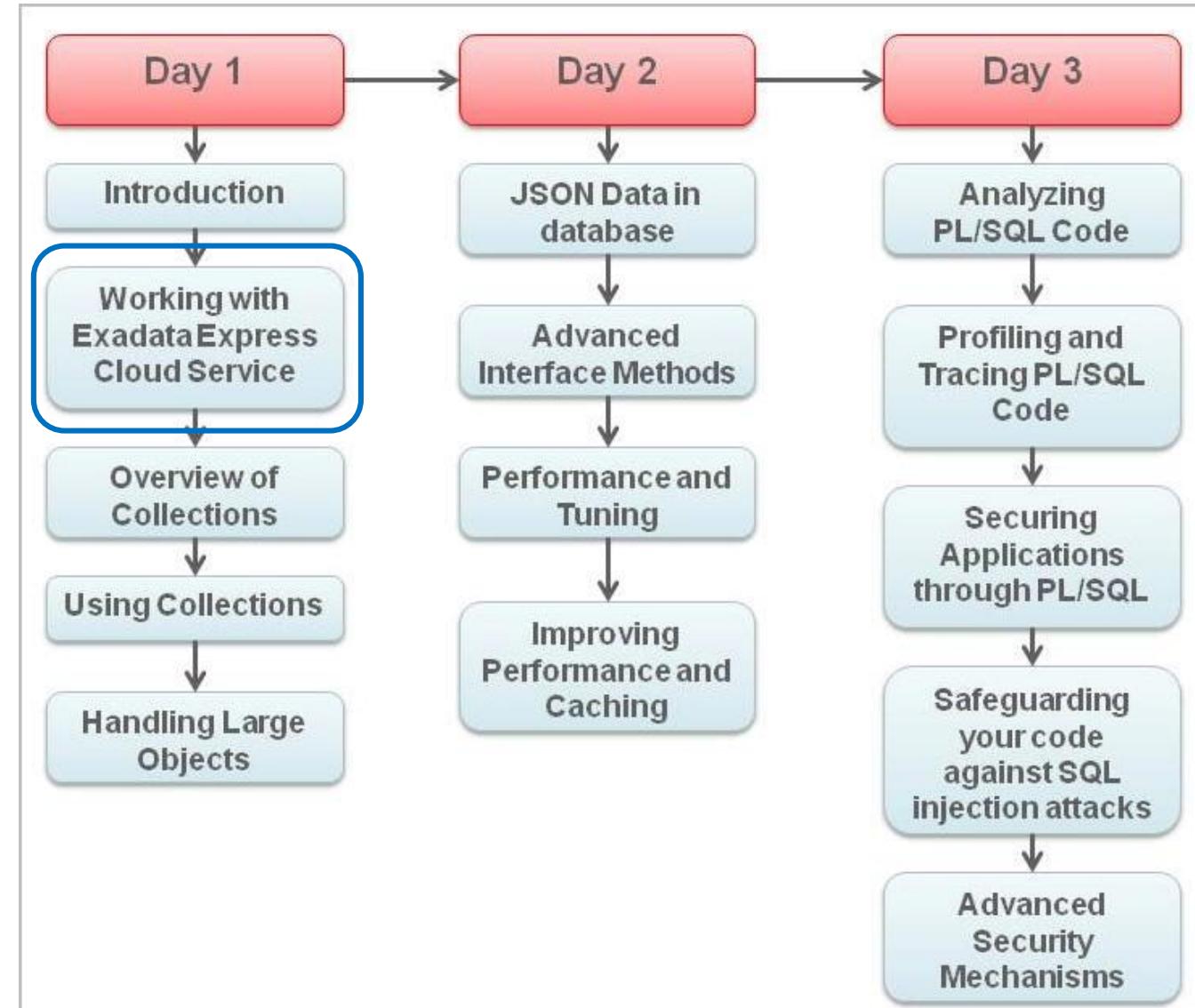
In this lesson, you should have learned how to:

- 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



Working with Exadata Express Cloud Service

Course Agenda



Lesson Objectives

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

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

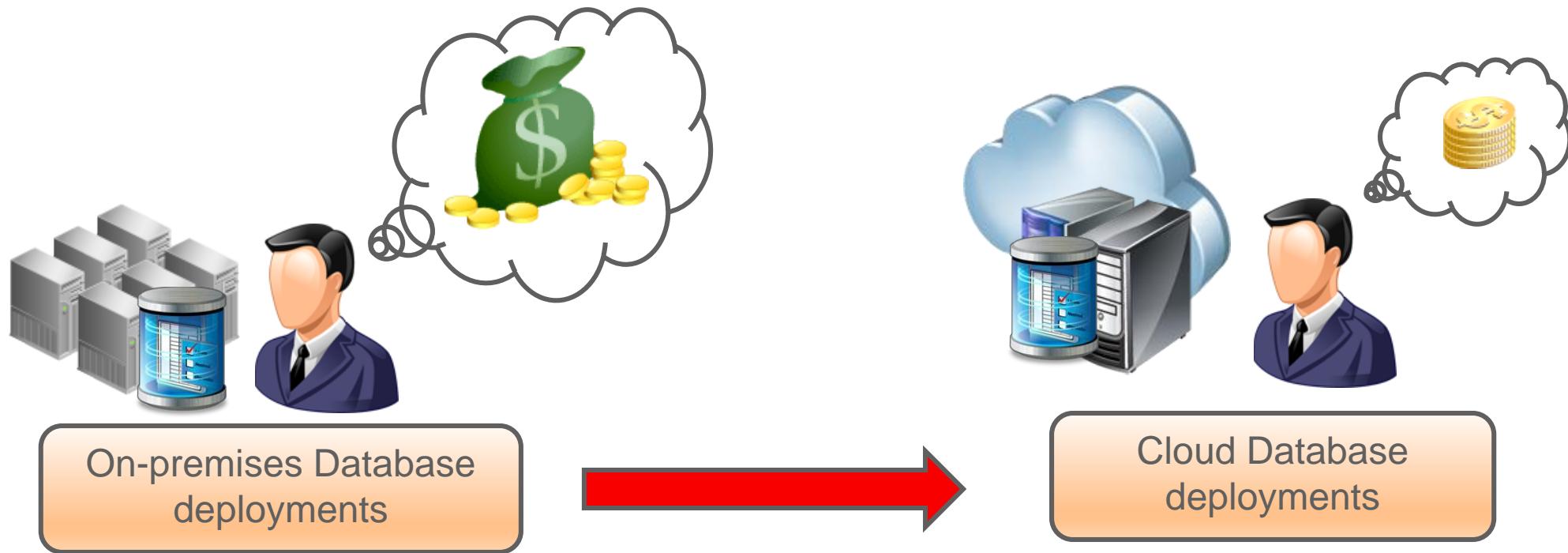


Lesson Agenda

- 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

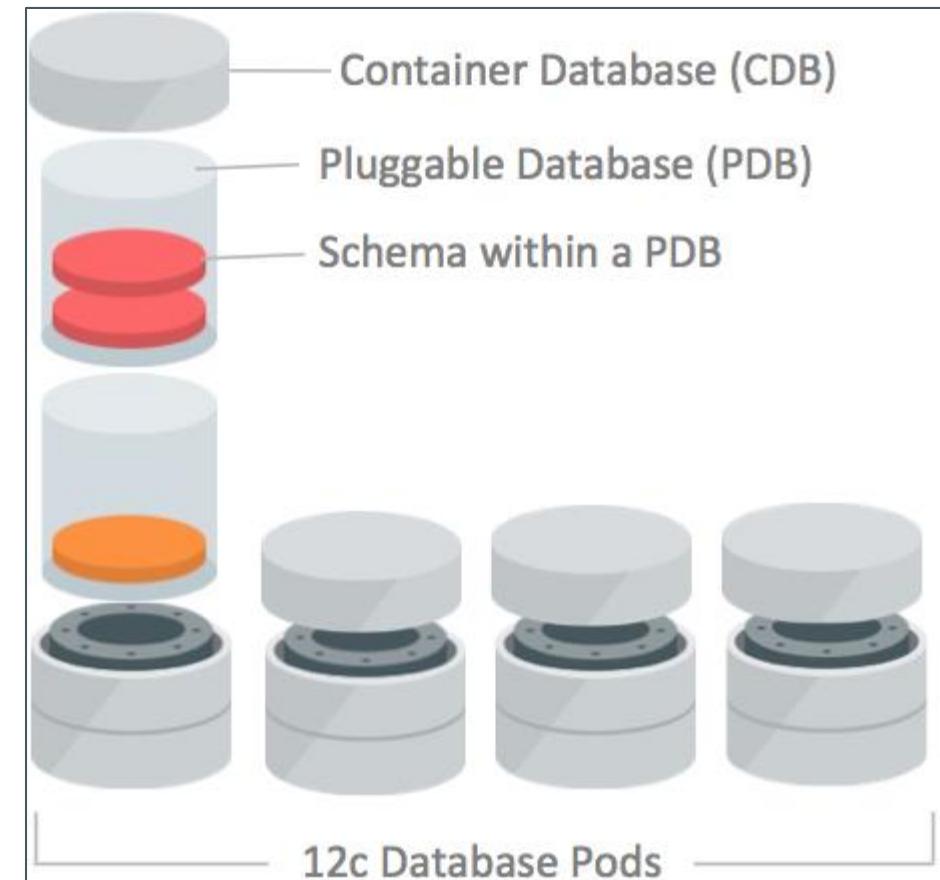


Evolving from On-premises to Exadata Express



Exadata Express for Users

- Oracle manages the service as multiple Container databases (CDBs), also known as database pods.
- Each CDB can accommodate up to 1000 Pluggable databases (PDBs).
- Each user is provisioned with a PDB on subscribing to the service, where the user can create several schemas.



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



Getting Started with Exadata Express

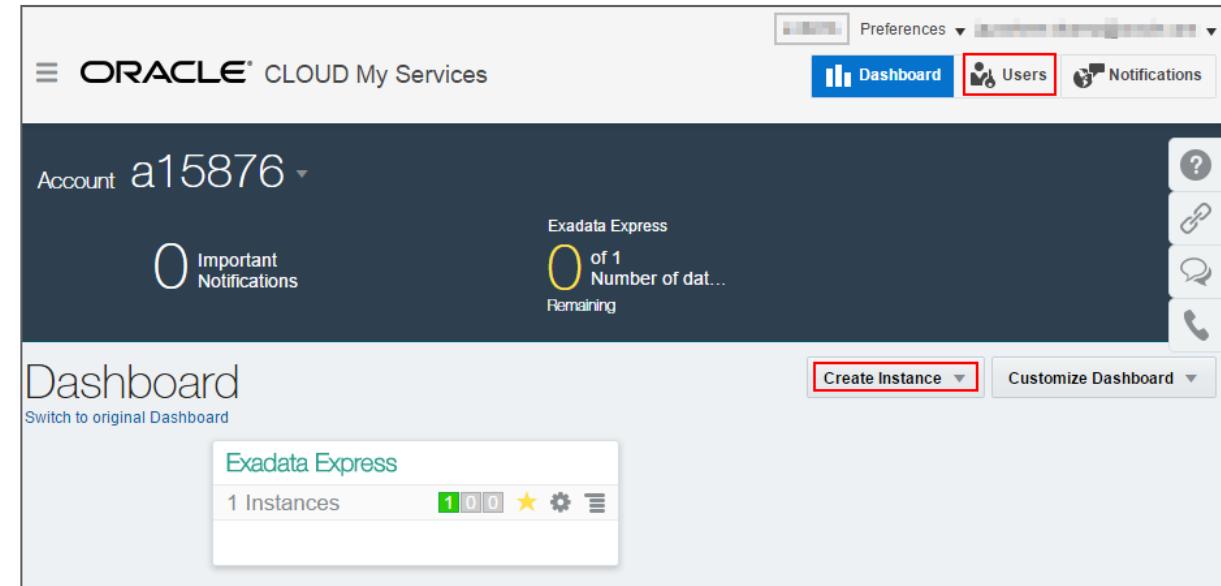
1. Purchase a subscription.
2. Activate and verify the service.
3. Verify activation.
4. Learn about users and roles.
5. Create accounts for your users and assign them appropriate privileges and roles.
6. Set the password for the database user authorized to perform administrative tasks for your service (PDB_ADMIN).

Note: Refer to *Using Oracle Database Exadata Express Cloud Service* for details on the subscription process:

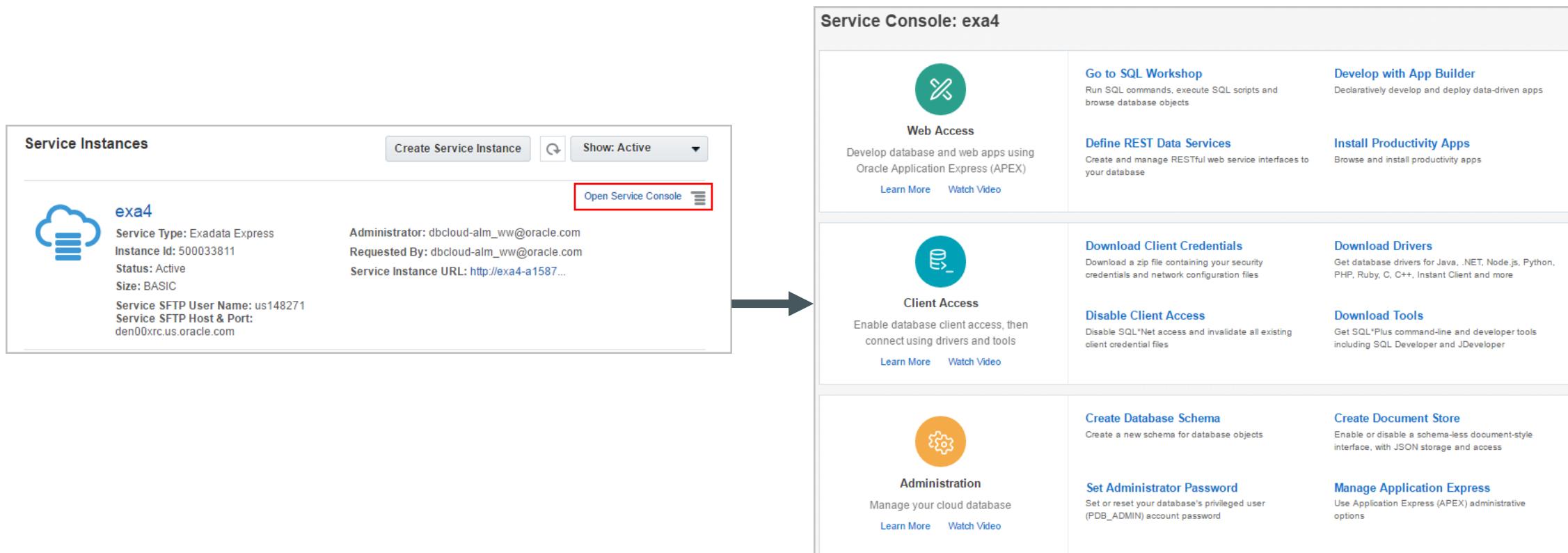
- <https://docs.oracle.com/cloud/latest/exadataexpress-cloud/CSDBP/toc.htm>

Getting Started with Exadata Express

- On signing in to 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.



Managing Exadata



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



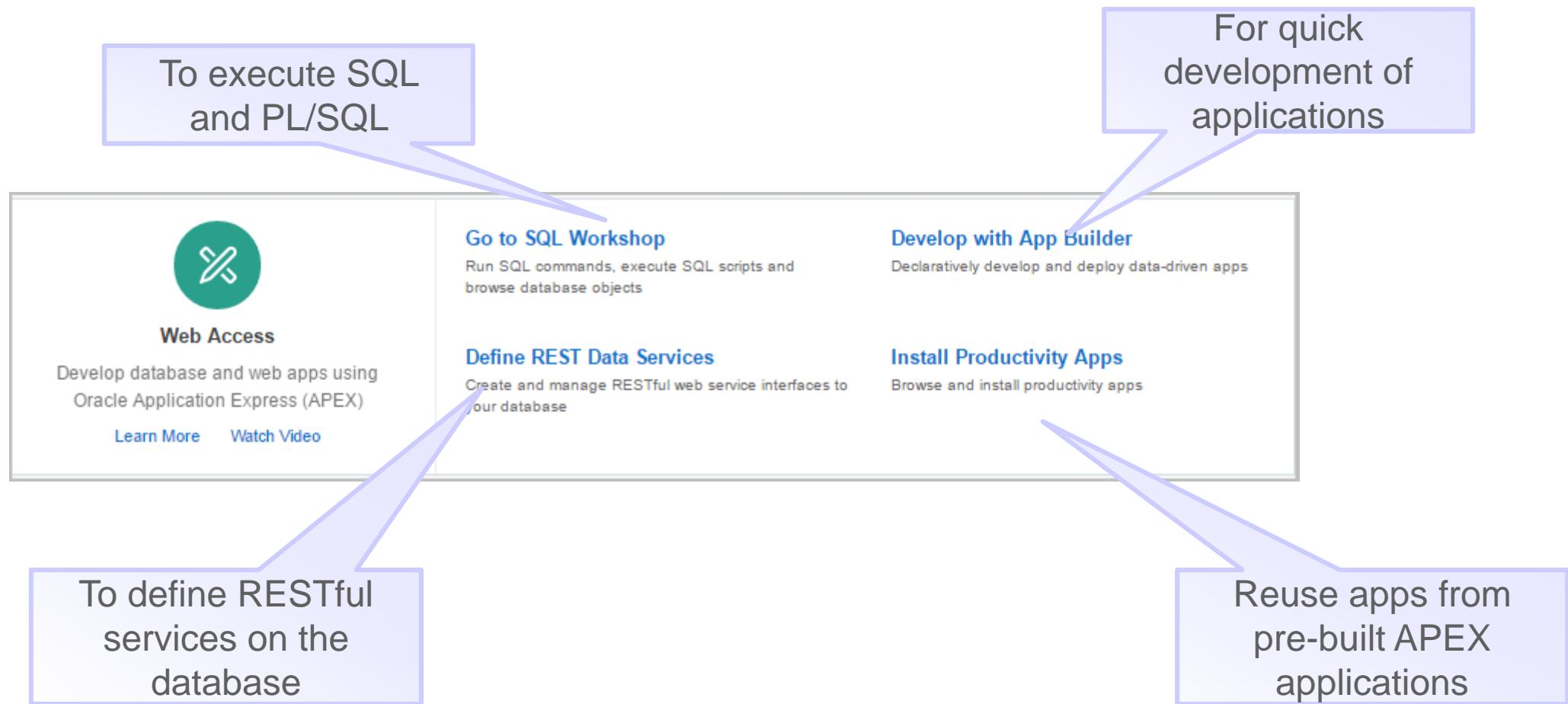
Service Console

- 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

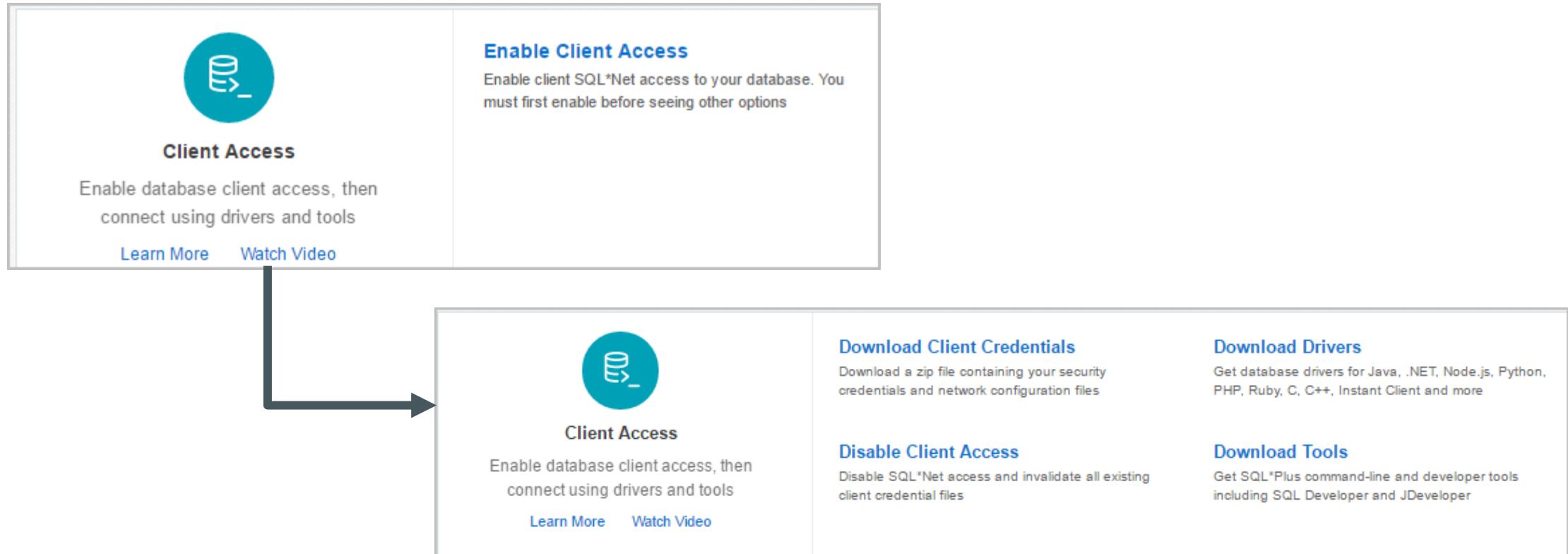
Service Console: exa4

 Web Access Develop database and web apps using Oracle Application Express (APEX) Learn More Watch Video	Go to SQL Workshop Run SQL commands, execute SQL scripts and browse database objects	Develop with App Builder Declaratively develop and deploy data-driven apps
 Client Access Enable database client access, then connect using drivers and tools Learn More Watch Video	Define REST Data Services Create and manage RESTful web service interfaces to your database	Install Productivity Apps Browse and install productivity apps
 Administration Manage your cloud database Learn More Watch Video	Download Client Credentials Download a zip file containing your security credentials and network configuration files	Download Drivers Get database drivers for Java, .NET, Node.js, Python, PHP, Ruby, C, C++, Instant Client and more
	Disable Client Access Disable SQL*Net access and invalidate all existing client credential files	Download Tools Get SQL*Plus command-line and developer tools including SQL Developer and JDeveloper
	Create Database Schema Create a new schema for database objects	Create Document Store Enable or disable a schema-less document-style interface, with JSON storage and access
	Set Administrator Password Set or reset your database's privileged user (PDB_ADMIN) account password	Manage Application Express Use Application Express (APEX) administrative options

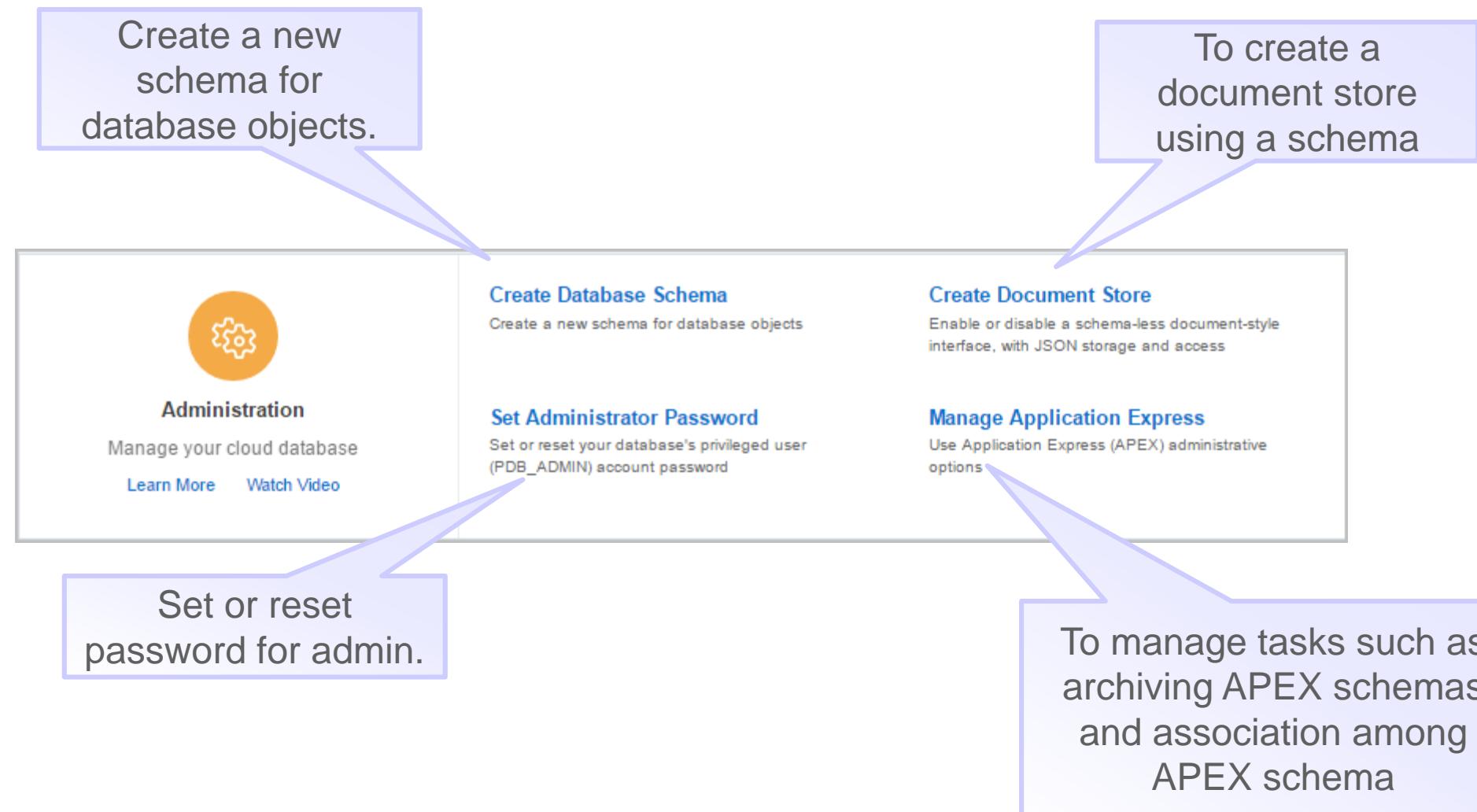
Web Access Through Service Console



Client Access Configuration Through Service Console



Database Administration Through Service Console



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



SQL Workshop

The screenshot shows the Oracle Database homepage with several navigation options:

- Web Access**: Develop database and web apps using Oracle Application Express (APEX). Includes "Learn More" and "Watch Video" links.
- Go to SQL Workshop**: Run SQL commands, execute SQL scripts and browse database objects.
- Develop with App Builder**: Declaratively develop and deploy data-driven apps.
- Define REST Data Services**: Create and manage RESTful web service interfaces to your database.
- Install Productivity Apps**: Browse and install productivity apps.



1

Clicking SQL workshop will lead you to the APEX interface.

The screenshot shows the Oracle Application Express interface with the following tabs in the top navigation bar:

- ORACLE Application Express
- Application Builder
- SQL Workshop** (highlighted with a green bar)
- Team Development
- Packaged Apps

The main content area displays five utility icons:

- Object Browser
- SQL Commands (with a cursor icon over it)
- SQL Scripts
- Utilities
- RESTful Services

2

To run SQL or PL/SQL, you can use the SQL commands utility.

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. The top navigation bar includes tabs for Application Builder, SQL Workshop (which is selected), Team Development, and Packaged Apps. Below the tabs, there's a search icon, a help icon, and a user profile icon. The main workspace is titled "SQL Commands" and shows the schema "BZJNMKSSA". A "Rows" dropdown is set to 10, and there are buttons for "Clear Command" and "Find Tables". On the right side of the workspace are "Save" and "Run" buttons. The SQL command entered is "SELECT * FROM emp;". The results section displays a table with columns: EMPNO, ENAME, JOB, MGR, HIREDATE, SAL, COMM, and DEPTNO. The data rows are:

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

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 other options like Help and Log In. The main workspace is titled "SQL Commands" and shows the following PL/SQL code:

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

Below the code, there are buttons for "Save" and "Run". The "Run" button is highlighted in blue. The results section at the bottom displays the output of the executed code:

Results	Explain	Describe	Saved SQL	History
Hello World				
Statement processed.				
0.34 seconds				

The results show the text "Hello World" followed by the message "Statement processed." and a execution time of "0.34 seconds".

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



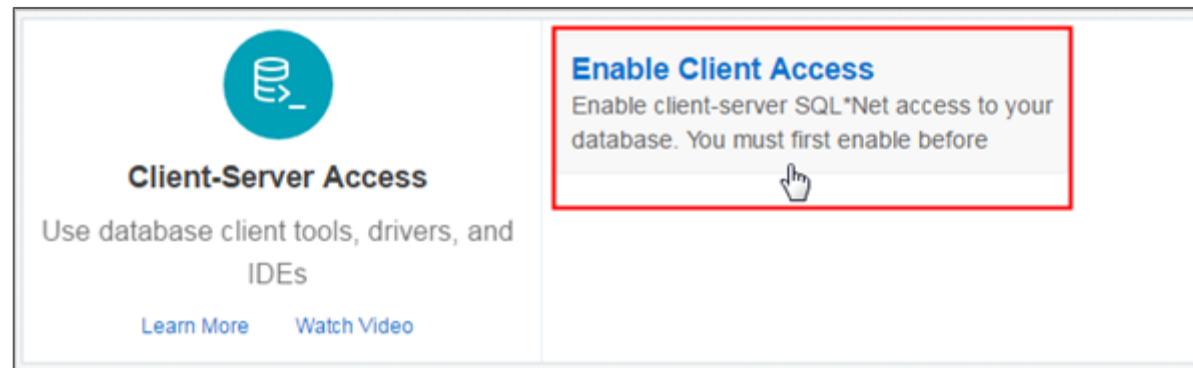
Connecting Through Database Clients

You can connect to Exadata Express by using various database clients:

- SQL*Plus
- SQLcl
- SQL Developer
- .Net and Visual Studio
- JDBC Thin Client

Enabling SQL*Net Access for Client Applications

Enable SQL*Net Access in the Service Console to obtain the various Database Client options.



Downloading Client Credentials

Client-Server Access

Use database client tools, drivers, and IDEs

[Learn More](#) [Watch Video](#)

Download Client Credentials
Download a zip file containing your security credentials and network configuration files

Disable Client Access
Disable SQL*Net access and invalidate all existing client credential files

1

Download Client Credentials

Provide a password for your client credentials, then click **Download** below.

Password *
.....

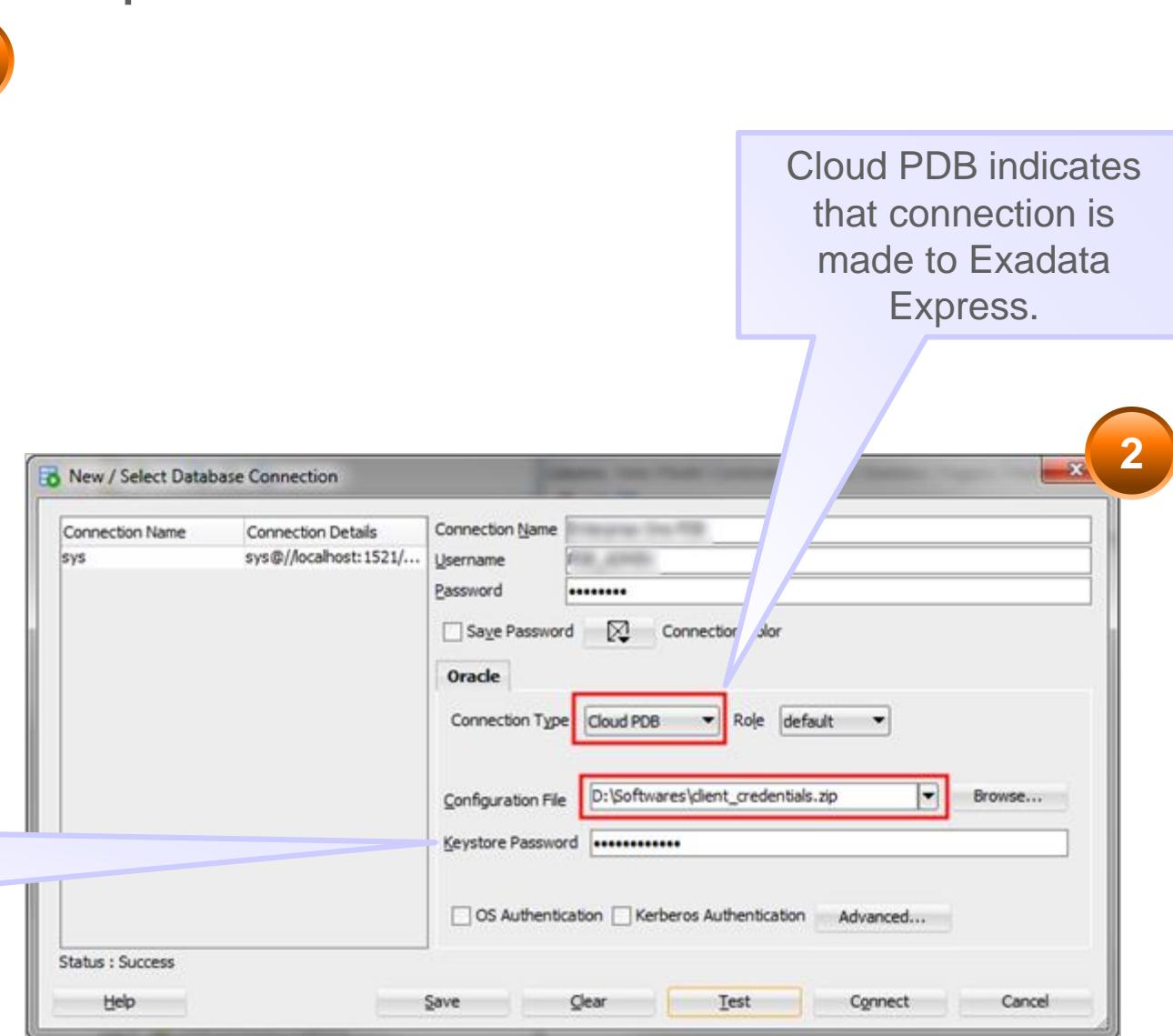
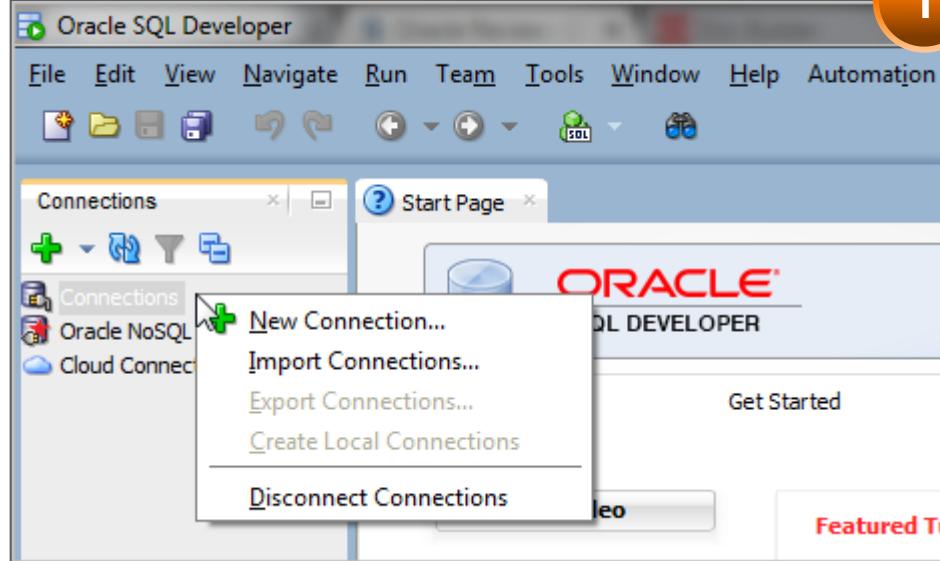
Confirm Password *
..... Passwords match!

The client credentials file is required in addition to entering database username and password to access your cloud database. This zip file includes an Oracle Wallet and Java Keystore containing a client certificate. You may wish to share it with authorized users who need to connect to your cloud database. Please keep it secure to avoid unauthorized access.

Cancel **Download**

2

Connecting Oracle SQL Developer



Connecting Oracle SQLcl

```
D:\PDB Service\SQL CL\sqlcl-no-jre-latest\sqlcl\bin>sql /nolog
```

```
SQLcl: Release 20.2.0.174.1557 RC on Tue Sep 08 12:18:07 2020
```

```
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

Summary

In this lesson, you should have learned about:

- Oracle Database Exadata Express Cloud Service
- The 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



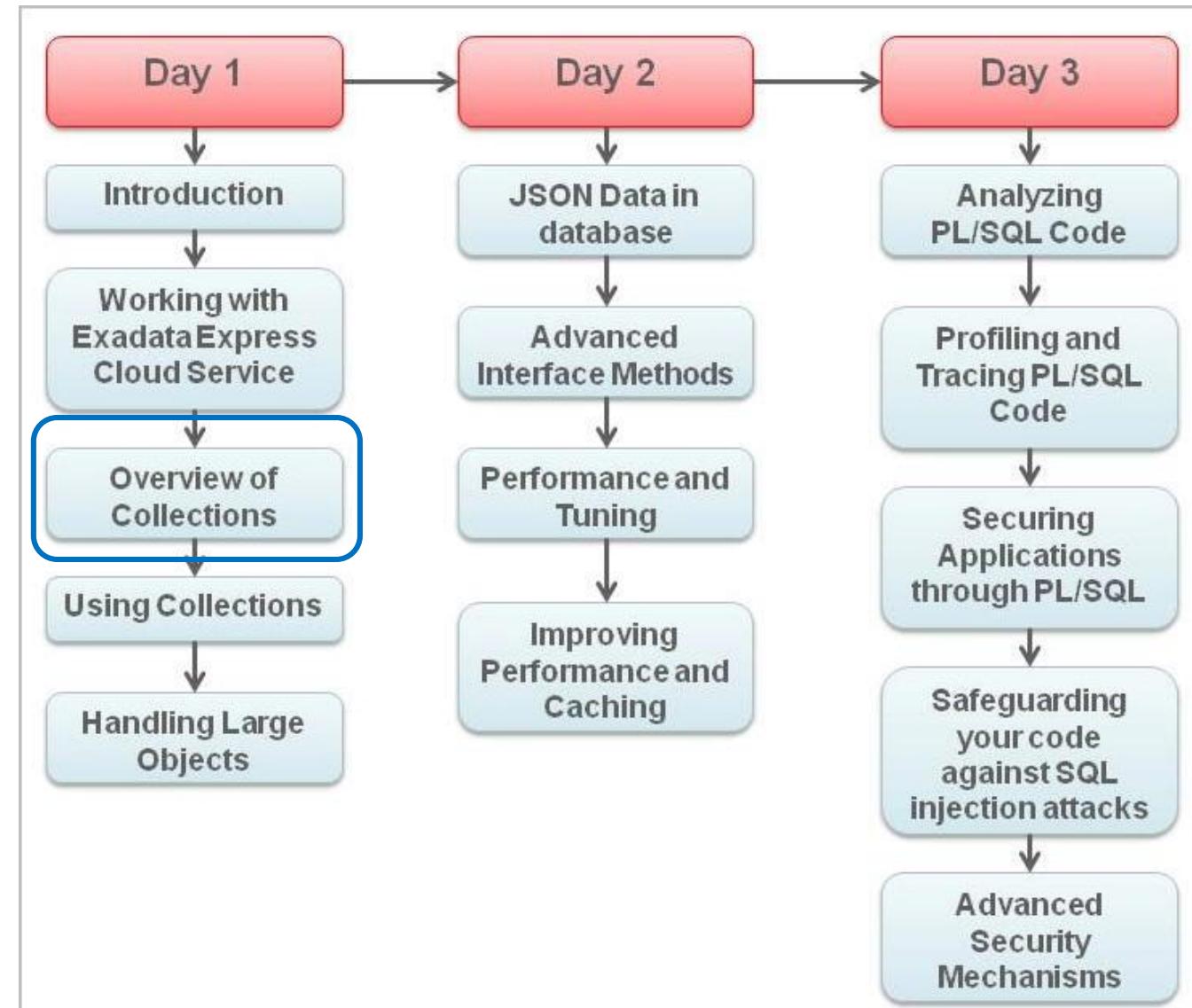
Practice 2: Overview

There is no practice for lesson 2.



Overview of Collections

Course Agenda



Objectives

After completing this lesson, you should be able to create and traverse:

- Associative arrays
- Varrays
- Nested tables



Lesson Agenda

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



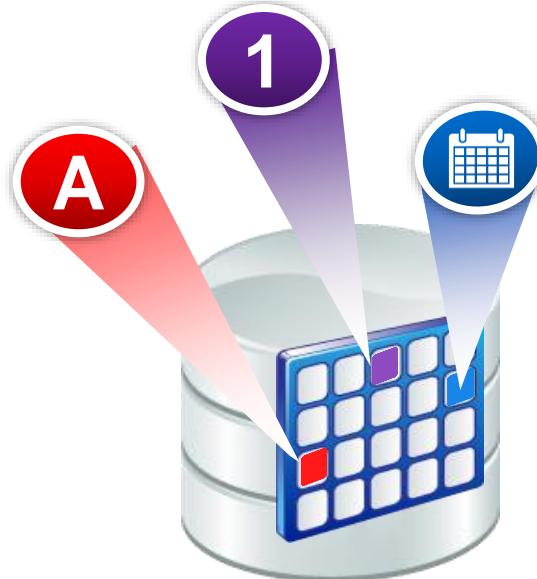
Collections

- 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



Why Collections?

We have variables.



We have cursors.

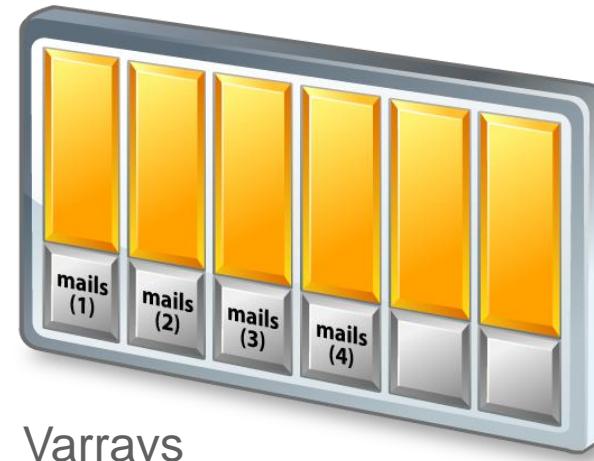


Then why collections?

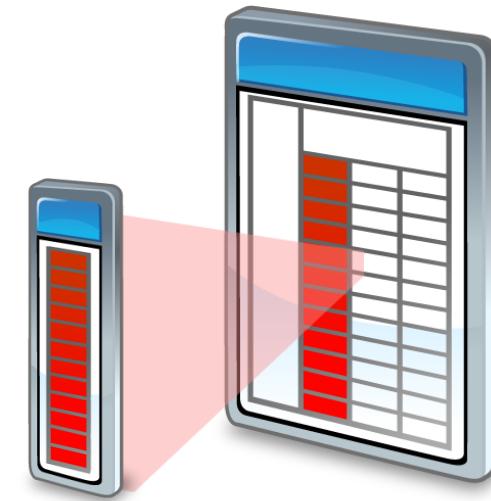
Collection Types



Associative Arrays



Varrays



Nested Tables

Lesson Agenda

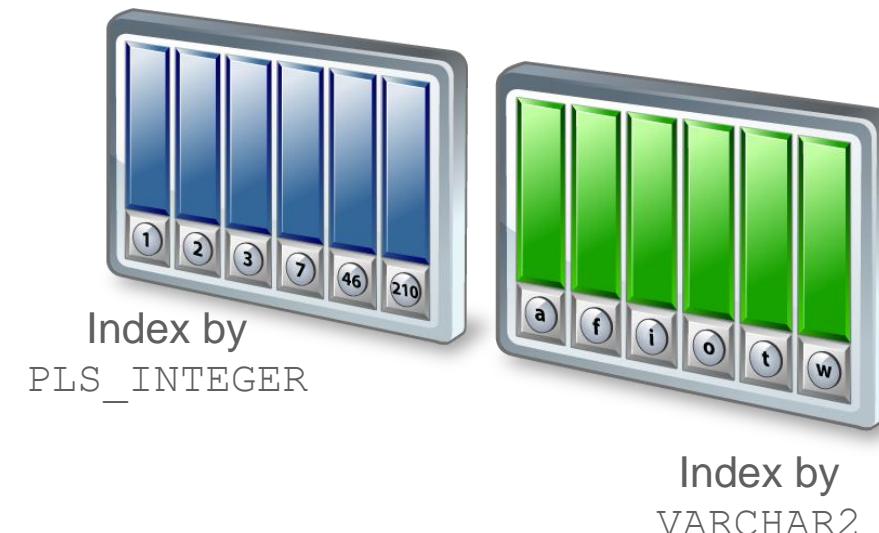
- Understanding Collections
- Creating Associative Arrays
- Creating Nested Tables
- Creating Varrays



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



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;
```

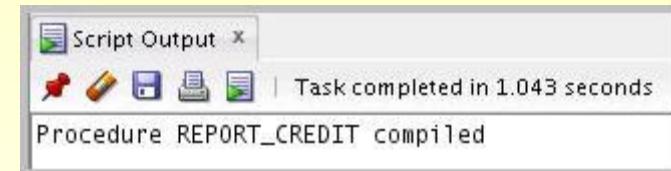
Create the string-indexed associative array type.

Create the string-indexed associative array variable.

Populate the string-indexed associative array variable.

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

Collection Methods

- A collection method is a PL/SQL subprogram.
- 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
```

Lesson Agenda

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



Nested Tables

- A collection 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.
- Values can be stored as a column type in a table.
- Each value in a nested table is accessed through an index starting from 1.

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];
```

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;
```



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');
```

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

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)));
```

1

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

2

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

1

2

Using Nested Tables

- Querying the results:

```
select * from porder;
```

The screenshot shows a 'Query Result' window in Oracle SQL Developer. The title bar says 'Query Result x'. Below it are icons for Run, Stop, Refresh, and SQL. The status bar indicates 'All Rows Fetched: 2 in 0.002 seconds'. The main area is a table with the following data:

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

The screenshot shows a 'Query Result' window in Oracle SQL Developer. The title bar says 'Query Result x'. Below it are icons for Run, Stop, Refresh, and SQL. The status bar indicates 'All Rows Fetched: 4 in 0.002 seconds'. The main area is a table with the following data:

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

Referencing Collection Elements

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

- Syntax:

```
collection_name(subscript)
```

- Example:

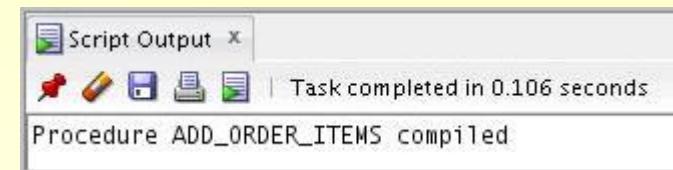
```
v_with_discount(i)
```

- To reference a field in a collection:

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

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;
```



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	500 555
2	500 566
3	500 577
4	800 61.75
5	800 39.9
6	800 760
7	800 13.3

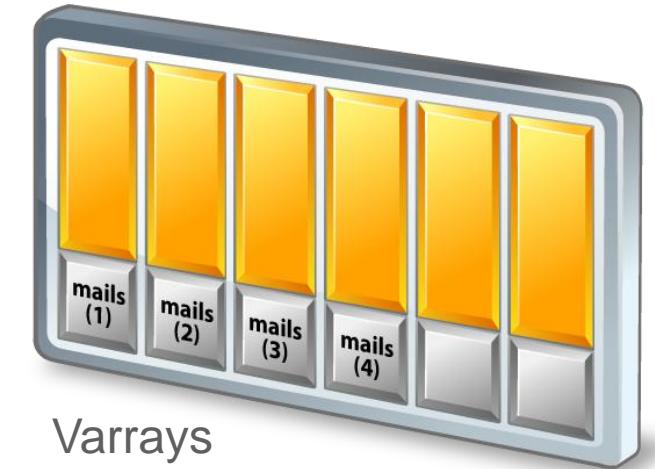
Lesson Agenda

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



Varrays

- Are variable sized arrays
- Can hold data values from zero to a declared maximum size
- Can be stored in the database as a table column



Varrays

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];
```

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

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

1

2

Using Varrays

- Querying the results:

```
SELECT * FROM department;
```

The screenshot shows a 'Query Result' window from Oracle SQL Developer. The title bar says 'Query Result x'. Below it, there are icons for Run, Stop, Refresh, and a red X. The status bar indicates 'SQL | All Rows Fetched: 2 in 0.478 seconds'. The main area is a grid with four columns: DEPT_ID, NAME, BUDGET, and PROJECTS. Row 1 has DEPT_ID 1, NAME 'Executive Administration', BUDGET 30000000, and PROJECTS '0E.TYP_PROJECTLIST([0E.TYP_PROJECT], [0E.TYP_PROJECT])'. Row 2 has DEPT_ID 2, NAME 'Information Technology', BUDGET 5000000, and PROJECTS '0E.TYP_PROJECTLIST([0E.TYP_PROJECT])'.

DEPT_ID	NAME	BUDGET	PROJECTS
1	Executive Administration	30000000	0E.TYP_PROJECTLIST([0E.TYP_PROJECT], [0E.TYP_PROJECT])
2	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;
```

The screenshot shows a 'Query Result' window from Oracle SQL Developer. The title bar says 'Query Result x'. Below it, there are icons for Run, Stop, Refresh, and a red X. The status bar indicates 'SQL | All Rows Fetched: 3 in 0.005 seconds'. The main area is a grid with five columns: DEPT_ID, NAME, PROJECT_NO, TITLE, and COST. Row 1 has DEPT_ID 1, NAME 'Executive Administration', PROJECT_NO 1001, TITLE 'Travel Monitor', and COST 400000. Row 2 has DEPT_ID 1, NAME 'Executive Administration', PROJECT_NO 1002, TITLE 'Open World', and COST 10000000. Row 3 has DEPT_ID 2, NAME 'Information Technology', PROJECT_NO 2001, TITLE 'DB11gR2', and COST 900000.

DEPT_ID	NAME	PROJECT_NO	TITLE	COST
1	Executive Administration	1001	Travel Monitor	400000
2	Executive Administration	1002	Open World	10000000
3	Information Technology	2001	DB11gR2	900000

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



Quiz



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

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



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



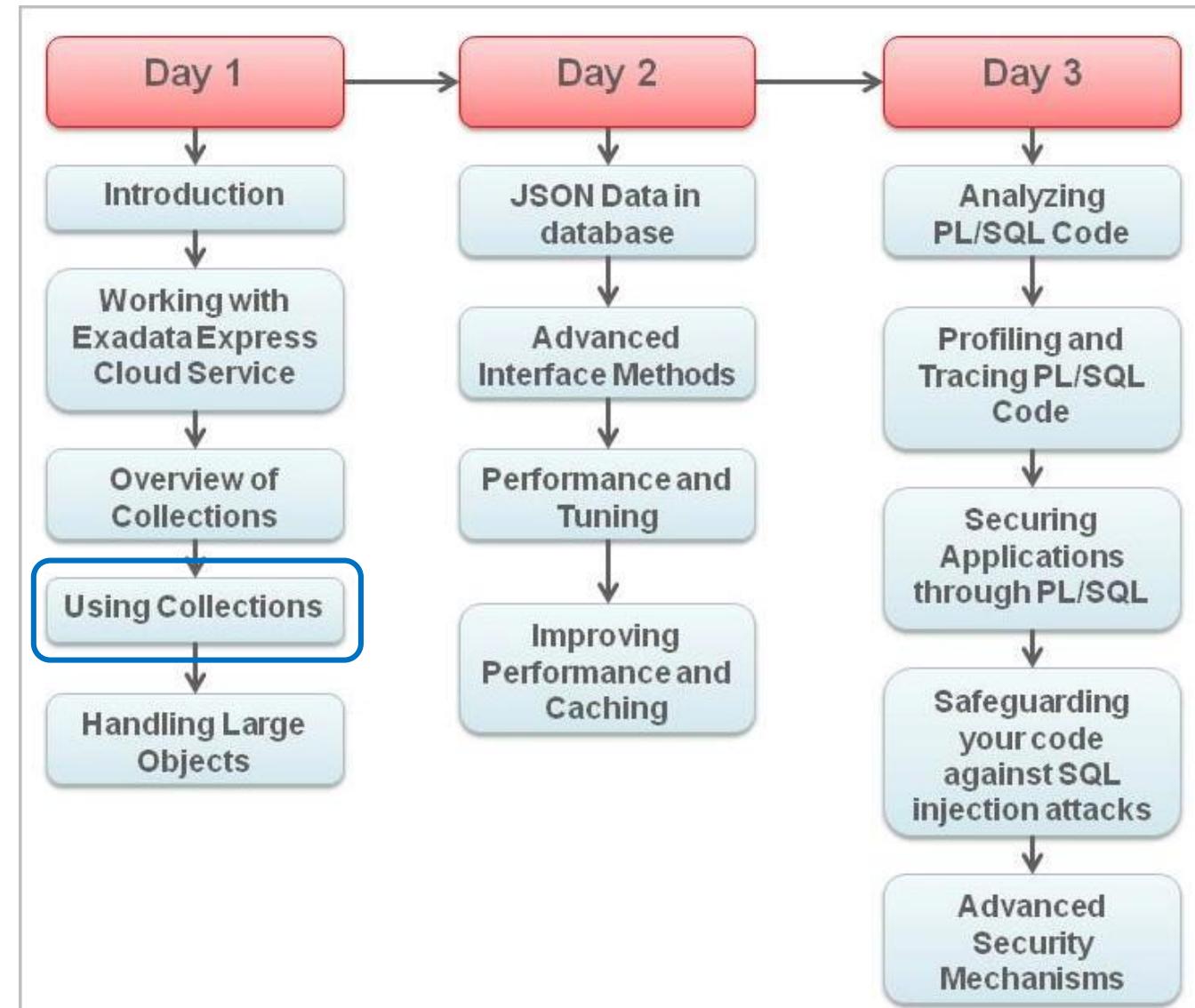
Practice 3: Overview

In this practice, you analyze collections for common errors, and create a collection.



Using Collections

Course Agenda



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



Lesson Agenda

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

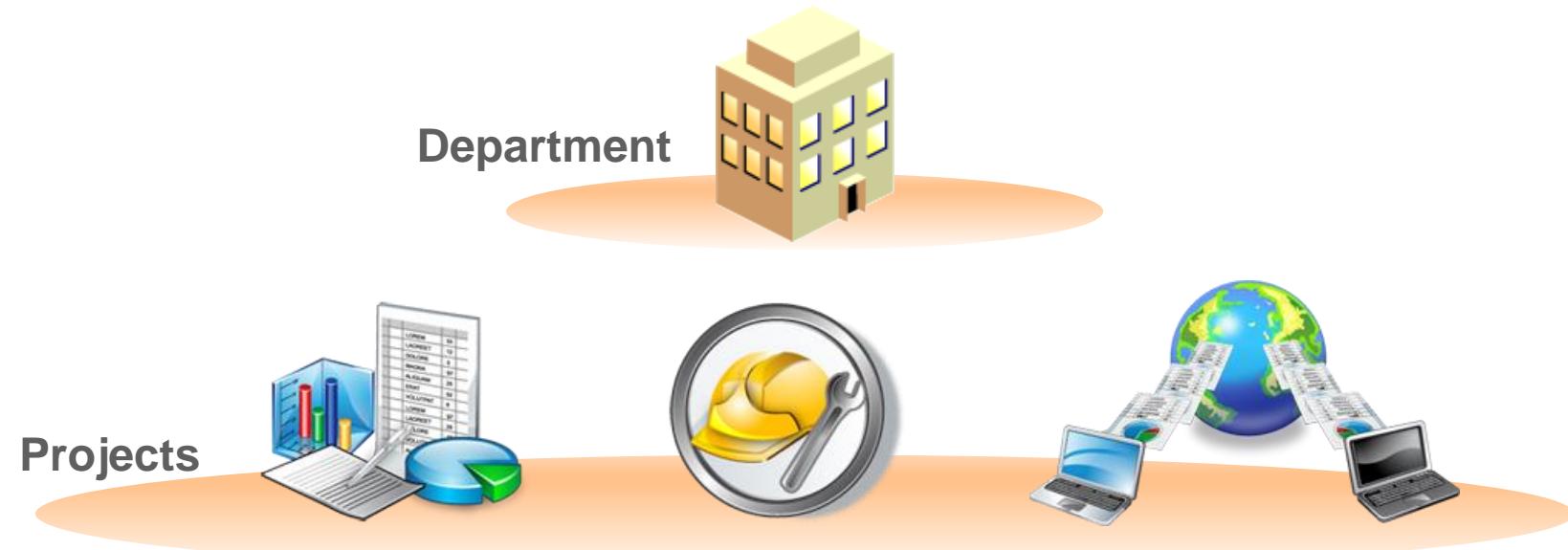


Usage of Collections in Applications

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.



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;
```

Assigning Values to Collection Variables

You can initialize a constructor by 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 subprogram.
- Fetch from the database into the collection.

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;
```

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;
```

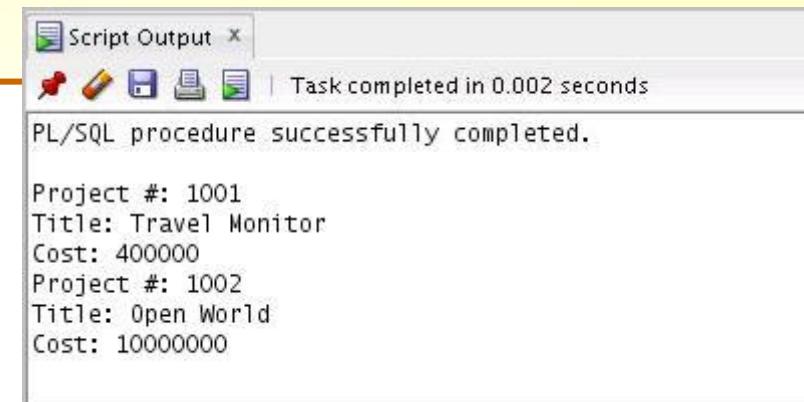
1

```
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

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;
```



Working with Collection Methods

Collection methods:

- Are predefined PL/SQL programs
- Help users to operate on collections

collection_name.method_name [(parameters)]

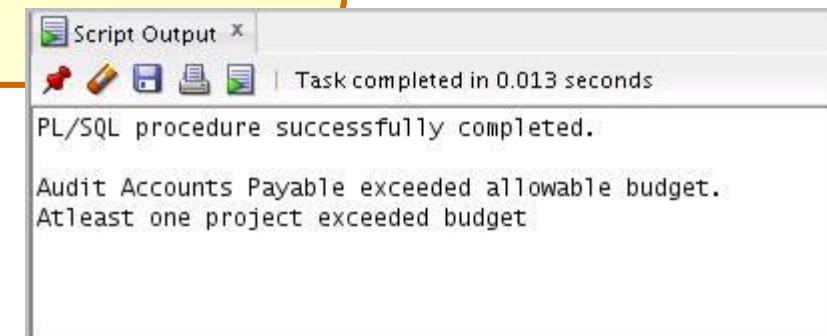
Using Collection Methods

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.');
        RETURN TRUE;
    END IF;
    i := p_project_list.NEXT(i);
END LOOP;
RETURN null;
END check_costs;
```

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;
```



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;
```

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	10000000
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;
```

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	10000000
3	20 Information Technology	2001	DB11gR2	900000
4	20 Information Technology	2002	AQM	80000

Querying a Collection by 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



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
/
```

Lesson Agenda

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



Collection Exceptions

Common exceptions with collections:

- COLLECTION_IS_NULL
- NO_DATA_FOUND
- SUBSCRIPT_BEYOND_COUNT
- SUBSCRIPT_OUTSIDE_LIMIT
- VALUE_ERROR

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
    ...

```

Lesson Agenda

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



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_INTEGER	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

Lesson Agenda

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



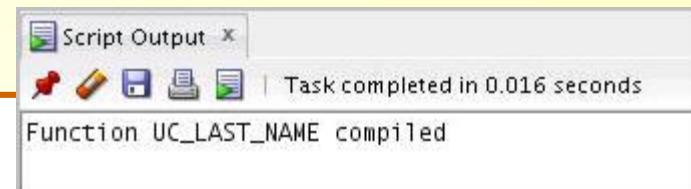
PL/SQL Bind Types

- PL/SQL bind types refer to the data types that can be passed as parameters to PL/SQL program units while invoking them from dynamic SQL or clients such as JDBC or OCI.
- In 19c, you can use the following types as parameters while invoking PL/SQL program units:
 - Boolean
 - Records
 - Collections

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;
/
```



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

Quiz



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

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



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



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

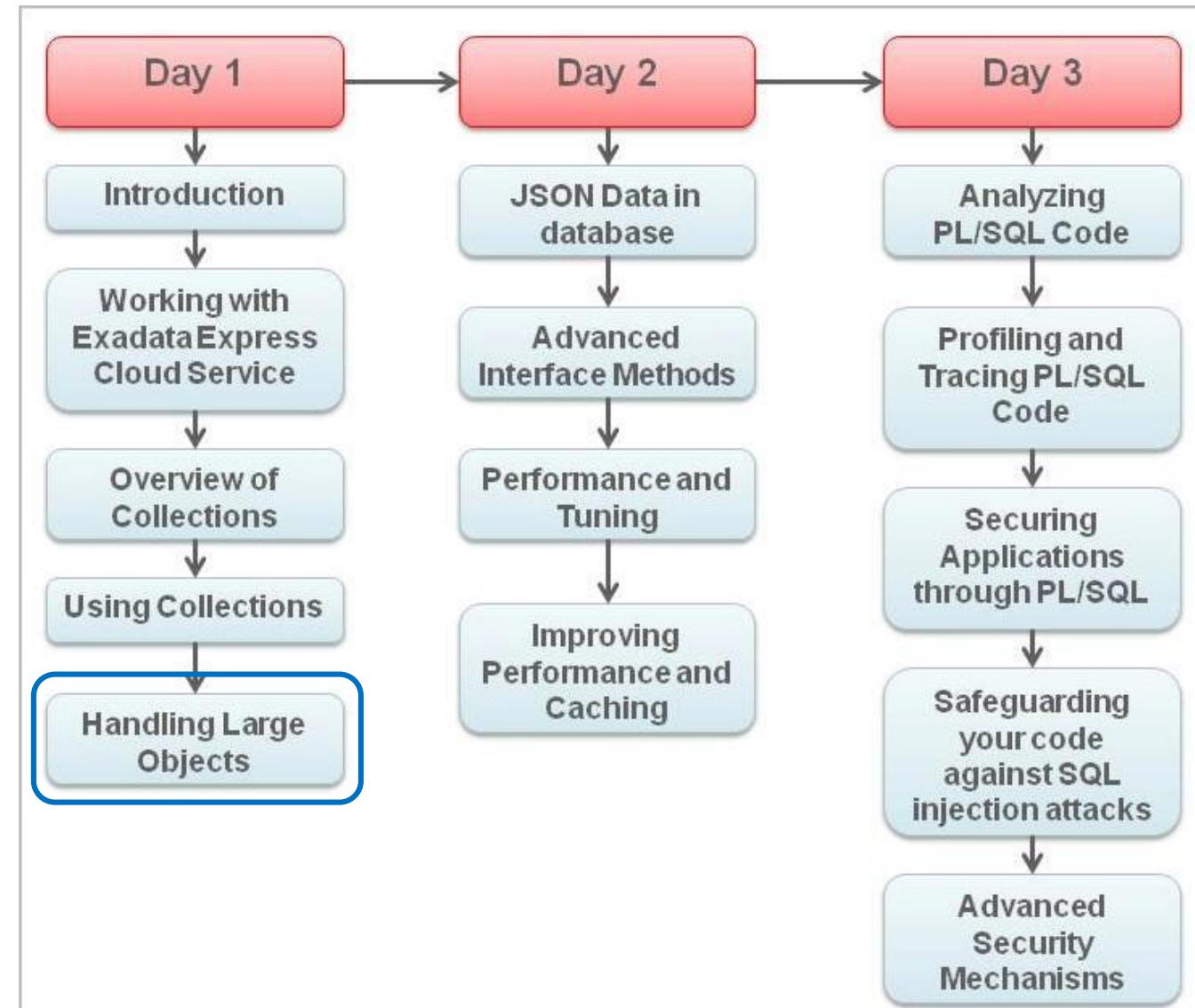


Practice 4: Overview

This practice covers using collections.

Handling Large Objects

Course Agenda



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 SecureFile LOB



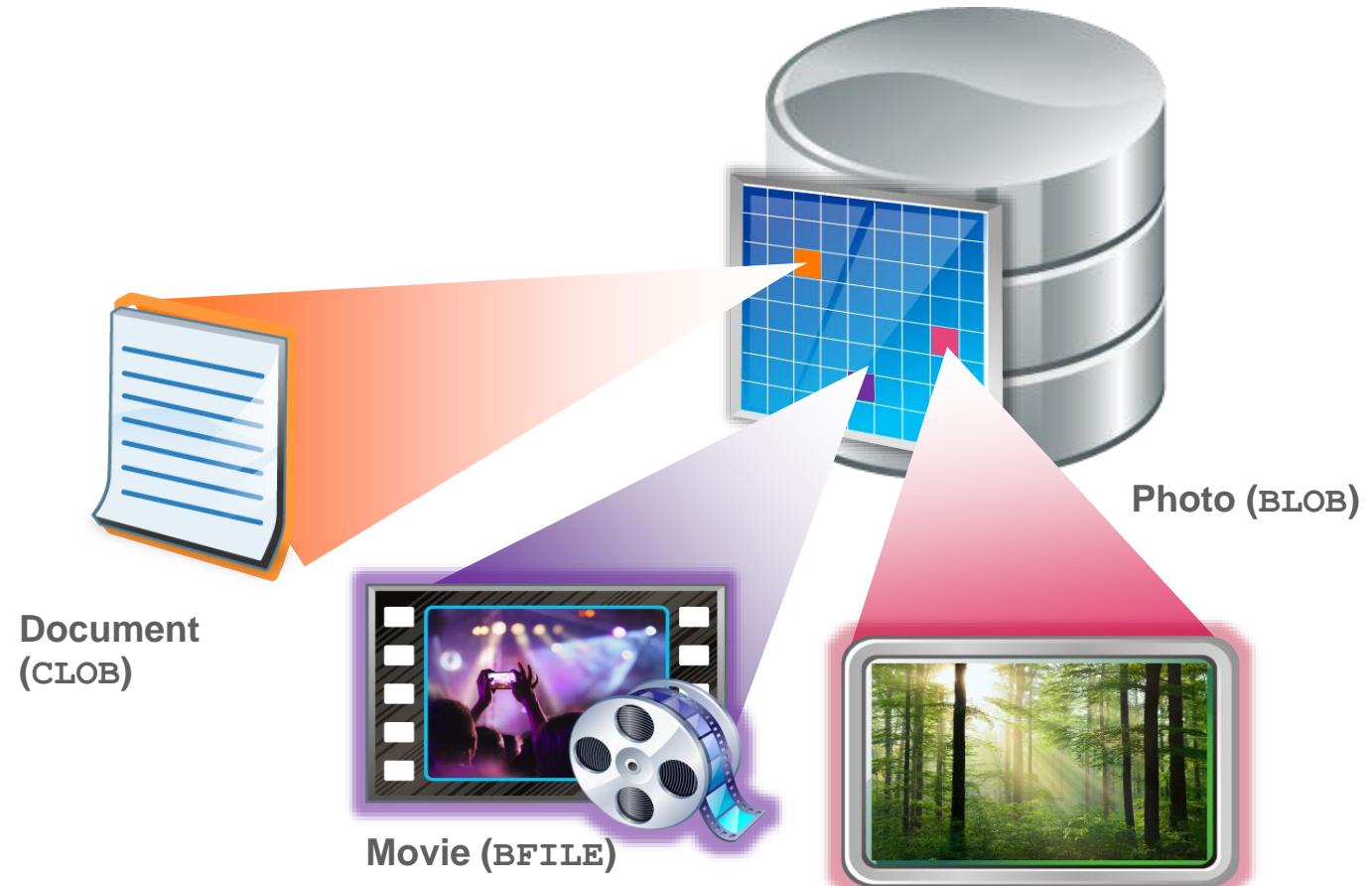
Lesson Agenda

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



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



Types of LOBs

Based on the physical location of the large object, there are two types of LOBs:

- Internal LOBs
- External LOBs



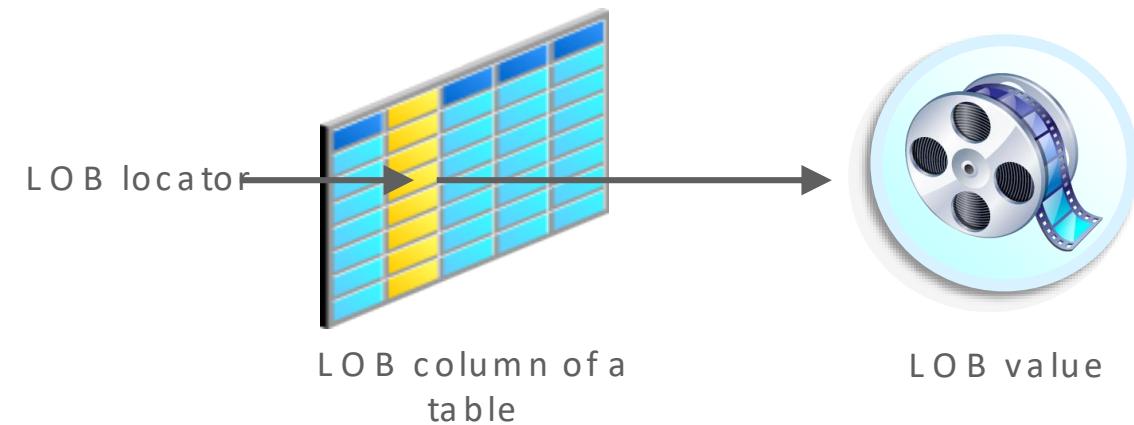
Internal LOBs



External LOBs

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.



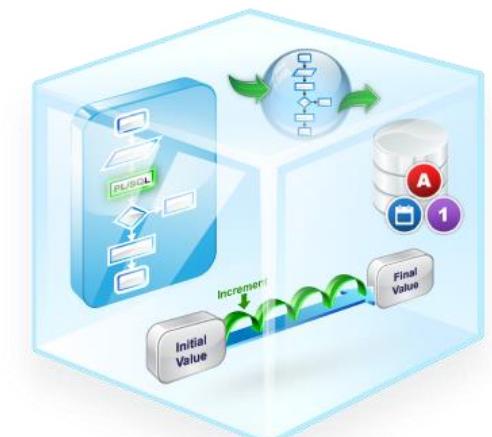
Lesson Agenda

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



DBMS_LOB Package

- DBMS_LOB is an Oracle supplied package.
- It has subprograms, which can read and modify LOBs.
- DBMS_LOB subprograms perform operations based on LOB locators.



Security Model of the DBMS_LOB Package

- The DBMS_LOB package is created as a SYS user.
- You can use the subprograms 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.

What Is 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
- Is created by administrators or users with CREATE DIRECTORY privileges

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 appropriate database users



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



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



Working on BFILEs

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

1. Create a DIRECTORY object as the SYSDBA user.
2. Grant access to the application user to access the 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.

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 appropriate users:

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

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)  
RETURN BFILE;
```

- Example:

- Add a BFILE column to a table:

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

- Update the column by using the BFILENAME function:

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

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;
```

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
- 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;
/
```

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



Working on CLOBs

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

1. Create a tablespace 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.

Initializing the 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;
```

Initializing the LOB Columns Added to a Table

Initialize the column LOB locator value with the DEFAULT option or the DML statements by using:

- The EMPTY_CLOB() function for a CLOB column
- The 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);
```

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;
```

Loading Data to a LOB Column

Create the procedure to read 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_LOBLOADBLOBFROMFILE(p_dest_loc, v_src_loc,
                           v_amount, offset, offset);
  DBMS_LOB.CLOSE(v_src_loc);
END loadLOBFromBFILE_proc;
```

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;
```

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')
```

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



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_LOBLOADCLOBFROMFILE
        (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;
/
```

Updating LOB by Using DBMS_LOB in PL/SQL

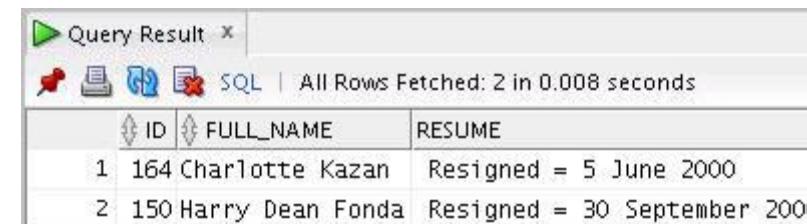
```
DECLARE
    v_loblock CLOB;      -- serves as the LOB locator
    v_text    VARCHAR2(50) := 'Resigned = 5 June 2000';
    v_amount NUMBER ;   -- amount to be written
    v_offset INTEGER;   -- where to start writing
BEGIN
    SELECT resume INTO v_loblock FROM customer_profiles
    WHERE id = 164 FOR UPDATE;
    v_offset := DBMS_LOB.GETLENGTH(v_loblock) + 2;
    v_amount := length(v_text);
    DBMS_LOB.WRITE (v_loblock, v_amount, v_offset, v_text);
    v_text := ' Resigned = 30 September 2000';
    SELECT resume INTO v_loblock FROM customer_profiles
    WHERE id = 150 FOR UPDATE;
    v_amount := length(v_text);
    DBMS_LOB.WRITEAPPEND(v_loblock, v_amount, v_text);
    COMMIT;
END;
```

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:



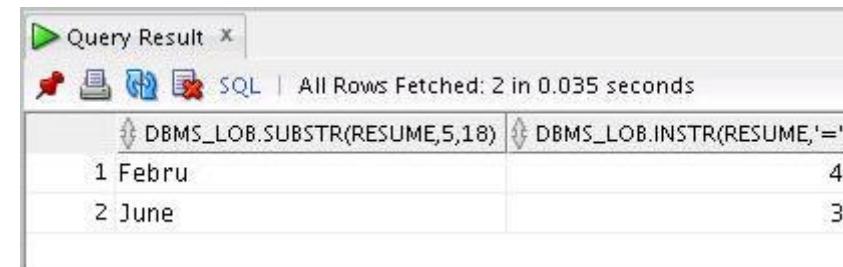
ID	FULL_NAME	RESUME
1	164 Charlotte Kazan	Resigned = 5 June 2000
2	150 Harry Dean Fonda	Resigned = 30 September 2000

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



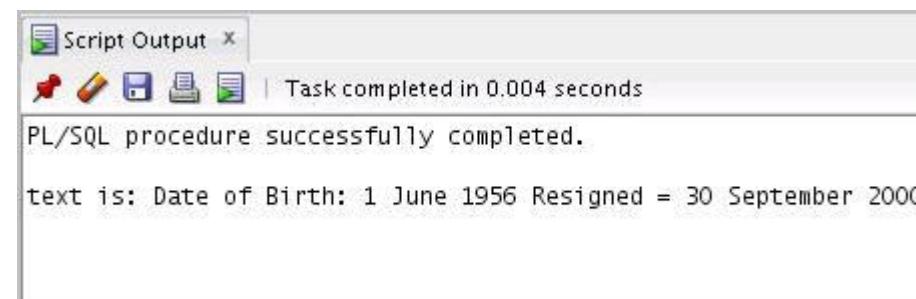
The screenshot shows the Oracle SQL Developer interface with a "Query Result" window. The window title is "Query Result". Below the title, there are icons for Refresh, Print, Copy, Paste, and SQL. The status bar indicates "All Rows Fetched: 2 in 0.035 seconds". The query results are displayed in a table with two columns: "DBMS_LOB.SUBSTR(RESUME,5,18)" and "DBMS_LOB.INSTR(RESUME,'=')". The data rows are:

DBMS_LOB.SUBSTR(RESUME,5,18)	DBMS_LOB.INSTR(RESUME,'=')
1 Febru	41
2 June	37

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;
/
```



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;
```

Quiz



The BFILE data type stores a locator to the physical file.

- a. True
- b. False



Quiz



You use the BFILENAME function to:

- a. Create a BFILE column
- b. Initialize a BFILE column
- c. Update a BFILE column



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. You can 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.



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

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.

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;
/
```

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

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



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.

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

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 roll back 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.



Summary

In this lesson, you should have learned how to:

- Identify four built-in types for large objects: BLOB, CLOB, NCLOB, and BFILE
- 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



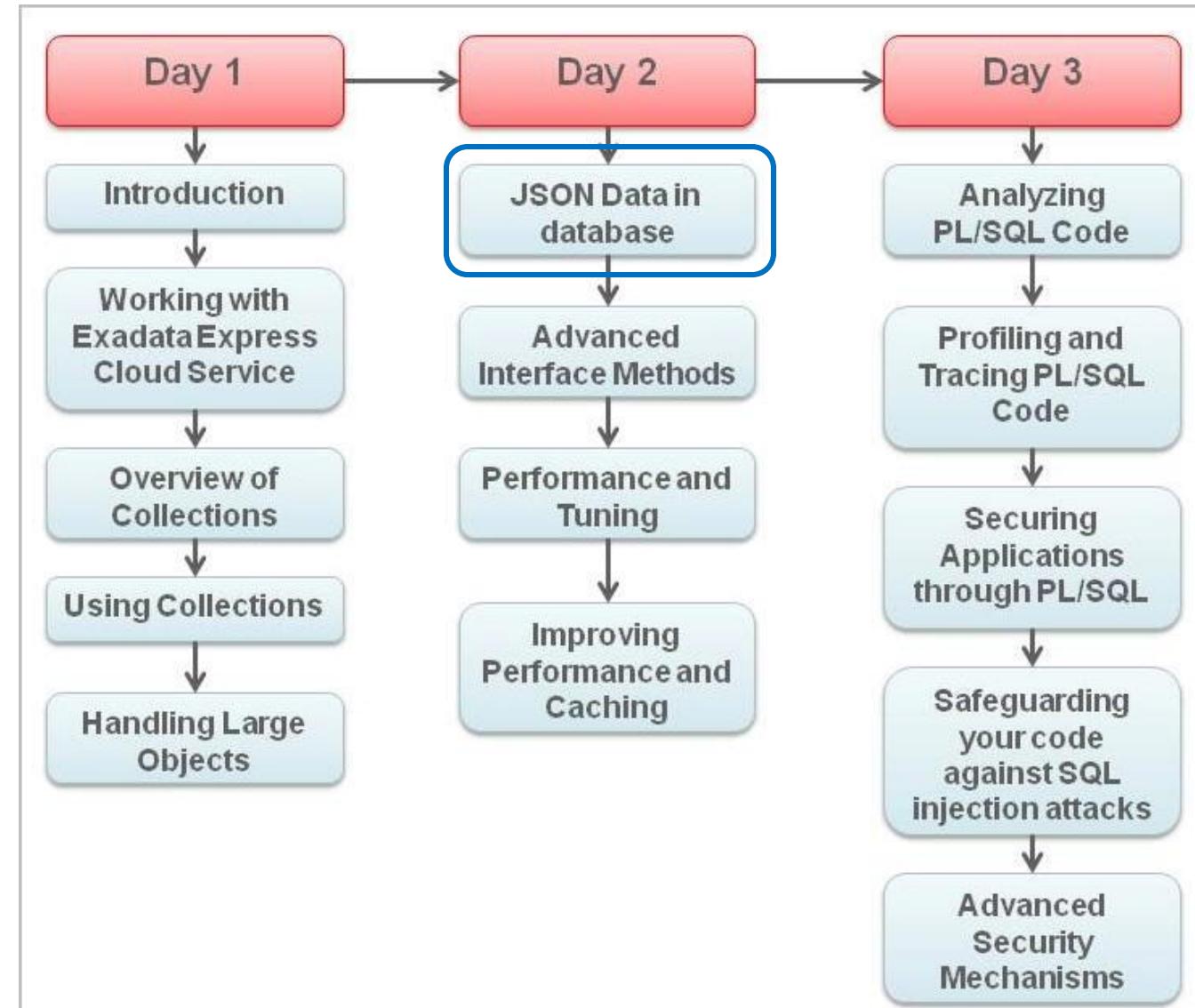
Practice 5: Overview

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

JSON Data in Database

Course Agenda



Objectives

After completing this lesson, you should be able to:

- Describe JSON data
- Store JSON Data in database tables
- Retrieve and perform operations on JSON data
- Manipulate JSON data from a PL/SQL block



Lesson Agenda

- JSON Data
- JSON data in Oracle Database
- SQL/JSON generation functions
- JSON Data in PL/SQL blocks
- PL/SQL object types



What Is JSON?

JavaScript Object Notation (JSON) is:

- Programming language used by web browsers and web servers to communicate
- A lightweight data interchange format
- A readable format for data structuring
- Easy for humans to read and write
- Easy for software to parse and generate



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 braces.
- The property name and property value is separated by a colon.

```
{ property 1: value 1,  
  property 2:value 2,  
  property 3: value 3 ... }
```

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,  
    ...}
```

Why JSON?

- A simple way of data interchange
- Written in text format
- Interoperable
- Inherently compatible with JavaScript
- Easy for machines to parse and generate

Lesson Agenda

- JSON Data
- JSON data in Oracle Database
- SQL/JSON generation functions
- JSON Data in PL/SQL blocks
- PL/SQL object types



JSON Data in Oracle Database: Scenario

- Consider a scenario in the 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.



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.

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.

Creating a Table with JSON Column

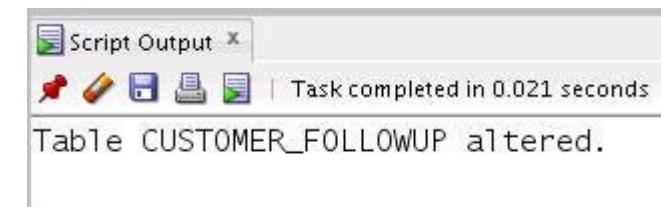
- JSON column can assume any one of the following standard SQL data types in a table:
 - VARCHAR2
 - CLOB
 - BLOB
- Decide on the data type to be used based on the volume of data you would store in the column.
- Storing JSON data using standard SQL data types allows all the features of Oracle Database to be applied to it.

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));
```



JSON or Not?

When we store JSON as VARCHAR2 column, how do we differentiate a JSON and a non-JSON column?

- JSON objects should have the right structure.
- You can use `is_json` SQL/JSON condition to check the structure of JSON data.
- Oracle recommends that you define an `is_json` constraint on every JSON column.

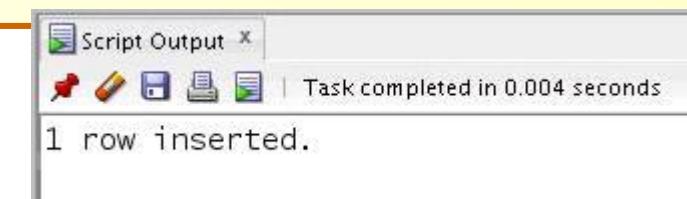
```
ALTER TABLE customer_followup
ADD CONSTRAINT json_check CHECK(order_history IS JSON);
```



Inserting Data into JSON Columns

- You can insert data into JSON column by using the `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;
```



Lesson Agenda

- JSON Data
- JSON data in Oracle Database
- SQL/JSON generation functions
- JSON Data in PL/SQL blocks
- PL/SQL object types



SQL/JSON Generation Functions

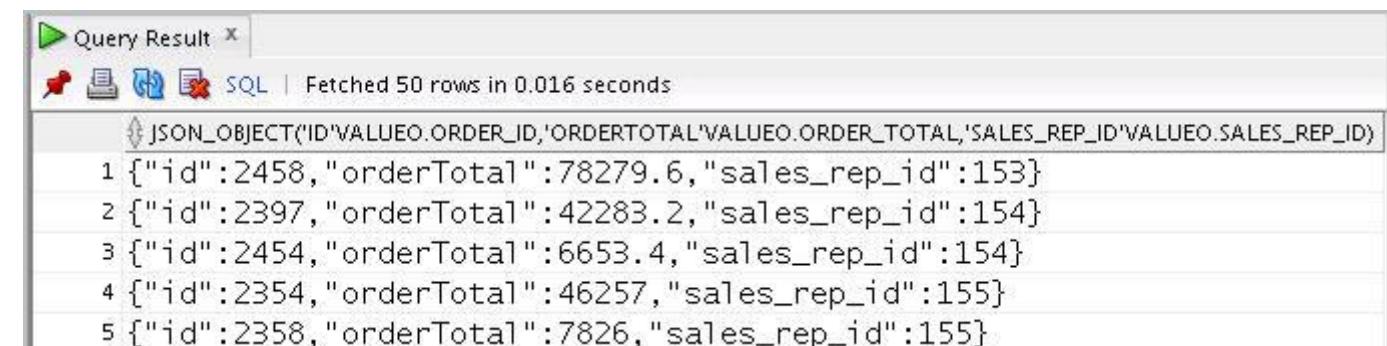
You can use the following SQL/JSON functions to generate JSON data from non-JSON data:

- **JSON_OBJECT**
- **JSON_ARRAY**
- **JSON_OBJECTAGG**
- **JSON_ARRAYAGG**

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 ;
```



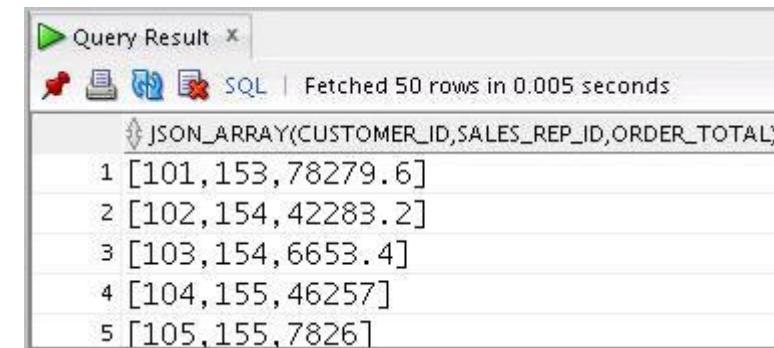
The screenshot shows a 'Query Result' window in Oracle SQL Developer. The title bar says 'Query Result x'. Below it, there are icons for refresh, save, and cancel, followed by 'SQL | Fetched 50 rows in 0.016 seconds'. The main area displays the results of the query:

	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}

JSON_ARRAY Function

- The JSON_ARRAY function generates a JSON array based on the parameters passed.

```
SELECT JSON_ARRAY(customer_id, sales_rep_id, order_total)
FROM orders o ;
```



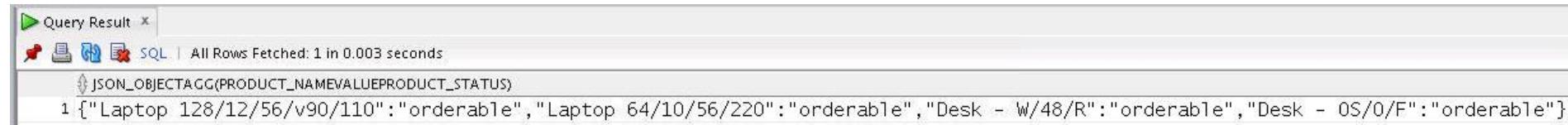
A screenshot of the Oracle SQL Developer interface. The title bar says "Query Result". Below it, there are icons for Refresh, Save, Print, and Stop, followed by "SQL | Fetched 50 rows in 0.005 seconds". The main area shows a table with one column labeled "JSON_ARRAY(CUSTOMER_ID,SALES REP_ID,ORDER_TOTAL)". The data consists of five rows of JSON arrays:

JSON_ARRAY(CUSTOMER_ID,SALES REP_ID,ORDER_TOTAL)
1 [101,153,78279.6]
2 [102,154,42283.2]
3 [103,154,6653.4]
4 [104,155,46257]
5 [105,155,7826]

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 'Query Result' tab in Oracle SQL Developer. The query executed is:

```
SELECT JSON_OBJECTAGG(product_name VALUE product_status)
FROM product_information where min_price>2000 ;
```

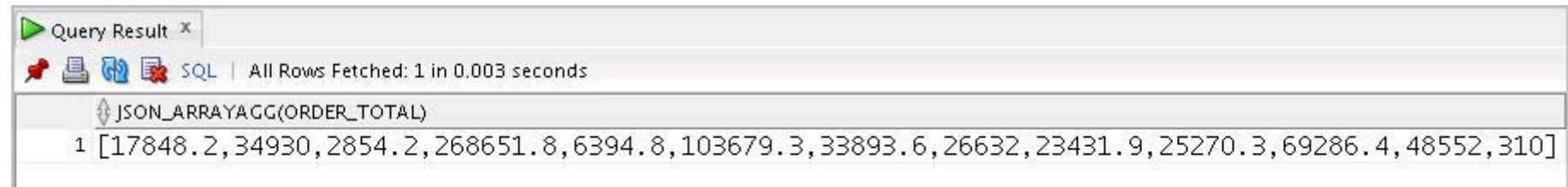
The result is a single row of JSON data:

```
1 {"Laptop 128/12/56/v90/110": "orderable", "Laptop 64/10/56/220": "orderable", "Desk - W/48/R": "orderable", "Desk - OS/O/F": "orderable"}
```

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 ;
```



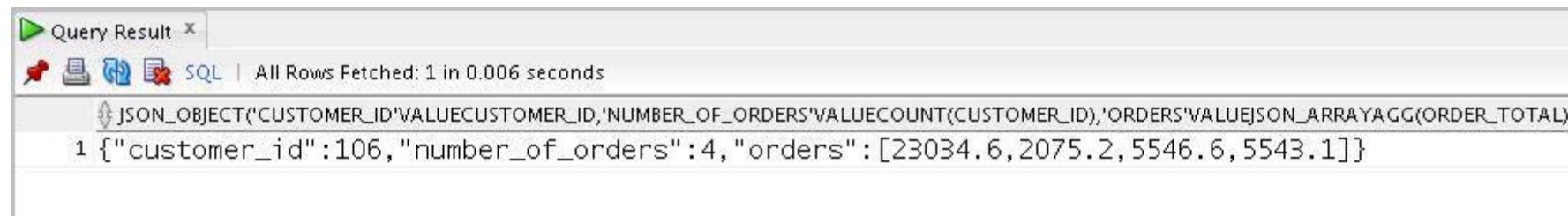
The screenshot shows a 'Query Result' window from Oracle SQL Developer. The window title is 'Query Result'. It contains a toolbar with icons for Run, Stop, Refresh, and SQL. Below the toolbar, it says 'All Rows Fetched: 1 in 0.003 seconds'. The main area displays the result of the query: a single row with one column labeled 'JSON_ARRAYAGG(ORDER_TOTAL)'. The value in the cell is '[17848.2,34930,2854.2,268651.8,6394.8,103679.3,33893.6,26632,23431.9,25270.3,69286.4,48552,310]'. The entire screenshot is enclosed in a red rounded rectangle.

JSON_ARRAYAGG(ORDER_TOTAL)
[17848.2,34930,2854.2,268651.8,6394.8,103679.3,33893.6,26632,23431.9,25270.3,69286.4,48552,310]

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 a 'Query Result' window from Oracle SQL Developer. The window title is 'Query Result'. The toolbar includes icons for play, refresh, and stop, followed by 'SQL | All Rows Fetched: 1 in 0.006 seconds'. The main area displays the query and its output:

```
JSON_OBJECT('CUSTOMER_ID' VALUE CUSTOMER_ID, 'NUMBER_OF_ORDERS' VALUE COUNT(CUSTOMER_ID), 'ORDERS' VALUE JSON_ARRAYAGG(ORDER_TOTAL))
1 {"customer_id":106, "number_of_orders":4, "orders": [23034.6, 2075.2, 5546.6, 5543.1]}
```

Lesson Agenda

- JSON Data
- JSON data in Oracle Database
- SQL/JSON generation functions
- SQL data from JSON objects
- PL/SQL object types



Retrieving SQL Data from JSON Object

- Simple dot notation access
- SQL/JSON path expression
- SQL/JSON query functions

Accessing JSON Data

- You can access certain values in the JSON object by 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;
```



A screenshot of the Oracle SQL Developer interface showing a query result window titled "Query Result". The window displays the output of the SQL query provided above. The results are shown in a table with one row and two columns. The first column is labeled "ID" and contains the value "1 2373". The window also includes standard toolbar icons and status information at the top right indicating "All Rows Fetched: 1 in 0.006 seconds".

ID
1 2373

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.

JSON_VALUE Function

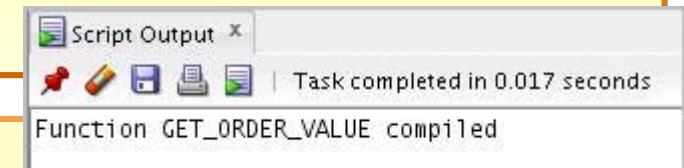
- The JSON_VALUE function is used to retrieve data from an object in a JSON column.
- It returns a scalar SQL value.
- The default return value is **VARCHAR2**.
- You can change it using a **RETURNING** clause.
- These functions are useful when you work on JSON data in PL/SQL blocks.

```
SELECT JSON_VALUE(order_history, '$.orderTotal')
FROM customer_followup
WHERE customer_id =120;
```

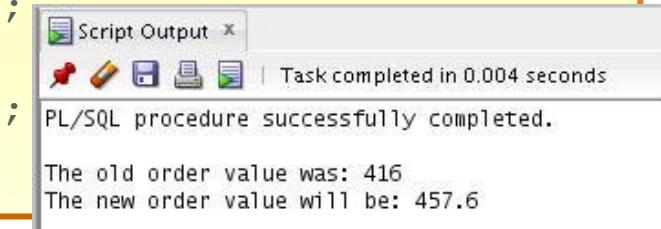


Using SQL/JSON Functions

```
CREATE OR REPLACE FUNCTION get_order_value(ord_his VARCHAR2)
RETURN NUMBER AS
BEGIN
    RETURN JSON_VALUE(ord_his,'$.orderTotal' RETURNING NUMBER);
END;
```



```
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;
```



Lesson Agenda

- JSON Data
- JSON data in Oracle Database
- SQL/JSON generation functions
- JSON Data in PL/SQL blocks
- PL/SQL object types



PL/SQL Objects for JSON

- PL/SQL object types:
 - Allow fine-grained programmatic construction and manipulation of in-memory JSON data
 - Are transient
- To persist the data in PL/SQL objects, you must serialize them into VARCHAR2 or LOB and store it in the database table.

JSON Object Types in PL/SQL

The 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`

JSON Object Methods

The following methods enable you to perform operations on JSON data:

- Parsing and serializing
- Getter and setter
- Introspection

Getter and Setter Methods

- The 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

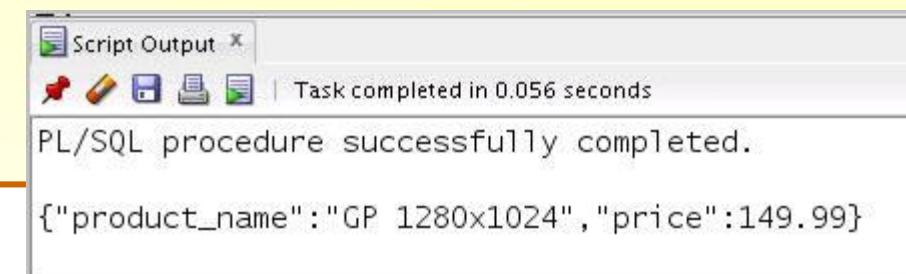
```
get()  
get_string()  
get_clob()  
get_blob()
```

Setter methods

```
put()  
put_null()  
append() (for  
JSON_ARRAY_T only)
```

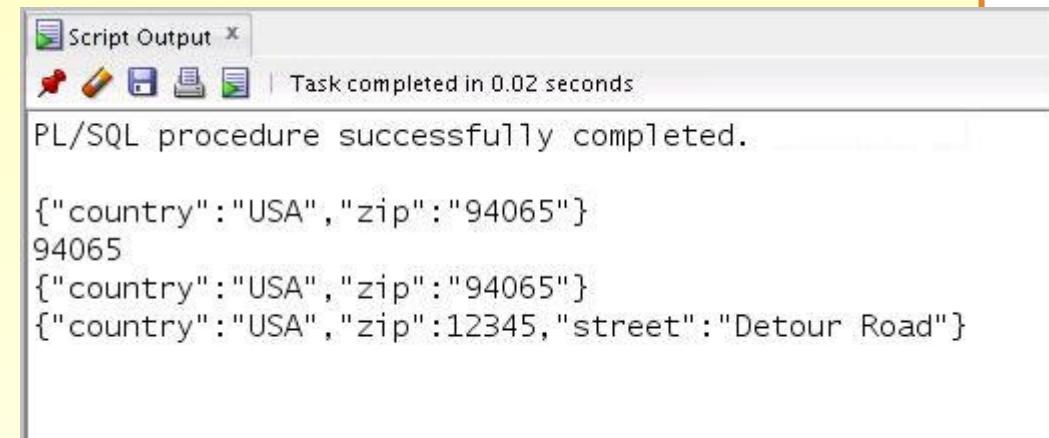
JSON Methods in PL/SQL: Example

```
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;
```



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;
/
```



```
Script Output x
Task completed in 0.02 seconds
PL/SQL procedure successfully completed.

{"country":"USA","zip":"94065"}
94065
{"country":"USA","zip":"94065"}
{"country":"USA","zip":12345,"street":"Detour Road"}
```

Summary

In this lesson, you should have learned how to:

- Create a table with a JSON column in it
- Insert data into a JSON column
- Generate JSON data from database tables
- Use PL/SQL JSON object types
- Use JSON methods in PL/SQL blocks

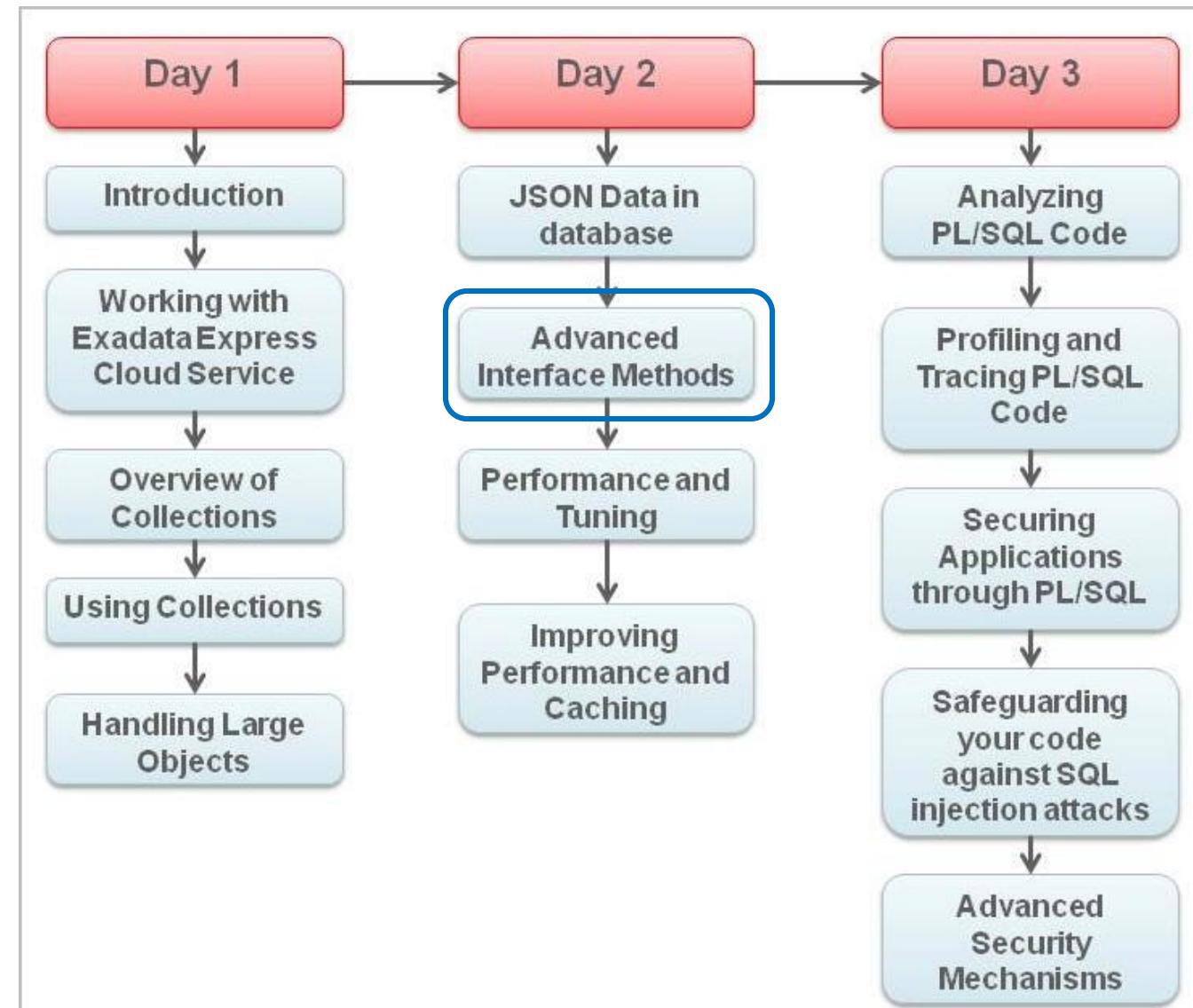


Practice 6: Overview

- 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.

Advanced Interface Methods

Course Agenda



Objectives

After completing this lesson, you should be able to:

- Explain the architecture of external procedures
- Execute external C programs from PL/SQL
- Execute Java programs from PL/SQL



Lesson Agenda

- Understanding External Procedures
- Defining an external C procedure
- Executing Java programs from PL/SQL



PL/SQL External Procedures

- An external procedure is a procedure written in a different programming language.
- You register the procedure with the base language and then call it for special-purpose processing.
- External procedures:
 - Interface the database server with external systems
 - Extend the functionality of the database server

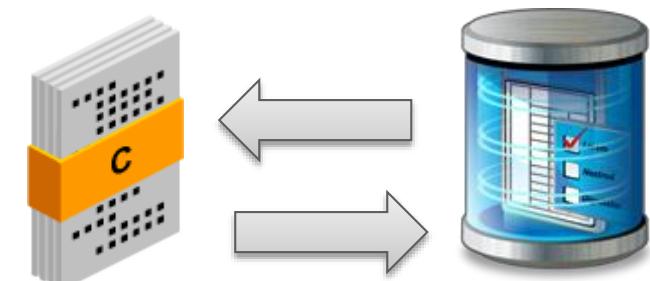
Oracle Database with Different Languages

Oracle Database allows programmers to work with different programming languages:

- C, C++
- Java
- COBOL
- Visual Basic
- .NET

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 C programs. After the execution of the C programs, depending on the result of the evaluations, data is inserted into appropriate Oracle database tables.

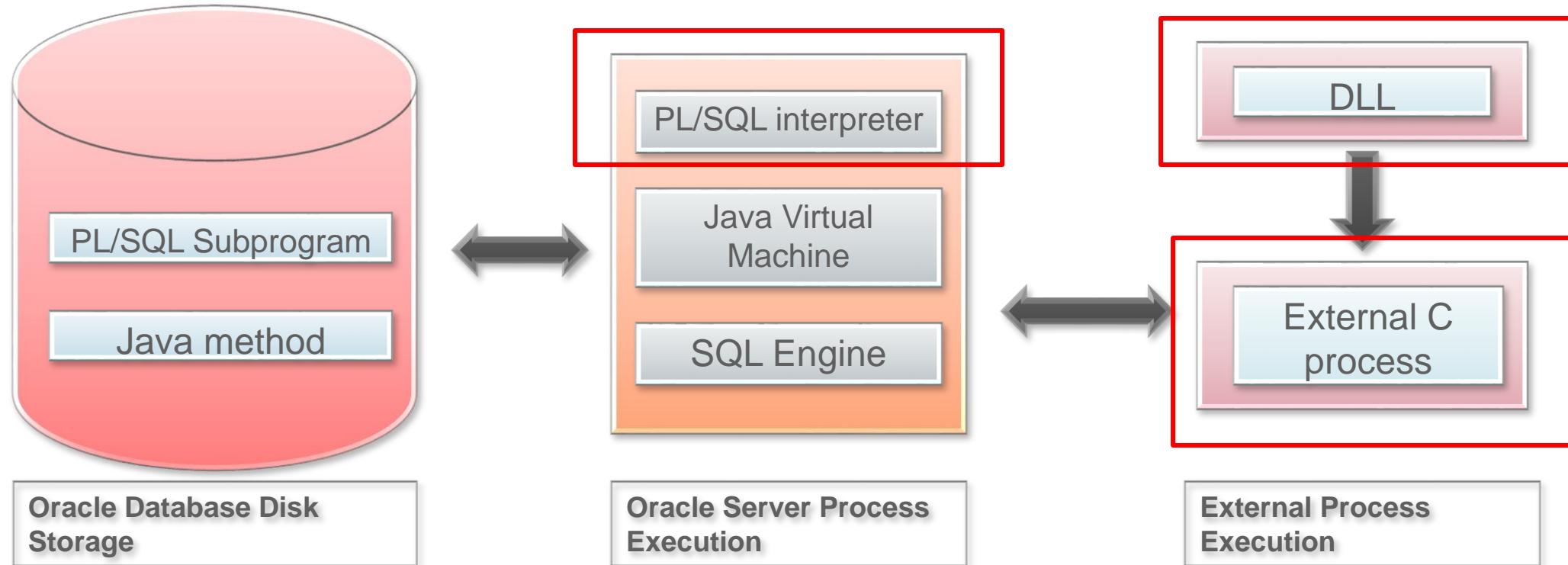


Lesson Agenda

- Understanding External Procedures
- Defining an external C procedure
- Executing Java programs from PL/SQL

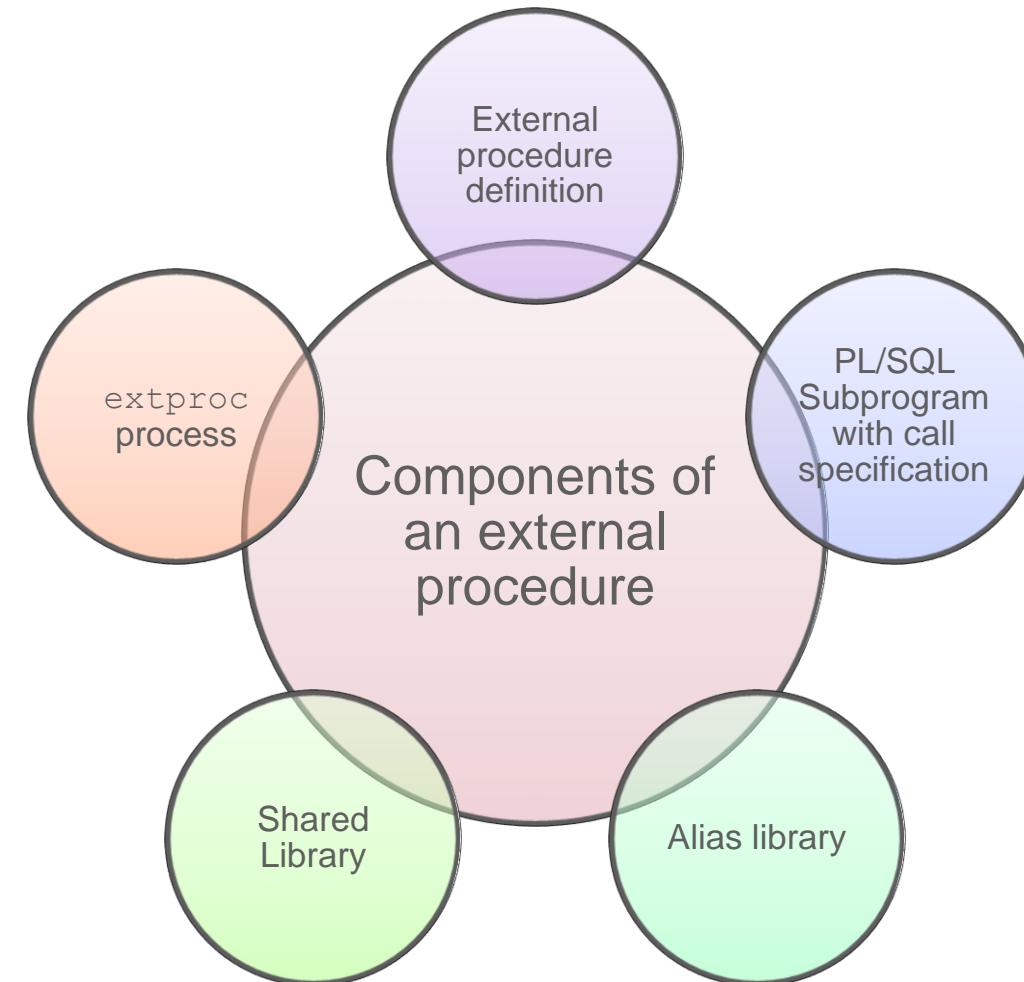


External Procedure Execution Architecture



Components for External C Procedure Execution

Various components work together to interface a C program with Oracle Database.



Defining an External C Procedure

1. Define the procedure in an external language (C, Java, and so on).
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.

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.

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;
```

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.

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]);
```

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]
```

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);
```

Executing an External C 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.

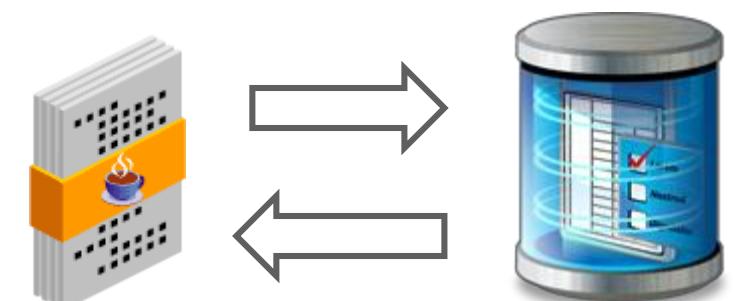
Lesson Agenda

- Understanding External Procedures
- Defining an external C procedure
- Executing Java programs from PL/SQL

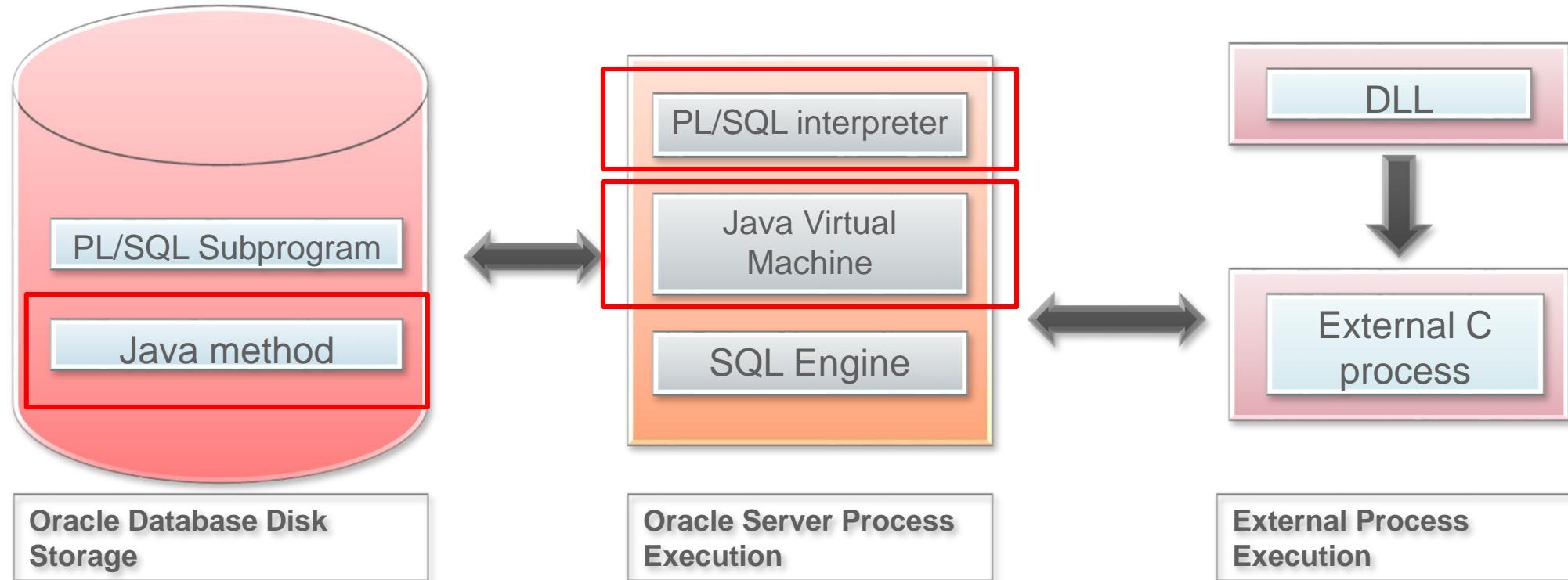


Executing Java Programs from PL/SQL

- 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 in the database.



External Procedure Execution Architecture



Development Steps for Java Class Methods

1. Upload the Java file by 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.

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
```

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.

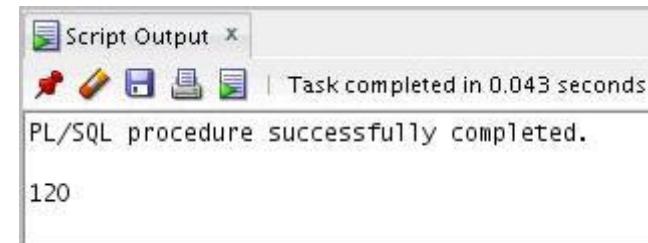
Example:

```
CREATE OR REPLACE FUNCTION plstojavafac_fun
  (N NUMBER)
RETURN NUMBER
AS
  LANGUAGE JAVA
  NAME 'Factorial.calcFactorial
        (int) return int';
```

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;
```



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)';
```

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.



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.



Quiz



Select the correct order of 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



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



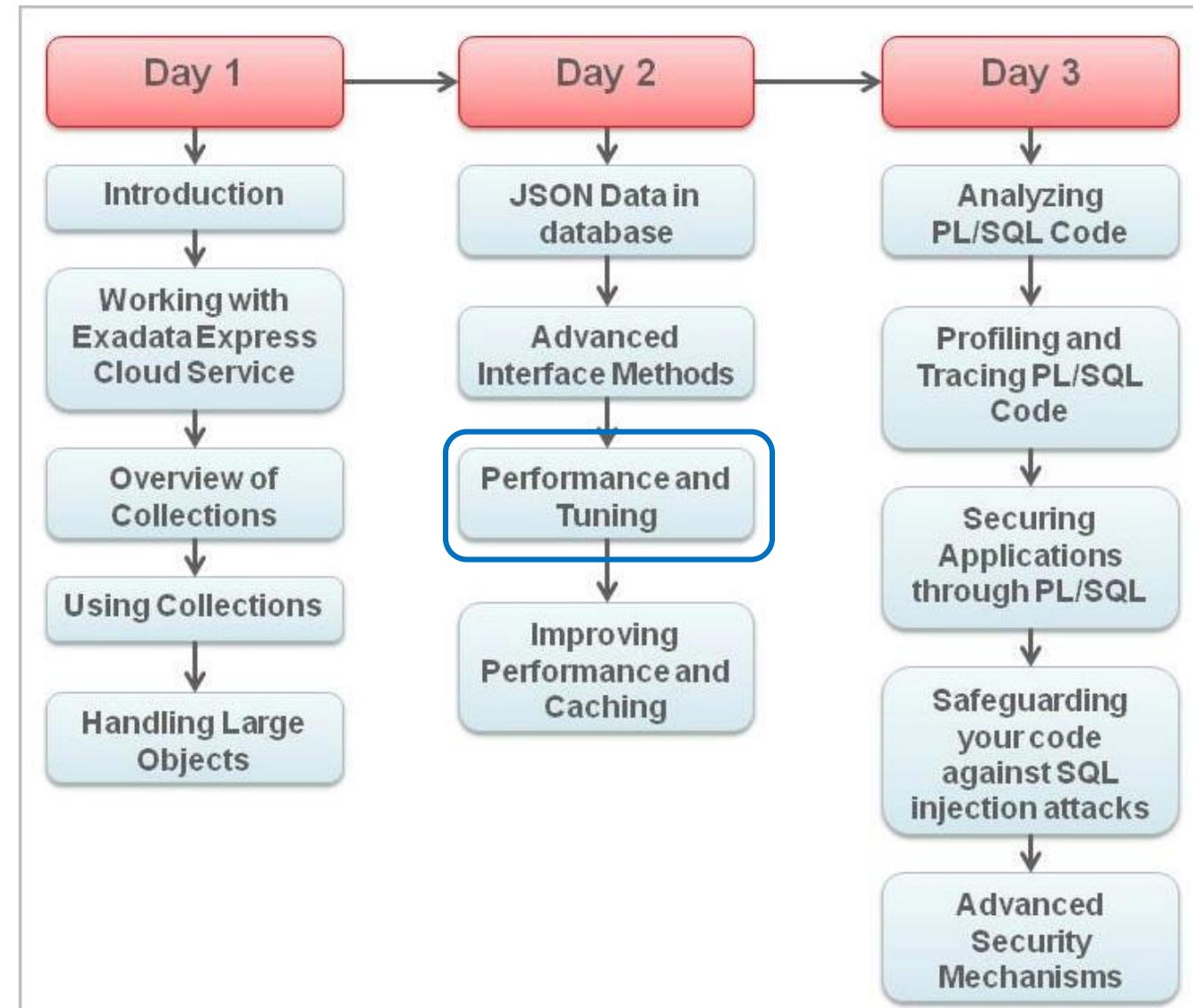
Practice 7: Overview

This practice covers writing programs to interact with:

- C routines
- Java code

Performance and Tuning

Course Agenda



Objectives

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

- Configure your compiler to the appropriate compilation mode
- Enable intra unit inlining to improve performance
- Tune PL/SQL code to improve performance



Lesson Agenda

- Configuring the compiler
- Enabling intraunit inlining
- Tuning PL/SQL code



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 tablespace

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

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 }
```

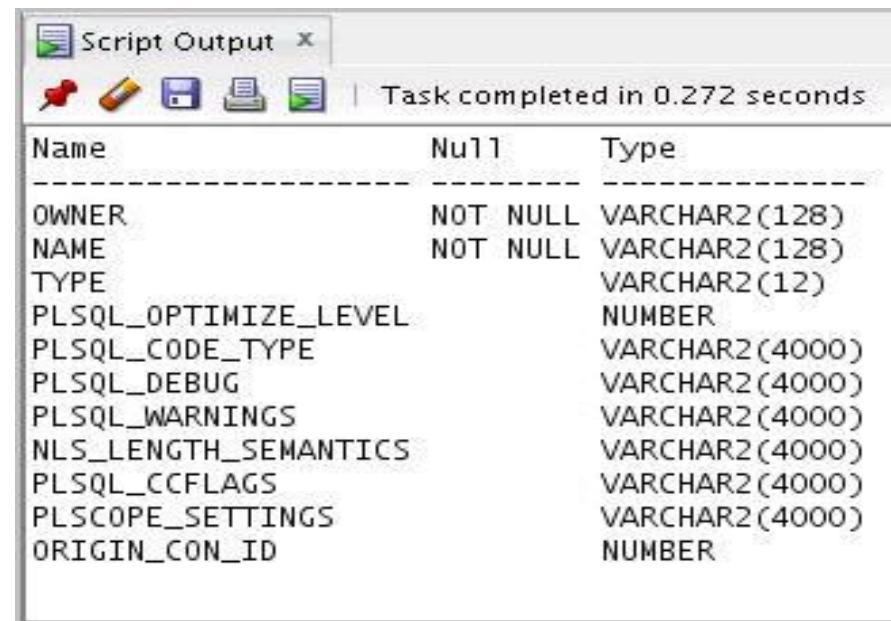
- In general, for faster performance, use the following setting:

```
PLSQL_CODE_TYPE = NATIVE  
PLSQL_OPTIMIZE_LEVEL = 2
```

Viewing the Compilation Settings

DESCRIBE ALL_PLSQL_OBJECT_SETTINGS

→ Displays the settings for a PL/SQL object



The screenshot shows the 'Script Output' window from Oracle SQL Developer. The title bar says 'Script Output' and 'Task completed in 0.272 seconds'. Below the title bar are several icons: a red script, a yellow pencil, a blue folder, a purple file, and a green refresh. The main area contains a table with the following data:

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

Viewing the Compilation Settings

```
SELECT name, plsql_code_type, plsql_optimize_level  
FROM user_plsql_object_settings;
```

→ To view the compilation settings

Query Result

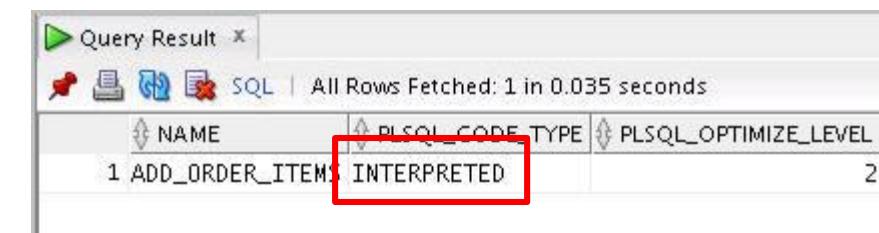
SQL | All Rows Fetched: 63 in 0.073 seconds

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

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 set up a database for native compilation:
 - You require DBA privileges
 - You have to set the `PLSQL_CODE_TYPE` compilation parameter to `NATIVE`

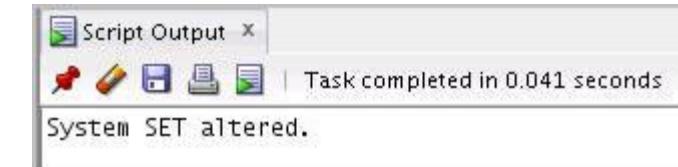
```
SELECT name, plsql_code_type, plsql_optimize_level
FROM   user_plsql_object_settings
WHERE  name = 'ADD_ORDER_ITEMS';
```



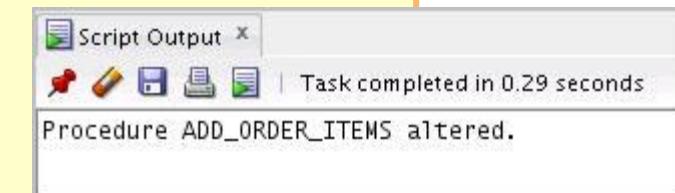
NAME	PLSQL_CODE_TYPE	PLSQL_OPTIMIZE_LEVEL
1 ADD_ORDER_ITEMS	INTERPRETED	2

Modifying Compilation Mode of a Program Unit

```
ALTER SYSTEM SET PLSQL_CODE_TYPE = NATIVE;
```



```
ALTER PROCEDURE add_order_items COMPILE;
```



```
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	NATIVE	2

Lesson Agenda

- Configuring the compiler
- Enabling Subprogram inlining
- Tuning PL/SQL code



PL/SQL Optimizer

- PL/SQL optimizer:
 - Rearranges the code for better performance
 - Is enabled by default
- The `PLSQL_OPTIMIZE_LEVEL` compilation parameter determines the extent of optimization.
- Subprogram inlining is one of the optimization techniques implemented by the optimizer.

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).



Using Inlining

- Implement inlining by using two methods:
 - Oracle parameter `PLSQL_OPTIMIZE_LEVEL`
 - `PRAGMA INLINE`
- It is recommended that you inline:
 - Small programs
 - Programs that are frequently executed

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 the touch function.

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;
...
```

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;
```

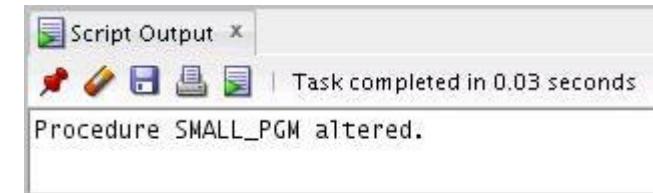
```
a := b;  
a := a*b;  
FOR i IN 1..10 LOOP  
...  
END LOOP;
```

```
a := b*b;  
FOR i IN 1..10 LOOP  
...  
END LOOP;
```

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;
```

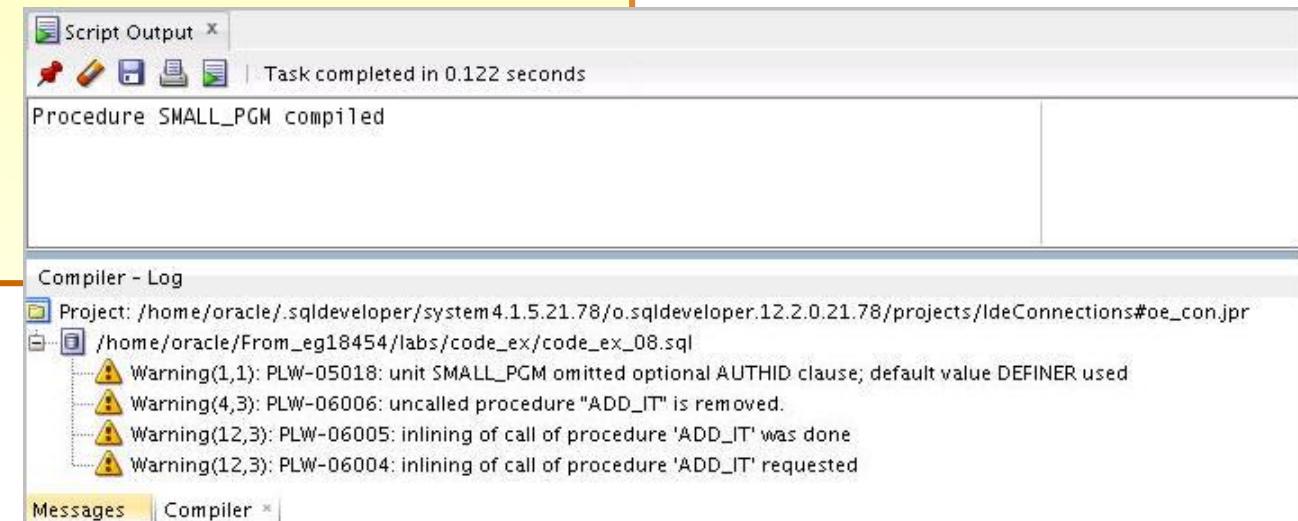


- Setting it to 2 means no automatic inlining is attempted.
- Setting it to 3 means automatic inlining is attempted, but no pragmas are necessary.
- Use PRAGMA INLINE, with the PL/SQL Subprogram.

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;
```



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.

Lesson Agenda

- Configuring the compiler
- Enabling intraunit inlining
- Tuning PL/SQL code



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.



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

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.



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.

PLS_INTEGER Data Type for Integers

Use PLS_INTEGER when dealing with integer data.

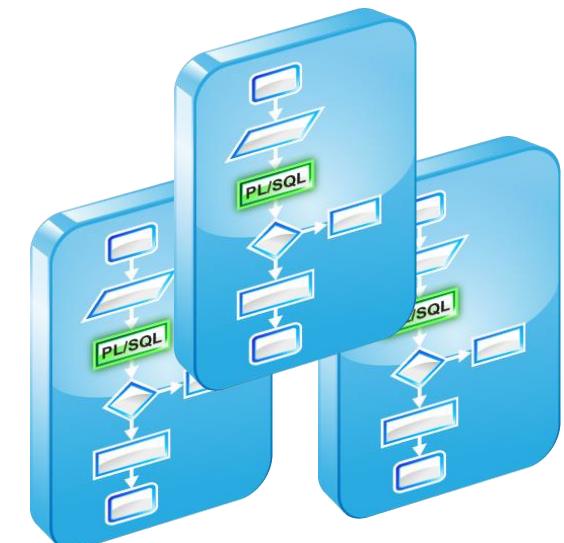
- It is an efficient data type for integer variables.
- It requires less storage than INTEGER or NUMBER.
- Its operations use machine arithmetic, which is faster than library arithmetic.

Using the SIMPLE_INTEGER Data Type

- Definition:
 - Is a predefined subtype
 - Has the range –2147483648 .. 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

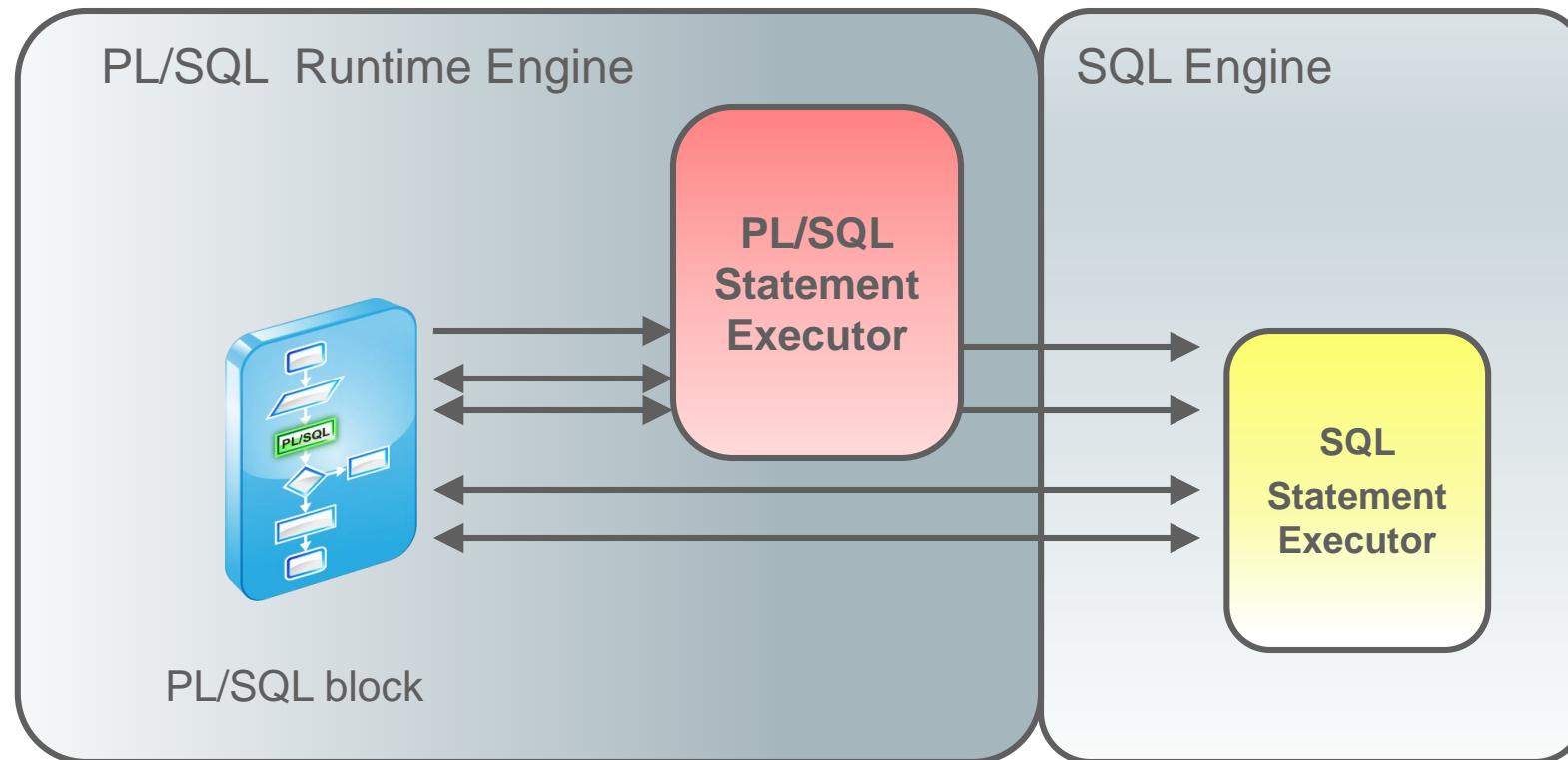
Modularizing Your Code

- 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.



Bulk Binding

Use bulk binds to reduce context switches between the PL/SQL engine and the SQL engine.



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(...);
...
```

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;
```

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;
```

Exception While Bulk Collecting

- You can use the `SAVE EXCEPTIONS` keyword in your `FORALL` statements:

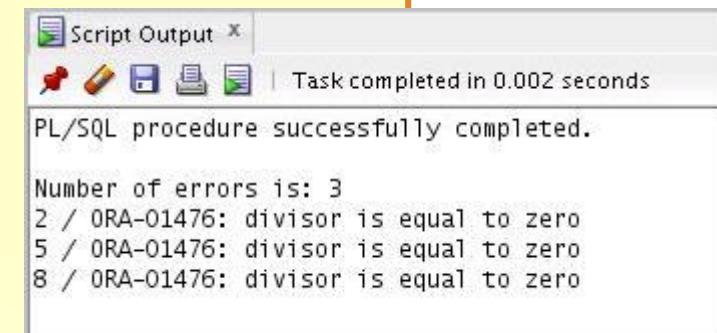
```
FORALL index IN lower_bound..upper_bound  
  SAVE EXCEPTIONS  
  {insert_stmt | update_stmt | delete_stmt}
```

- 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

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;
```



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 in the evaluation order.

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;
```

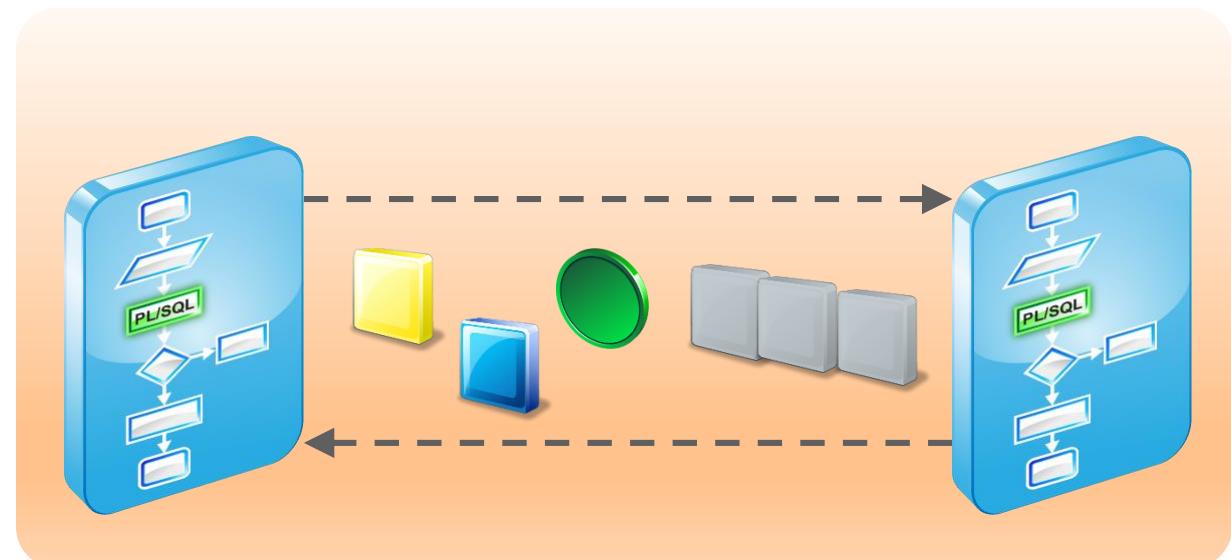


```
IF v_acct_mgr = 145
THEN
    process_acct_145;
ELSIF v_acct_mgr = 147
THEN
    process_acct_147;
ELSIF v_acct_mgr = 148
THEN
    process_acct_148;
ELSIF v_acct_mgr = 149
THEN
    process_acct_149;
END IF;
```



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.



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);
```

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;
```

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.



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



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.



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



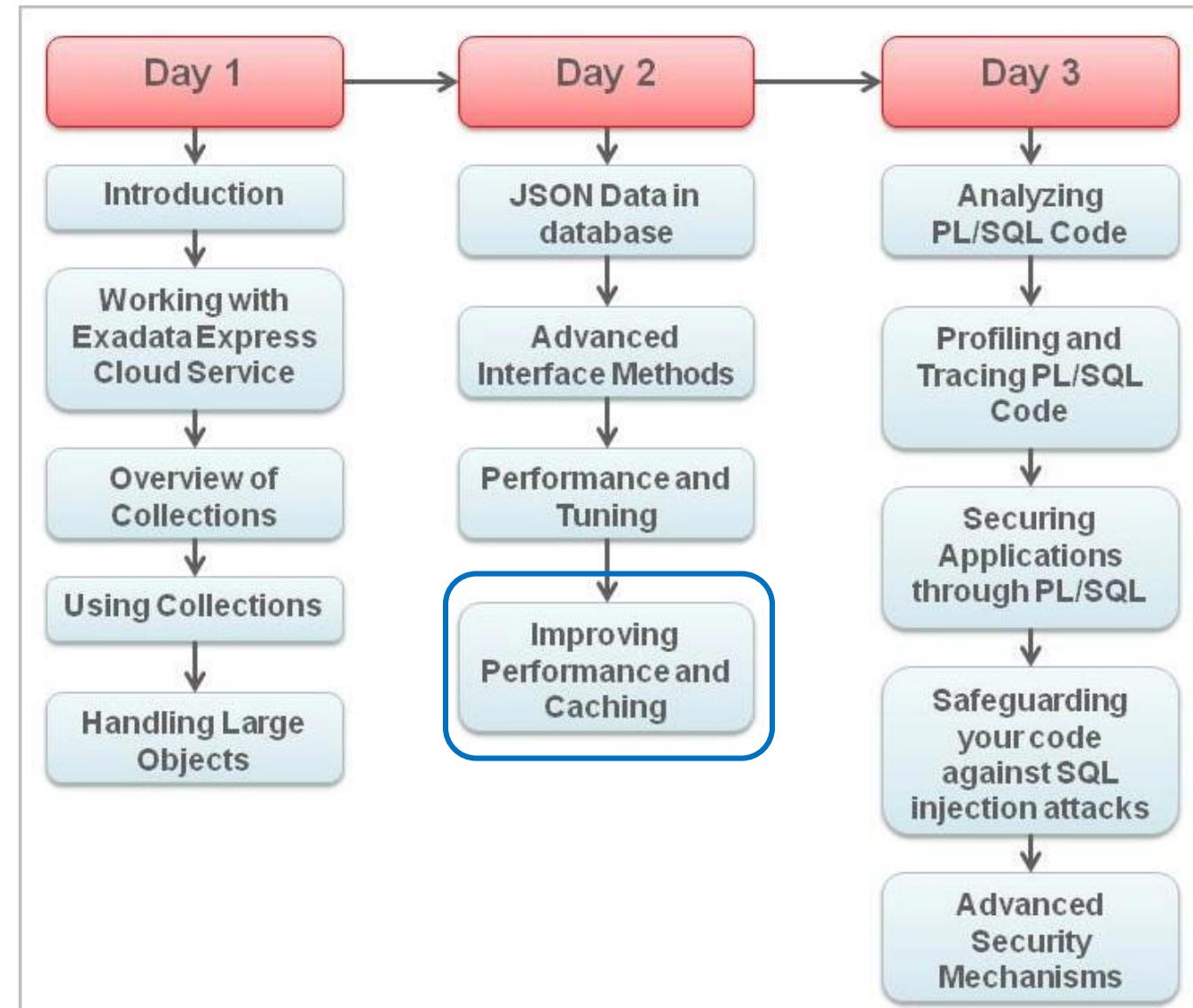
Practice 8: Overview

This practice covers the following topics:

- Tuning PL/SQL code to improve performance
- Coding with bulk binds to improve performance

Improving Performance with Caching

Course Agenda



Objectives

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

- Describe 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

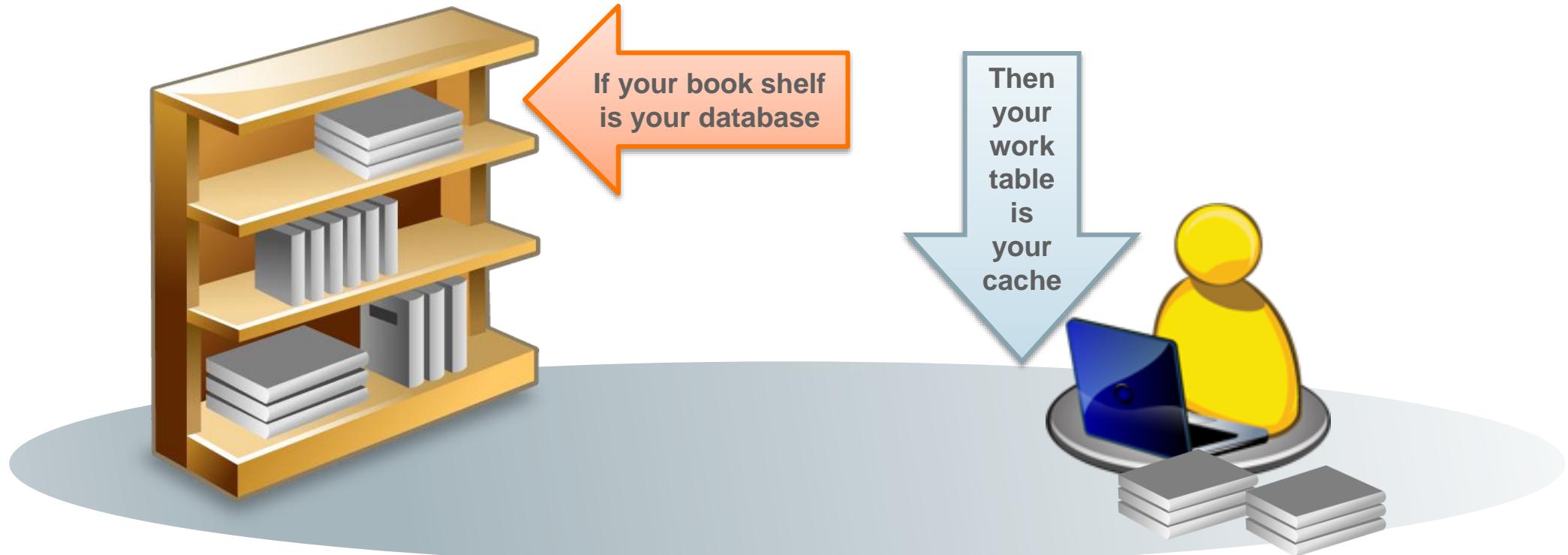


Lesson Agenda

- Caching in the Database Instance
- Result Caching
- SQL Result Caching
- PL/SQL Result Caching
- Oracle Database In-Memory



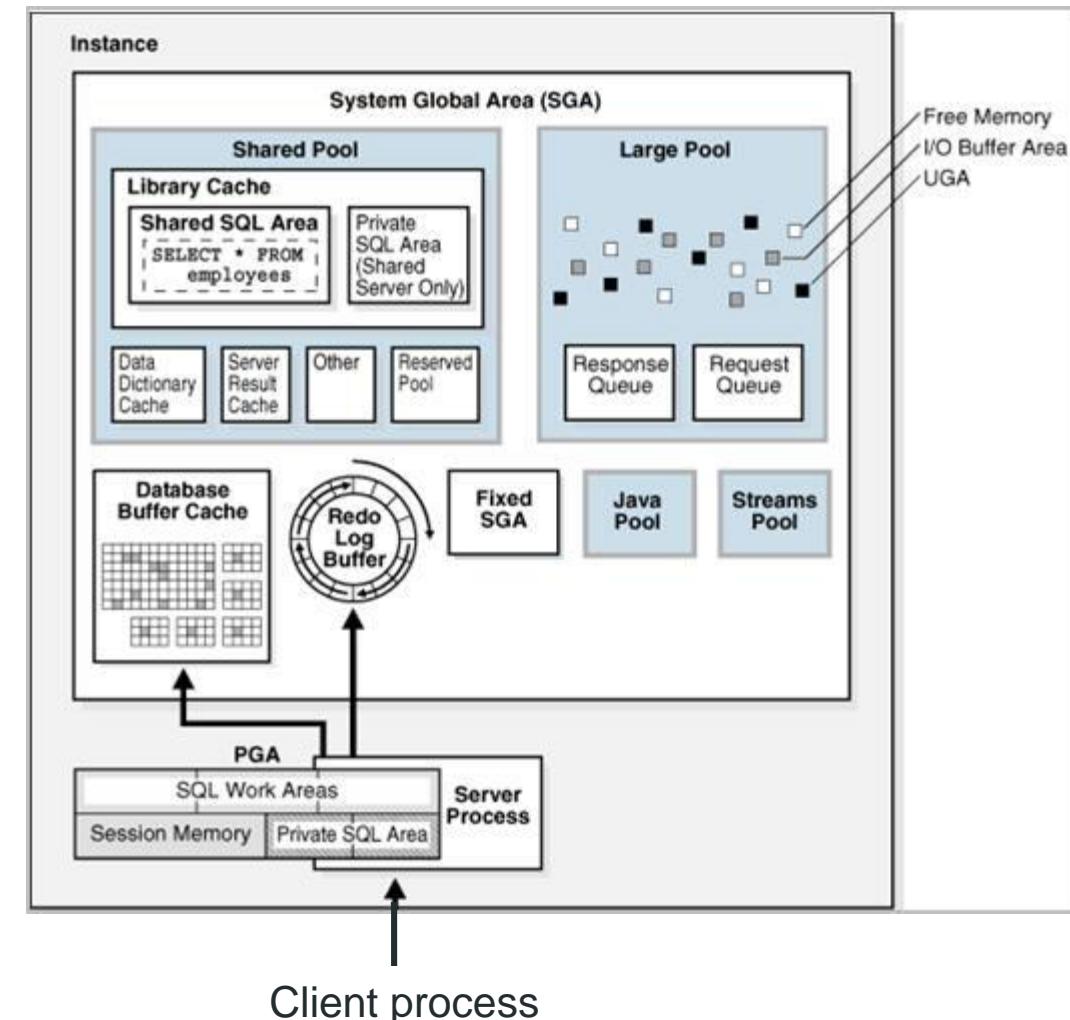
What Is Caching?



Memory Architecture

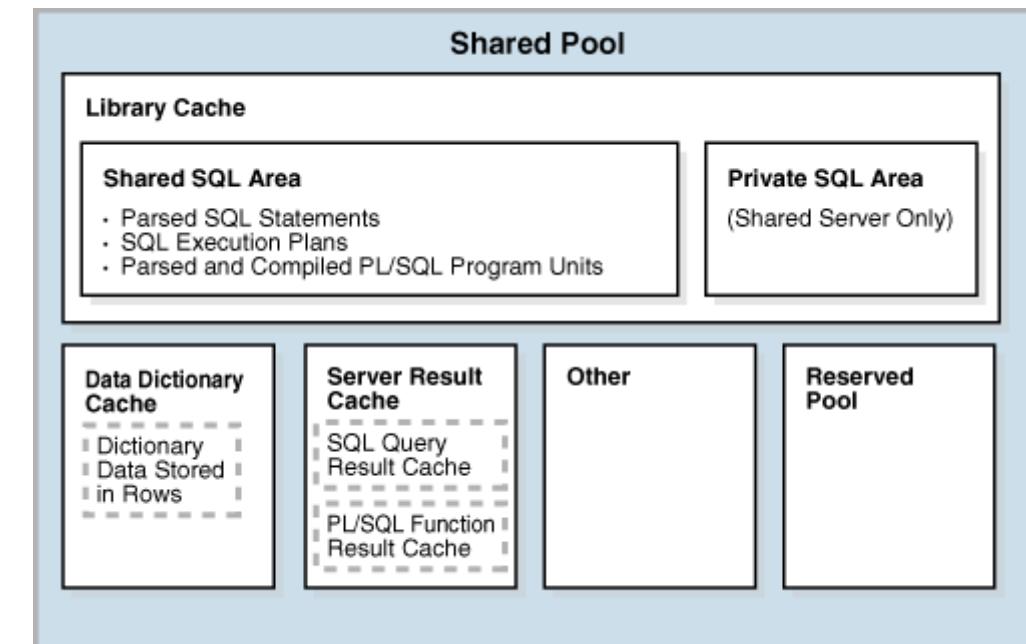
A database instance has the following memory structures:

- System Global Area (SGA)
- Program Global Area (PGA)
- User Global Area (UGA)
- Software code areas



Caching in the Database Instance

- Database Buffer Cache in the database instance stores currently 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)



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.

Lesson Agenda

- Caching in the Database Instance
- Result Caching
- SQL Result Caching
- PL/SQL Result Caching
- Oracle Database In-Memory



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

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';
```

NAME	VALUE
1 result_cache_max_size	12353536

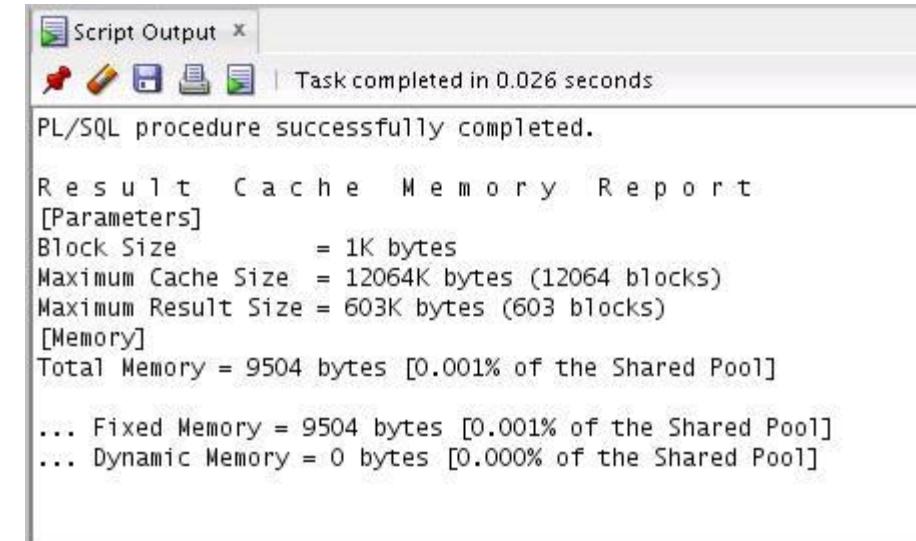
Setting the Result Cache Mode

- 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 the table level by using the `ALTER TABLE` command.
- You can set the result cache mode at the query level or PL/SQL program unit level explicitly.

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]

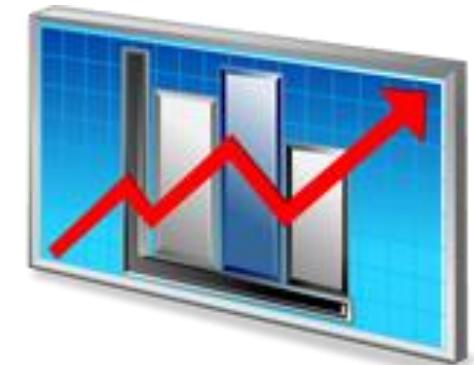
Lesson Agenda

- Caching in the Database Instance
- Result Caching
- SQL Result Caching
- PL/SQL Result Caching
- Oracle Database In-Memory



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.



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;
```

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
```

The screenshot shows the Oracle SQL Developer interface with a 'Script Output' window. The window title is 'Script Output x'. It displays the following output:

```
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]

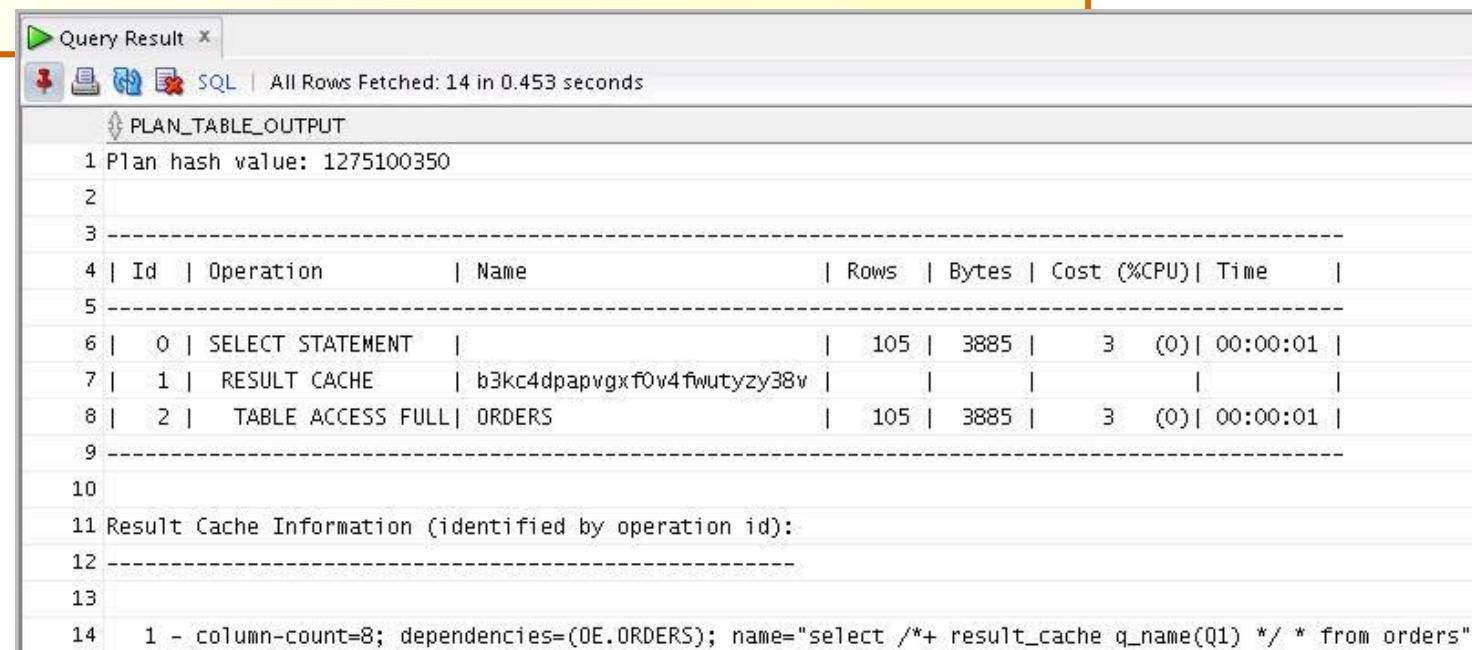
... Fixed Memory = 9504 bytes [0.001% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]

SQL> /
```

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'));
```



The screenshot shows the Oracle SQL Developer interface with a yellow background overlay containing the SQL code. Below the code is a 'Query Result' window titled 'PLAN_TABLE_OUTPUT'. The window displays the execution plan with the following details:

Id	Operation	Name	Rows	Bytes	Cost (%CPU)	Time
0	SELECT STATEMENT		105	3885	3 (0)	00:00:01
1	RESULT CACHE	b3kc4dpapvgxf0v4fwutzyzy38v				
2	TABLE ACCESS FULL	ORDERS	105	3885	3 (0)	00:00:01

Below the table, the message 'Result Cache Information (identified by operation id):' is shown, followed by a dependency entry:

```
1 - column-count=8; dependencies=(OE.ORDERS); name="select /*+ result_cache q_name(Q1) */ * from orders"
```

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

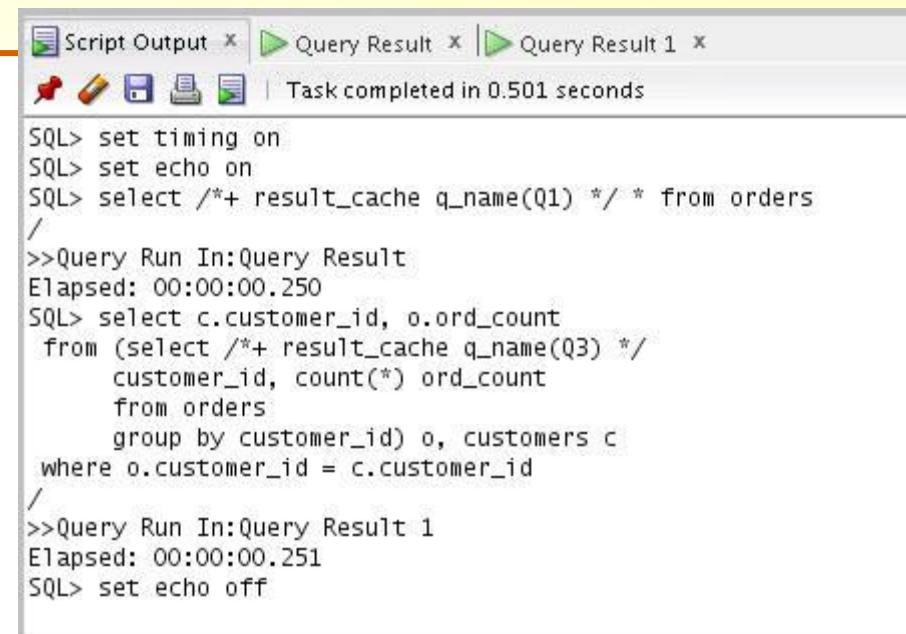
select plan_table_output from table(dbms_xplan.display('plan_table',
  null,'serial'));
```

Result Cache Information (identified by operation id):

```
-----  
3 - column-count=2; dependencies=(OE.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;
```

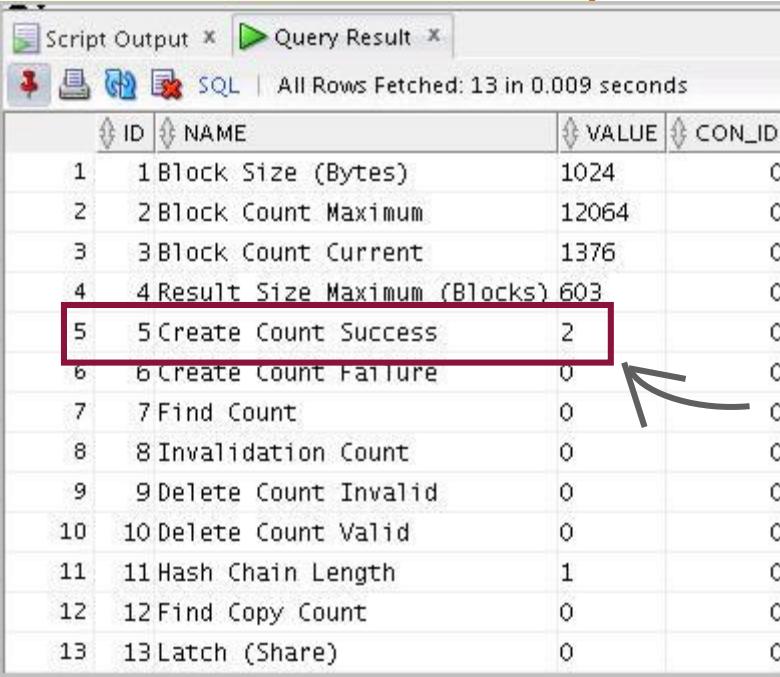


```
Script Output X | Query Result X | Query Result 1 X
Task completed in 0.501 seconds

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
```

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

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.

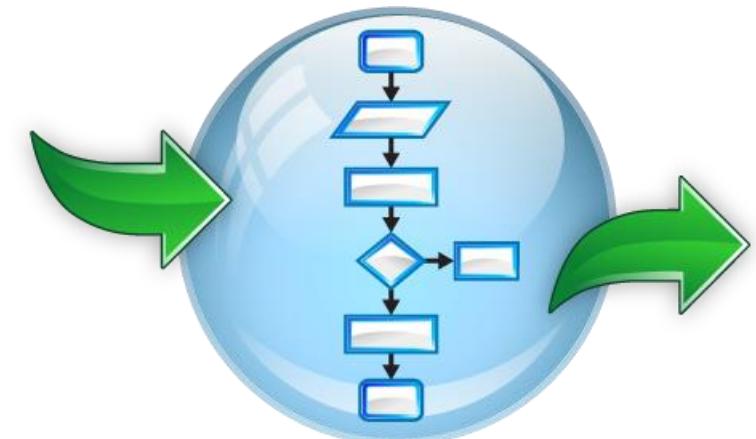
Lesson Agenda

- Caching in the Database Instance
- Result Caching
- SQL Result Caching
- PL/SQL Result Caching
- Oracle Database In-Memory



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.



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.

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
```

The screenshot shows the Oracle SQL Developer interface with a 'Script Output' window. The window title is 'Script Output'. It displays the following output:

```
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> /
```

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

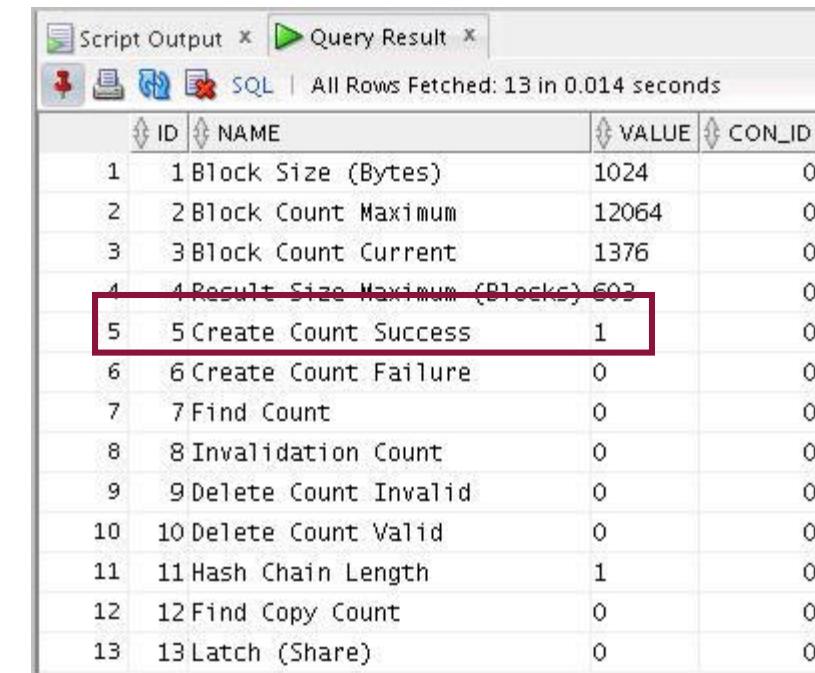
Calling the PL/SQL Function Inside a Query

```
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

Viewing Cache Results Created

```
col name format a55
select * from v$result_cache_statistics
/
```



The screenshot shows the Oracle SQL Developer interface with the 'Query Result' tab active. The results of the query are displayed in a table format:

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	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

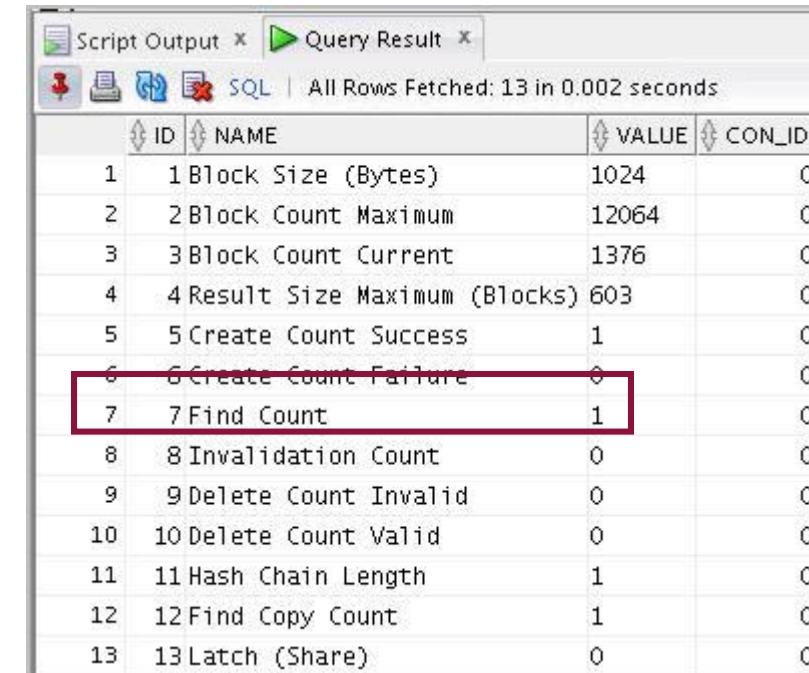
Calling the PL/SQL Function Again

```
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

Viewing Cache Results Found

```
col name format a55
select * from v$result_cache_statistics
/
```

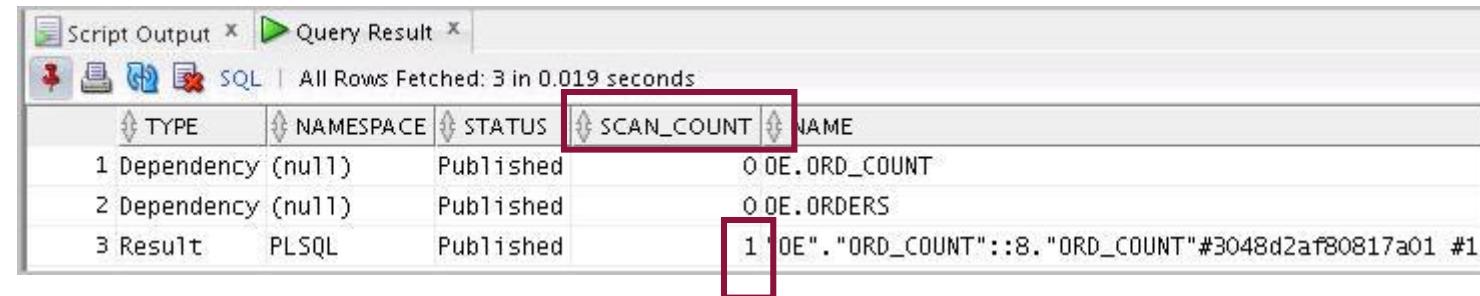


The screenshot shows the Oracle SQL Developer interface with a query result window open. The window title is "Query Result". It displays the output of the SQL query provided above. The results are presented in a table with columns: ID, NAME, VALUE, and CON_ID. The table contains 13 rows of data. Row 7, which corresponds to the "Find Count" statistic, is highlighted with a red border.

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

Confirming That the Cached Result Was Used

```
select type, namespace,status, scan_count,name  
from v$result_cache_objects  
/
```



The screenshot shows the Oracle SQL Developer interface with a query results window. The query has been executed successfully, returning three rows of data. The columns are labeled: TYPE, NAMESPACE, STATUS, SCAN_COUNT, and NAME. The SCAN_COUNT column is highlighted with a red box. The data is as follows:

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":8."ORD_COUNT"#3048d2af80817a01 #1

Lesson Agenda

- Caching in the Database Instance
- Result Caching
- SQL Result Caching
- PL/SQL Result Caching
- Oracle Database In-Memory



Oracle Database In-Memory

- A dual format architecture of 19c 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.

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 remain valid even after the database data accessed by the query is modified.



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



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



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



Quiz



You can create a function successfully that has both invoker's rights and is result cached.

- a. True
- b. False



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



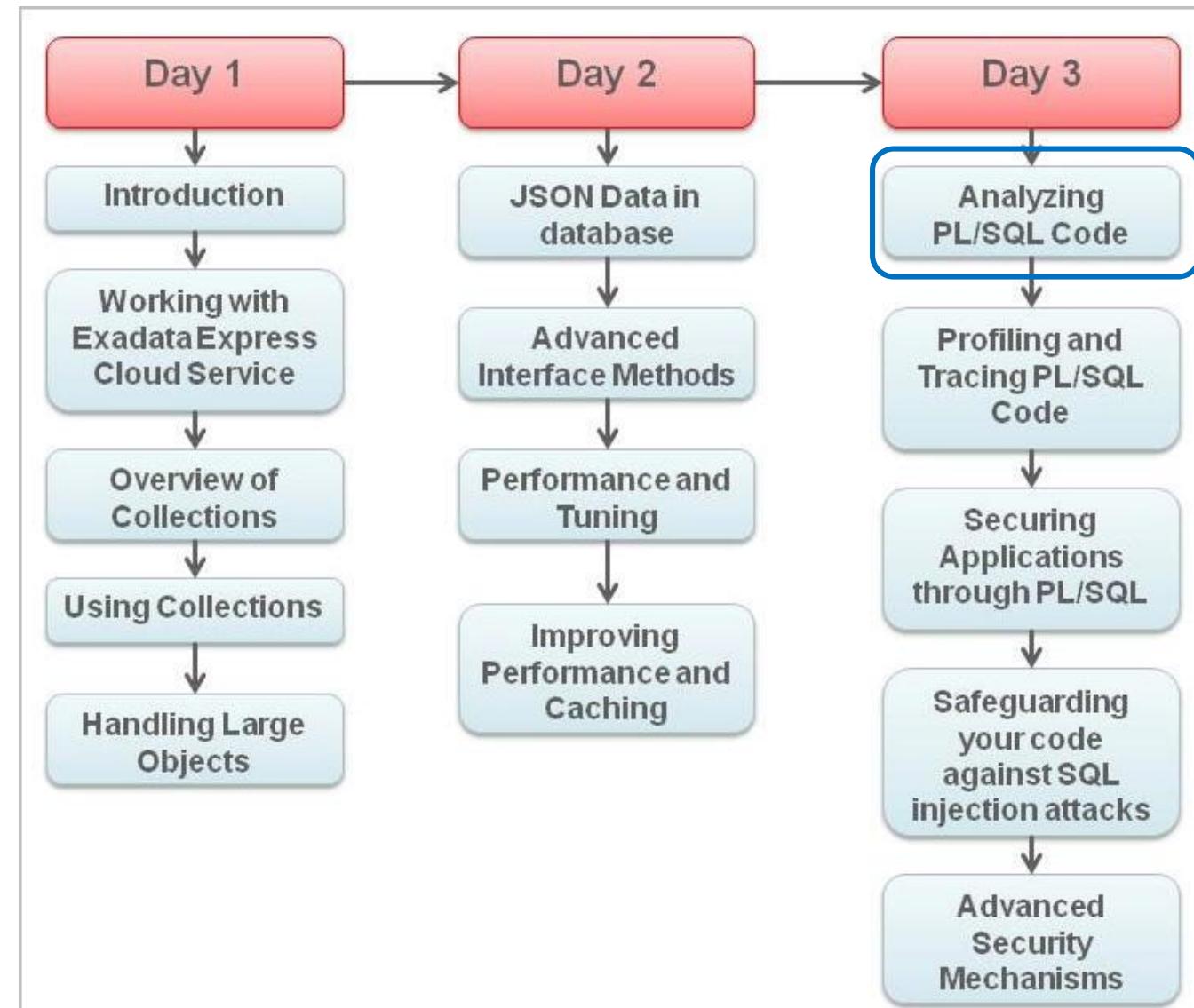
Practice 9: Overview

This practice covers the following topics:

- Writing code to use SQL caching
- Writing code to use PL/SQL caching

Analyzing PL/SQL Code

Course Agenda



Objectives

After completing this lesson, you should be able to:

- Use the supplied packages and dictionary views to find coding information
- Determine identifier types and usages with PL/Scope
- Use the DBMS_METADATA package



Lesson Agenda

- Data dictionary views
- PL/Scope
- Oracle Supplied Packages for Code Analysis



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?

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.

Data Dictionary Views

- Oracle provides data dictionary views that 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.



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 x

SQL | All Rows Fetched: 7 in 0.306 seconds

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\$" "XDI".")
4 PURCHASEORDER_T	1	TYPE "PURCHASEORDER_T" AS OBJECT ("SYS_XDBPD\$" "XDB".")
5 REJECTION_T	1	TYPE "REJECTION_T" AS OBJECT ("SYS_XDBPD\$" "XDB".")
6 SHIPPING_INSTRUCTIONS_T	1	TYPE "SHIPPING_INSTRUCTIONS_T" AS OBJECT ("SYS_XDBPD\$" "XDB".")
7 CUST_ADDRESS_TYP	8	, country_id CHAR(2)

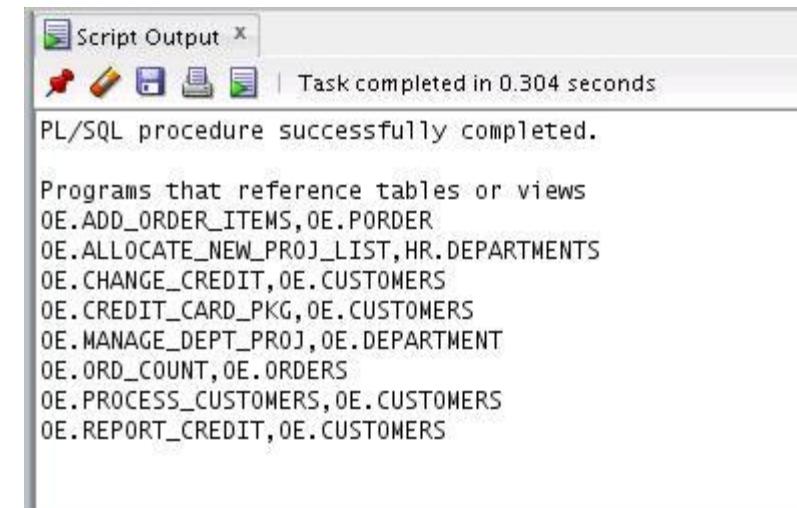
Analyzing PL/SQL Code

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;
/
```

Analyzing PL/SQL Code

```
EXECUTE query_code_pkg.encap_compliance
```

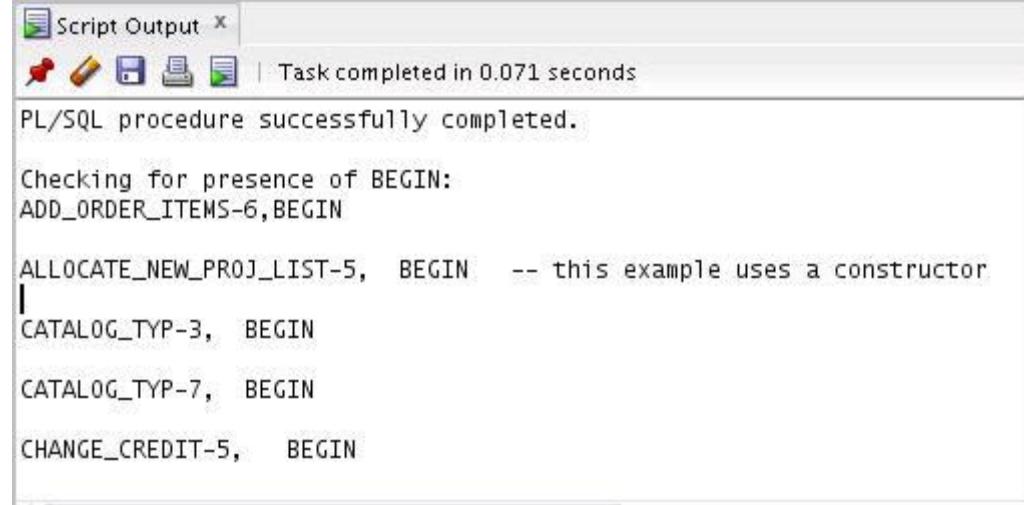


The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output X'. It contains the message 'Task completed in 0.304 seconds' and 'PL/SQL procedure successfully completed.' Below this, a list of programs is displayed, each followed by the tables or views it references:

- 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

Analyzing PL/SQL Code

```
EXECUTE query_code_pkg.find_text_in_code('BEGIN')
```



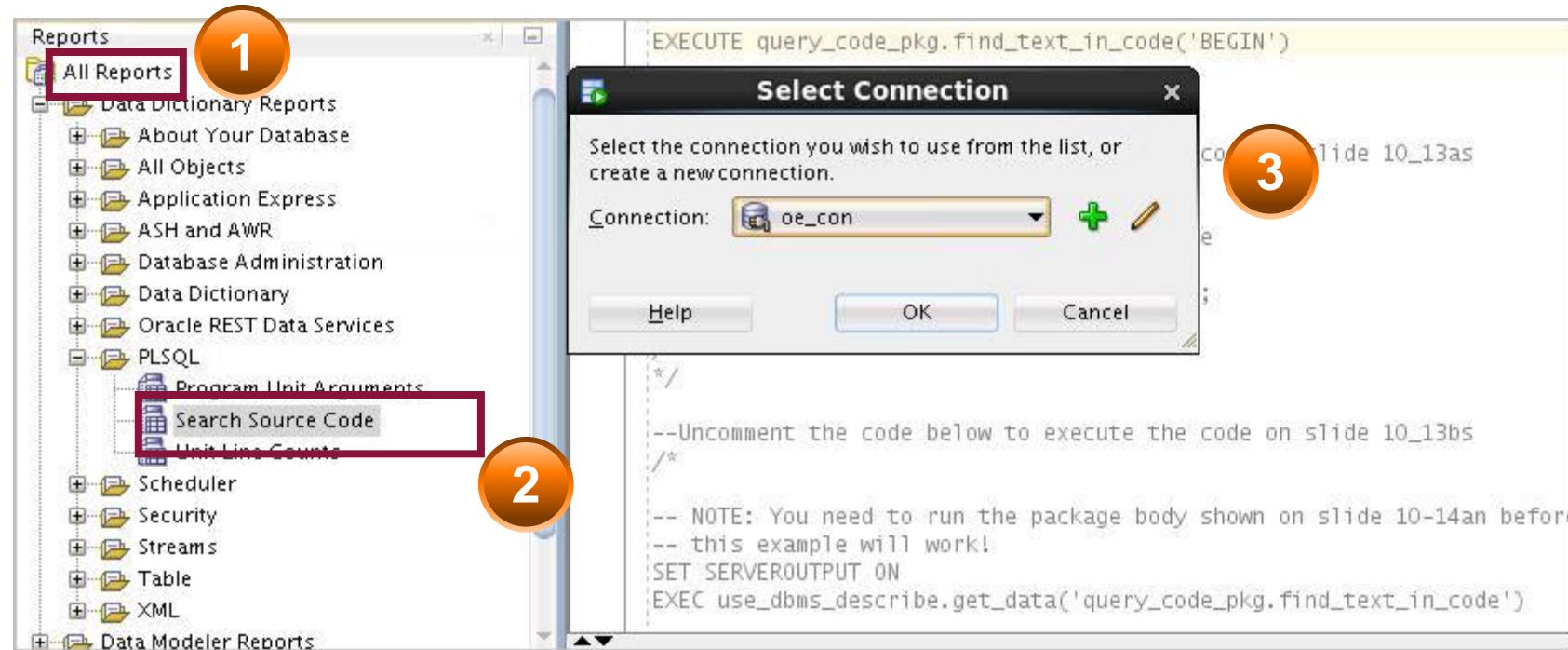
The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output'. It displays the message 'Task completed in 0.071 seconds' and 'PL/SQL procedure successfully completed.' Below this, it lists several PL/SQL procedures and their BEGIN statements:

- ADD_ORDER_ITEMS-6, BEGIN
- ALLOCATE_NEW_PROJ_LIST-5, BEGIN -- this example uses a constructor
- CATALOG_TYP-3, BEGIN
- CATALOG_TYP-7, BEGIN
- CHANGE_CREDIT-5, BEGIN

Note: The output image has partial result of the query.

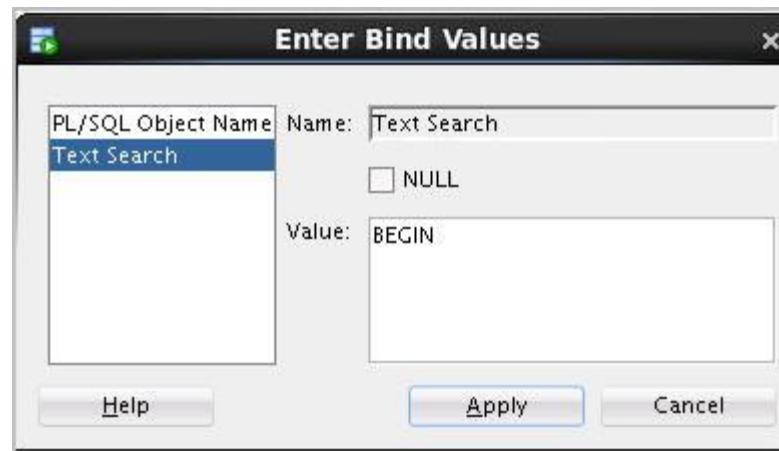
Using SQL Developer for Code Analysis

Use Reports:



Using SQL Developer for Code Analysis

Enter the text you want to search



4

The results window shows a table of found code snippets. The columns are: Owner, PL/SQL Object Name, Type, Line, and Text. The 'Text' column displays the word 'BEGIN' at various line numbers across different objects.

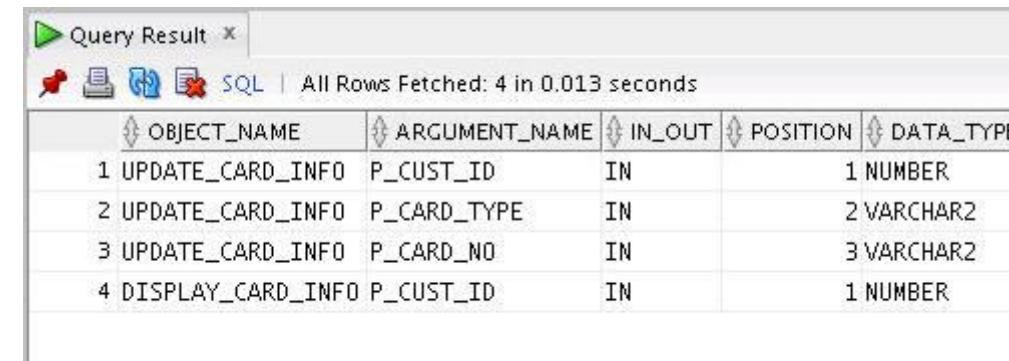
Owner	PL/SQL Object Name	Type	Line	Text
1	OE	ADD_ORDER_ITEMS	PROCEDURE	6 BEGIN
2	OE	ALLOCATE_NEW_PROJ_LIST	PROCEDURE	5 BEGIN -- this example uses a constructor
3	OE	CATALOG_TYP	TYPE BODY	3 BEGIN
4	OE	CATALOG_TYP	TYPE BODY	7 BEGIN
5	OE	CHANGE_CREDIT	PROCEDURE	5 BEGIN
6	OE	COMPOSITE_CATEGORY_TYP	TYPE BODY	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.

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';
```



The screenshot shows the 'Query Result' window in Oracle SQL Developer. The title bar says 'Query Result'. Below it, there are icons for Refresh, Undo, Redo, and Save, followed by 'SQL' and a status message 'All Rows Fetched: 4 in 0.013 seconds'. The main area is a grid table with the following data:

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

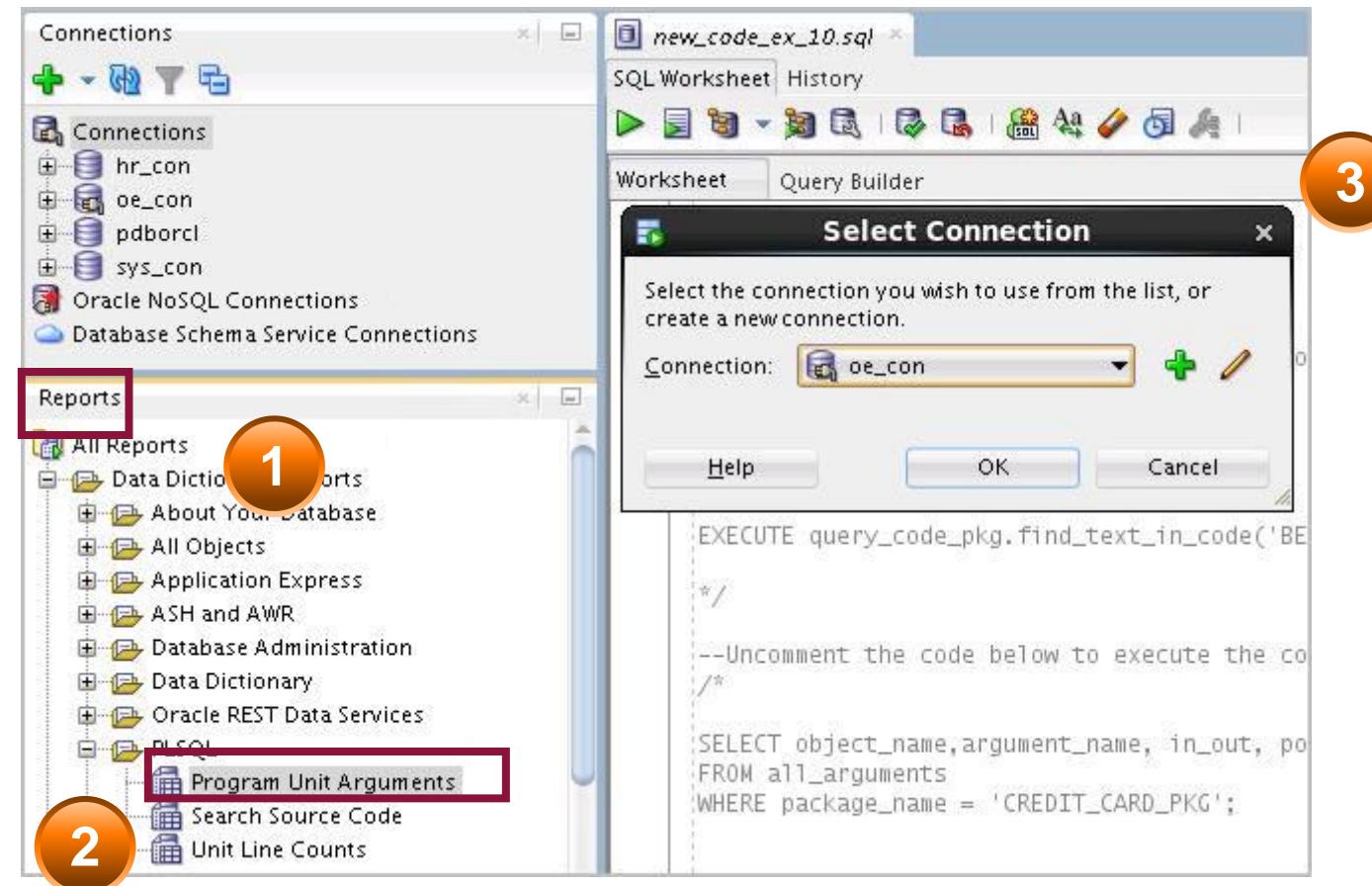
ALL_ARGUMENTS

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
OVERRLOAD		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

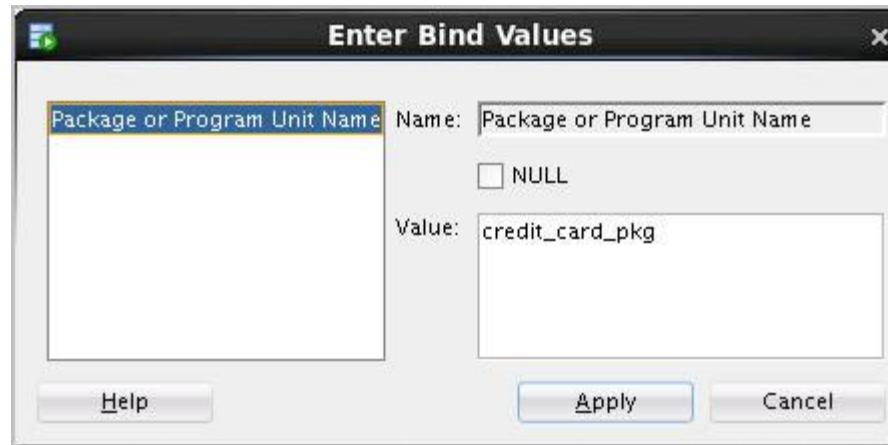
Using SQL Developer to Report on Arguments

Use Reports:



Using SQL Developer to Report on Arguments

4



5

The screenshot shows the 'Program Unit Arguments' window in SQL Developer. It displays a table of arguments for the package 'CREDIT_CARD_PKG'. The table has columns: Owner, Package, Program_Unit, Position, Argument, and In_Out. The data is as follows:

Owner	Package	Program_Unit	Position	Argument	In_Out
1	OE	CREDIT_CARD_PKG	DISPLAY_CARD_INFO	1 P_CUST_ID	In
2	OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	1 P_CUST_ID	In
3	OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	2 P_CARD_TYPE	In
4	OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	3 P_CARD_NO	In

Lesson Agenda

- Running reports on source code
- PL/Scope
- Oracle Supplied Packages for Code Analysis



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 that enables you to analyze SQL and PL/SQL code and answer such questions.

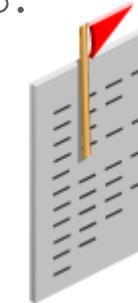


PL/Scope

- 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.

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.



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

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
```

Collecting Information on Identifiers

```
ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';  
  
ALTER PACKAGE credit_card_pkg COMPILE;
```

Identifier information is collected in the `USER_IDENTIFIERS` dictionary view, where you can:

- Perform a basic identifier search
- Use contexts to describe identifiers
- Find identifier actions
- Describe identifier actions

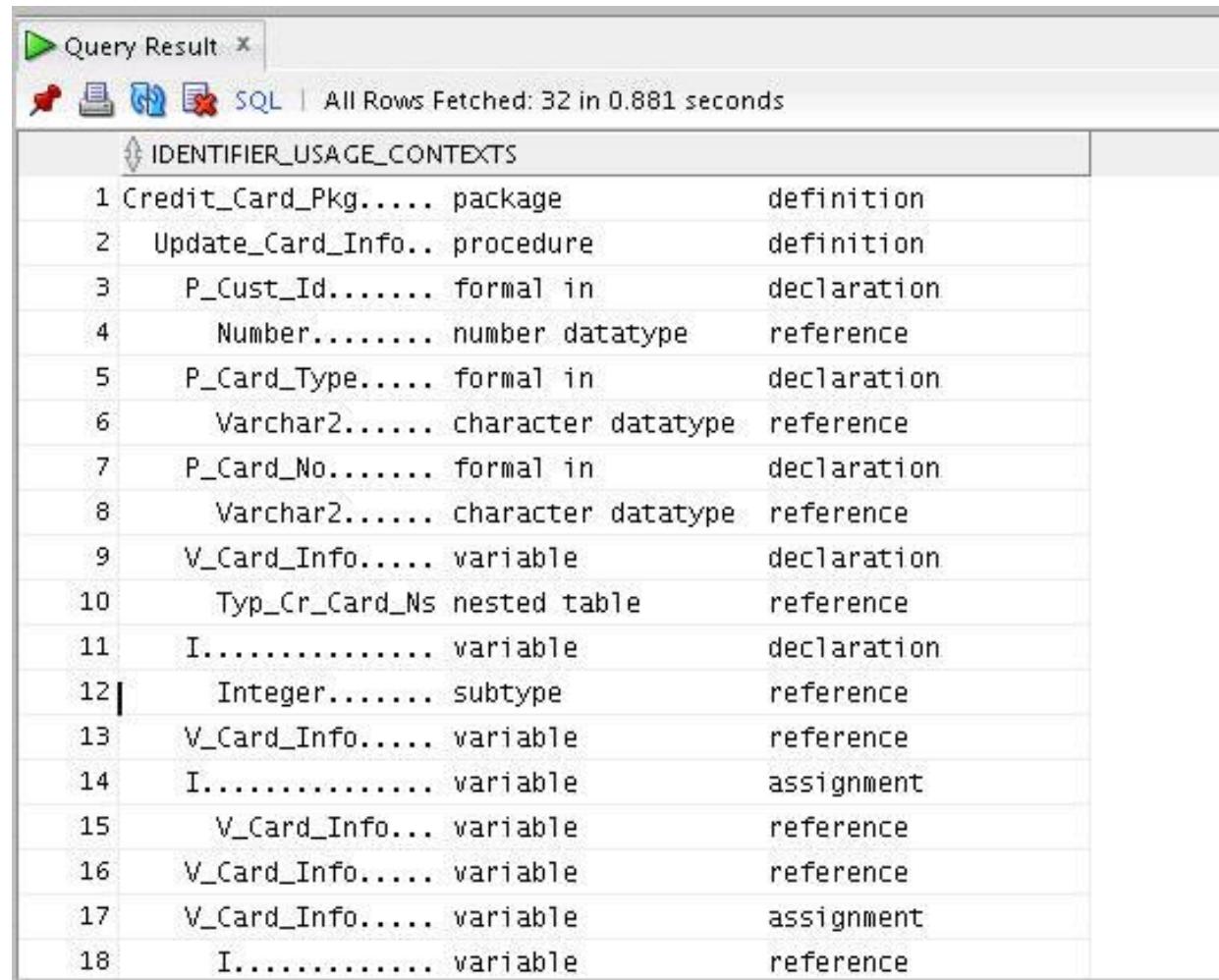
Viewing Identifier Information

Create a report based on the data in the `USER_IDENTIFIERS` view:

```
WITH v AS
  (SELECT Line,
          Col,
          INITCAP(NAME) Name,
          LOWER(TYPE) Type,
          LOWER(USAGE) Usage,
          USAGE_ID, USAGE_CONTEXT_ID
   FROM USER_IDENTIFIERS
  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;
```

Viewing Identifier Information

Results:



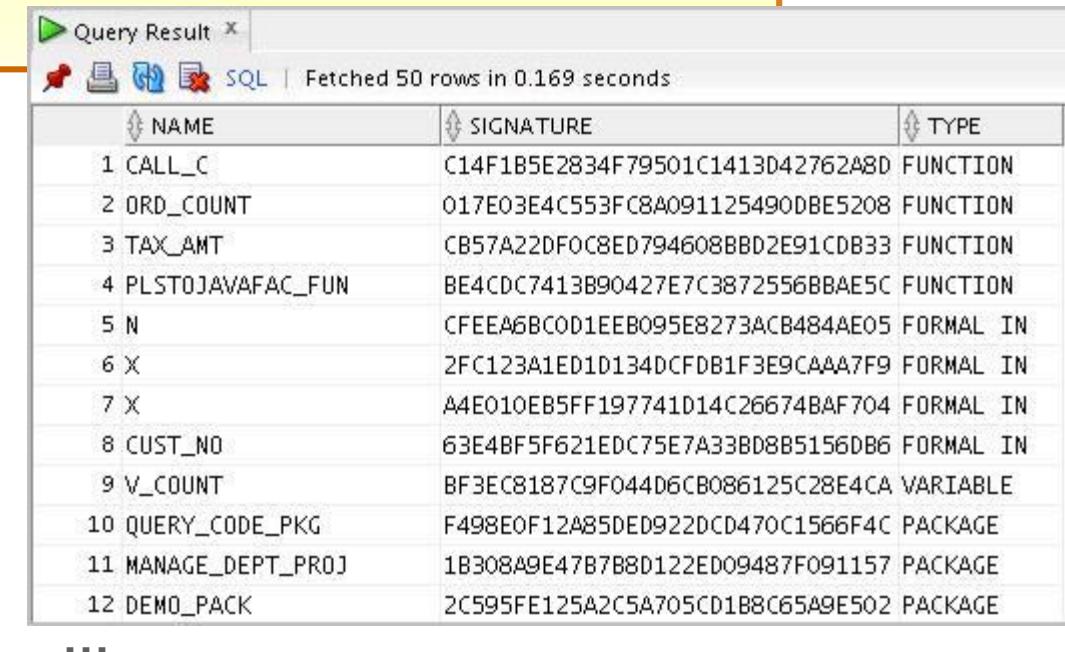
The screenshot shows a 'Query Result' window from Oracle SQL Developer. The title bar says 'Query Result' with a 'SQL' tab selected. Below the title bar, there are icons for refresh, export, and other functions, followed by 'SQL | All Rows Fetched: 32 in 0.881 seconds'. The main area is a table titled 'IDENTIFIER_USAGE_CONTEXTS' with 18 rows. The columns are numbered (1-18), identifier name, object type, and context.

#	Identifier Name	Object Type	Context
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

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;
```



The screenshot shows the 'Query Result' window from Oracle SQL Developer. The window title is 'Query Result x'. It displays the results of the SQL query above. The results are presented in a table with three columns: 'NAME', 'SIGNATURE', and 'TYPE'. The table has 12 rows, each representing a declaration in the schema. The 'NAME' column contains identifiers like 'CALL_C', 'ORD_COUNT', 'TAX_AMT', etc., with row numbers 1 through 12. The 'SIGNATURE' column contains long hex strings representing the object's signature. The 'TYPE' column indicates the object type, such as 'FUNCTION', 'FORMAL IN', 'VARIABLE', or 'PACKAGE'. The table has a header row and 12 data rows.

NAME	SIGNATURE	TYPE
1 CALL_C	C14F1B5E2834F79501C1413D42762A8D	FUNCTION
2 ORD_COUNT	017E03E4C553FC8A091125490DBE5208	FUNCTION
3 TAX_AMT	CB57A22DF0C8ED794608BB02E91CDB33	FUNCTION
4 PLSTOJAVAFAAC_FUN	BE4CDC7413B90427E7C3872556BBAE5C	FUNCTION
5 N	CFEEA6BC0D1EEB095E8273ACB484AE05	FORMAL IN
6 X	2FC123A1ED1D134DCFDB1F3E9CAA7F9	FORMAL IN
7 X	A4E010EB5FF197741D14C26674BAF704	FORMAL IN
8 CUST_NO	63E4BF5F621EDC75E7A33BD8B5156DB6	FORMAL IN
9 V_COUNT	BF3EC8187C9F044D6CB086125C28E4CA	VARIABLE
10 QUERY_CODE_PKG	F498EOF12A85DED922DCD470C1566F4C	PACKAGE
11 MANAGE_DEPT_PROJ	1B308A9E47B7B8D122ED09487F091157	PACKAGE
12 DEMO_PACK	2C595FE125A2C5A705CD1B8C65A9E502	PACKAGE

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

All Rows Fetched: 4 in 0.185 seconds

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	E5E48B87FD19043F4240B84ECA77E916

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;
```

Query Result x

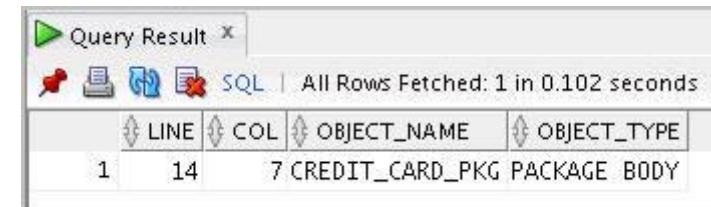
All Rows Fetched: 7 in 0.112 seconds

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

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';
```



The screenshot shows the 'Query Result' window from Oracle SQL Developer. The window title is 'Query Result'. It displays a single row of data with the following columns: LINE, COL, OBJECT_NAME, and OBJECT_TYPE. The data is as follows:

LINE	COL	OBJECT_NAME	OBJECT_TYPE
1	14	7 CREDIT_CARD_PKG	PACKAGE BODY

Below the table, a status message reads: 'All Rows Fetched: 1 in 0.102 seconds'.

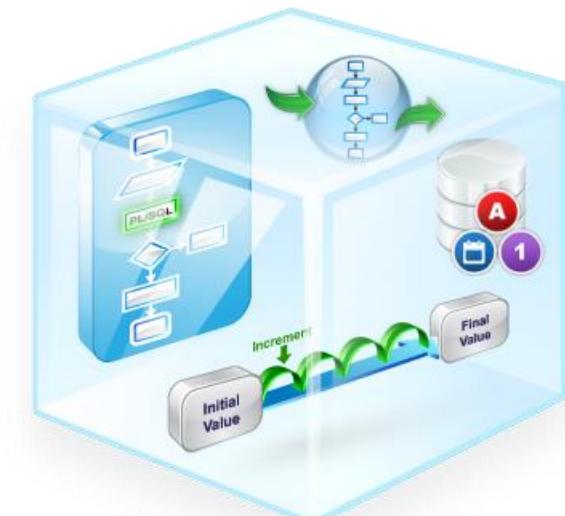
Lesson Agenda

- Running reports on source code
- PL/Scope
- Oracle Supplied Packages for Code Analysis



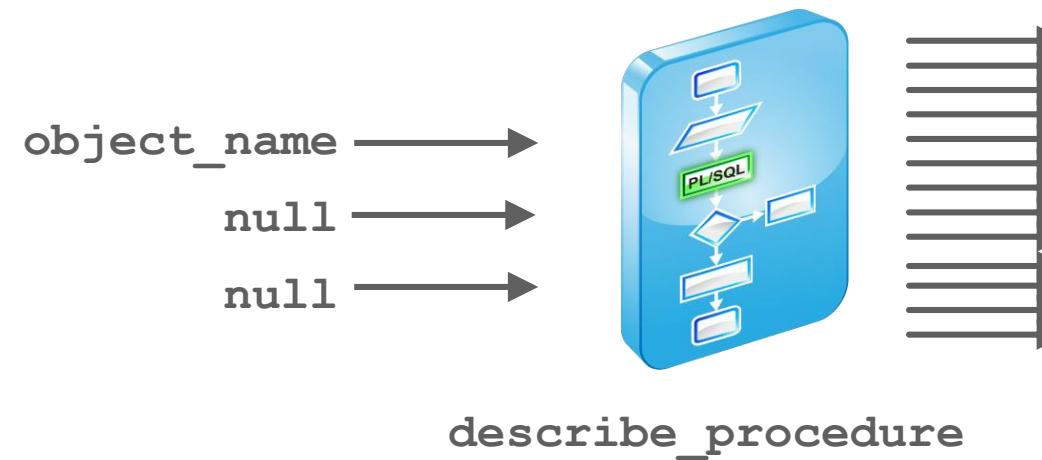
Oracle Supplied Packages for Code Analysis

- Following are Oracle packages you can use for PL/SQL code analysis:
 - DBMS_DESCRIBE
 - DBMS_UTLILITY
 - DBMS_METADATA
 - UTL_CALL_STACK



Using DBMS_DESCRIBE

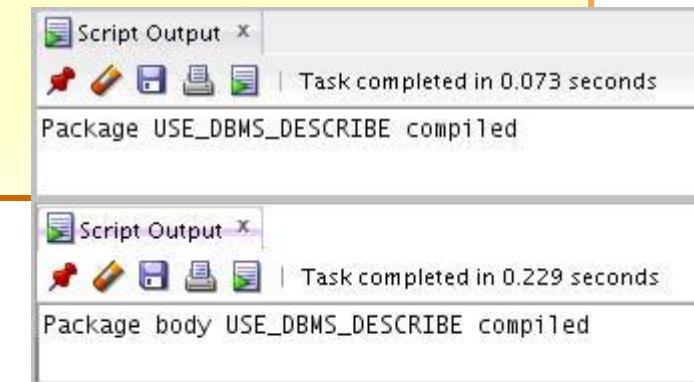
- Can be used to retrieve information about a PL/SQL object
- Contains one procedure: DESCRIBE_PROCEDURE
- Includes:
 - Three scalar IN parameters
 - One scalar OUT parameter
 - 12 associative array OUT parameters



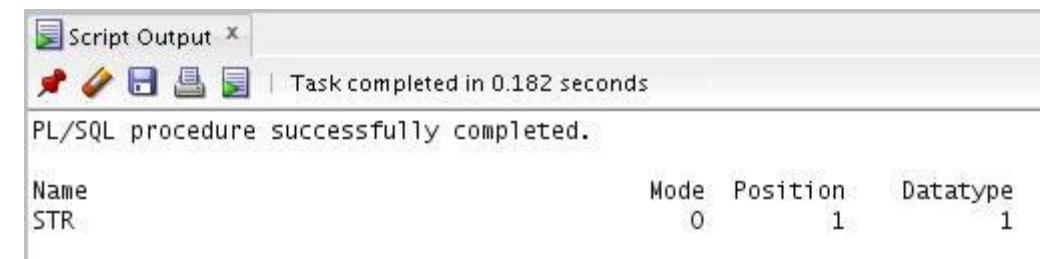
Using DBMS_DESCRIBE

Create a package to call the DBMS_DESCRIBE.DESCRIBE_PROCEDURE routine:

```
CREATE OR REPLACE PACKAGE use_dbms_describe
IS
    PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/
```



```
EXEC use_dbms_describe.get_data('query_code_pkg.find_text_in_code')
```



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 that 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.

Using DBMS_UTILITY.FORMAT_CALL_STACK

Consider the given code:

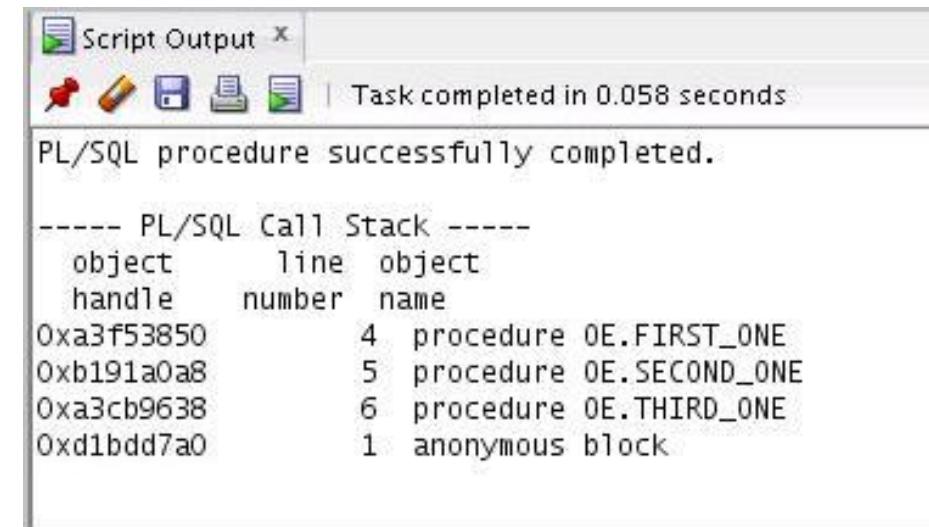
```
CREATE OR REPLACE PROCEDURE first_one
IS
BEGIN
    dbms_output.put_line(
        substr(dbms_utility.format_call_stack, 1, 255));
END;

CREATE OR REPLACE PROCEDURE second_one
...
CREATE OR REPLACE PROCEDURE third_one
IS
BEGIN
    null;
    null;
    second_one;
END;
```

Using DBMS_UTILITY.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
```



The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output'. Below the title bar, there are several icons: a red exclamation mark, a pencil, a blue folder, a green document, and a magnifying glass. To the right of these icons, the text 'Task completed in 0.058 seconds' is displayed. The main content area of the window shows the following text:
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

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.



Using DBMS_UTILITy

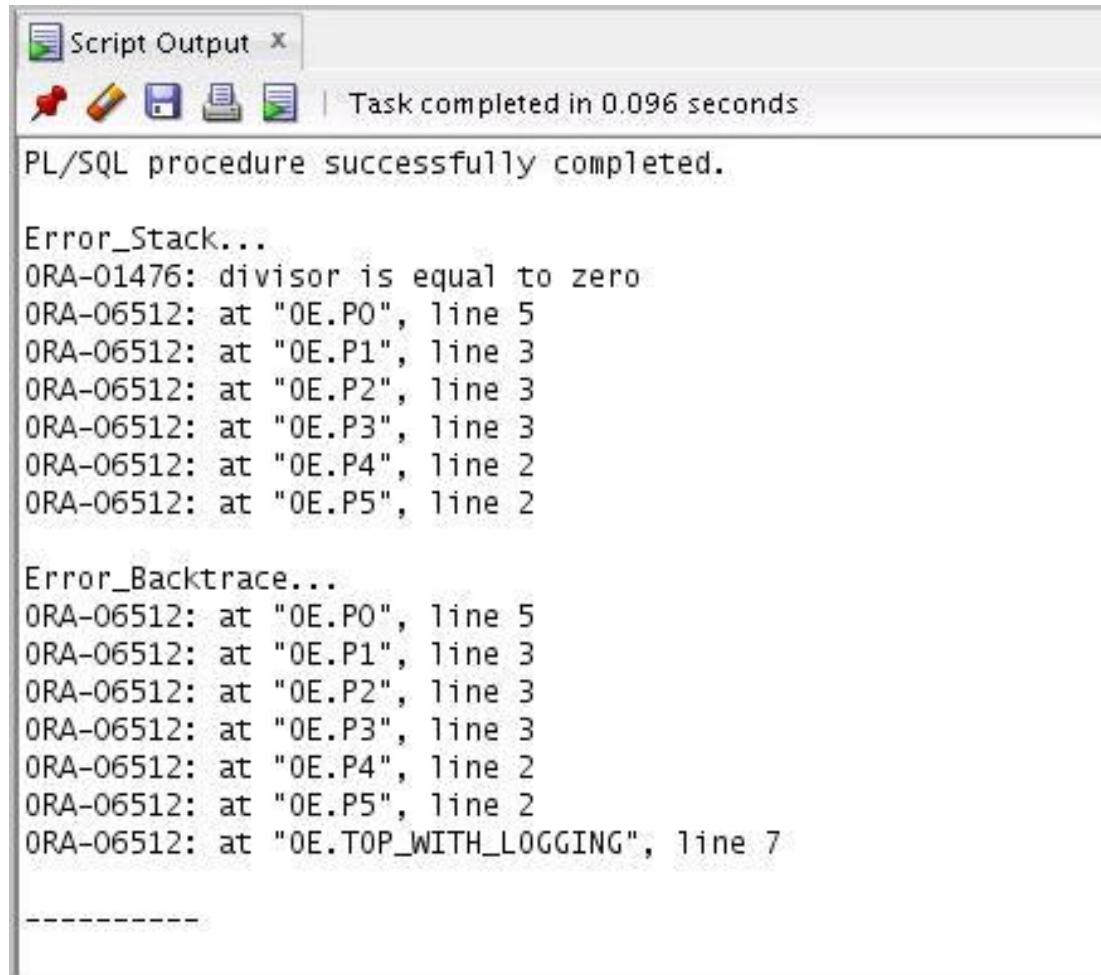
```
CREATE OR REPLACE PROCEDURE top_with_logging IS
  -- NOTE: SQLERRM in principle gives the same info
  -- as format_error_stack.
  -- But SQLERRM is subject to some length limits,
  -- while format_error_stack is not.
BEGIN
  P5(); -- this procedure, in turn, calls others,
         -- building a stack. 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;
/
```

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;
```

Finding Error Information

Results:



The screenshot shows the 'Script Output' window from Oracle SQL Developer. The window title is 'Script Output'. At the top, there are several icons: a red exclamation mark, a pencil, a blue folder, a printer, and a green document. To the right of these icons, the text 'Task completed in 0.096 seconds' is displayed. Below this, the message 'PL/SQL procedure successfully completed.' is shown. Underneath, two sections of error information are listed: 'Error_Stack...' and 'Error_Backtrace...'. Both sections list multiple ORA-06512 errors, indicating a divisor is equal to zero at various points in the procedure. A dashed line at the bottom indicates the end of the 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

-----
```

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 back up certain objects that are going to be changed or dropped.

DBMS_METADATA Subprograms

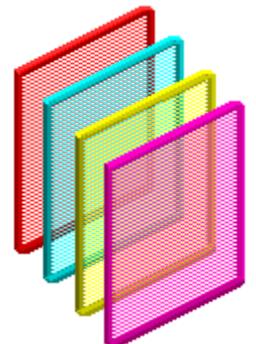
- You have subprograms for retrieving multiple objects from the database.
- You also have subprograms for submitting XML to the database.
- The package provides two types of retrieval interfaces for two types of usage:
 - For programmatic use
 - For use in SQL queries and for ad hoc browsing

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.

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.



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');
...
```

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');
```

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, ← 4
        'MODIFY', NULL, 'TABLE');
    DBMS_METADATA.SET_REMAP_PARAM(v_th1, ← 5
        'REMAP_TABLESPACE', 'SYSTEM', 'TBS1');
    v_th2:=DBMS_METADATA.ADD_TRANSFORM(v_hdl, 'DDL'); ← 6
    DBMS_METADATA.SET_TRANSFORM_PARAM(v_th2, ← 7
        'SQLTERMINATOR', TRUE);
    DBMS_METADATA.SET_TRANSFORM_PARAM(v_th2, ← 8
        'REF_CONSTRAINTS', FALSE, 'TABLE');
LOOP
    v_doc := DBMS_METADATA.FETCH_DDL(v_hdl); ← 7
    EXIT WHEN v_doc IS NULL;
END LOOP;
DBMS_METADATA CLOSE(v_hdl); ← 8
END;
```

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;
/
```

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.
Where xxx is:	DDL or XML

Using the UTL_CALL_STACK Package

The UTL_CALL_STACK package in Oracle Database 19c. 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

DEPRECATE Pragma

- DEPRECATE pragma marks a PL/SQL element as deprecated.
- You can apply it to a whole unit or a subprogram within a 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.

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



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



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



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



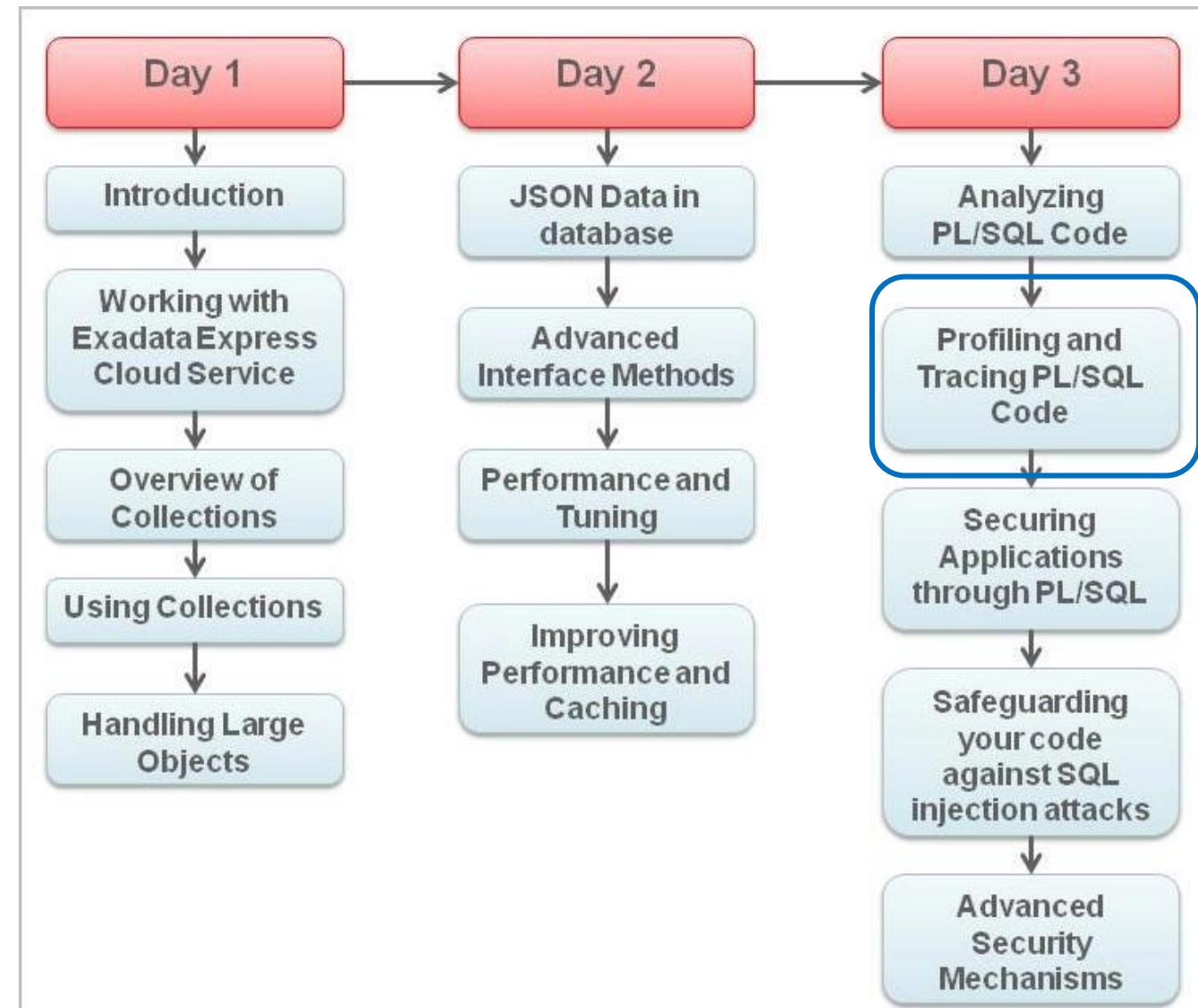
Practice 10: Overview

This practice covers the following topics:

- Analyzing PL/SQL Code
- Using PL/Scope
- Using DBMS_METADATA

Profiling and Tracing PL/SQL Code

Course Agenda



Objectives

After completing this lesson, you should be able to:

- Trace PL/SQL program execution
- Profile PL/SQL applications



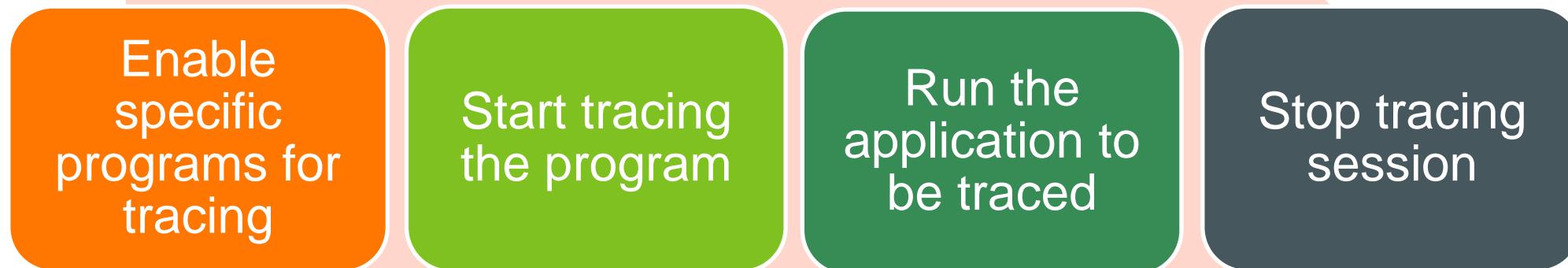
Lesson Agenda

- Tracing PL/SQL program execution
- Profiling PL/SQL applications



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.



Tracing PL/SQL Execution

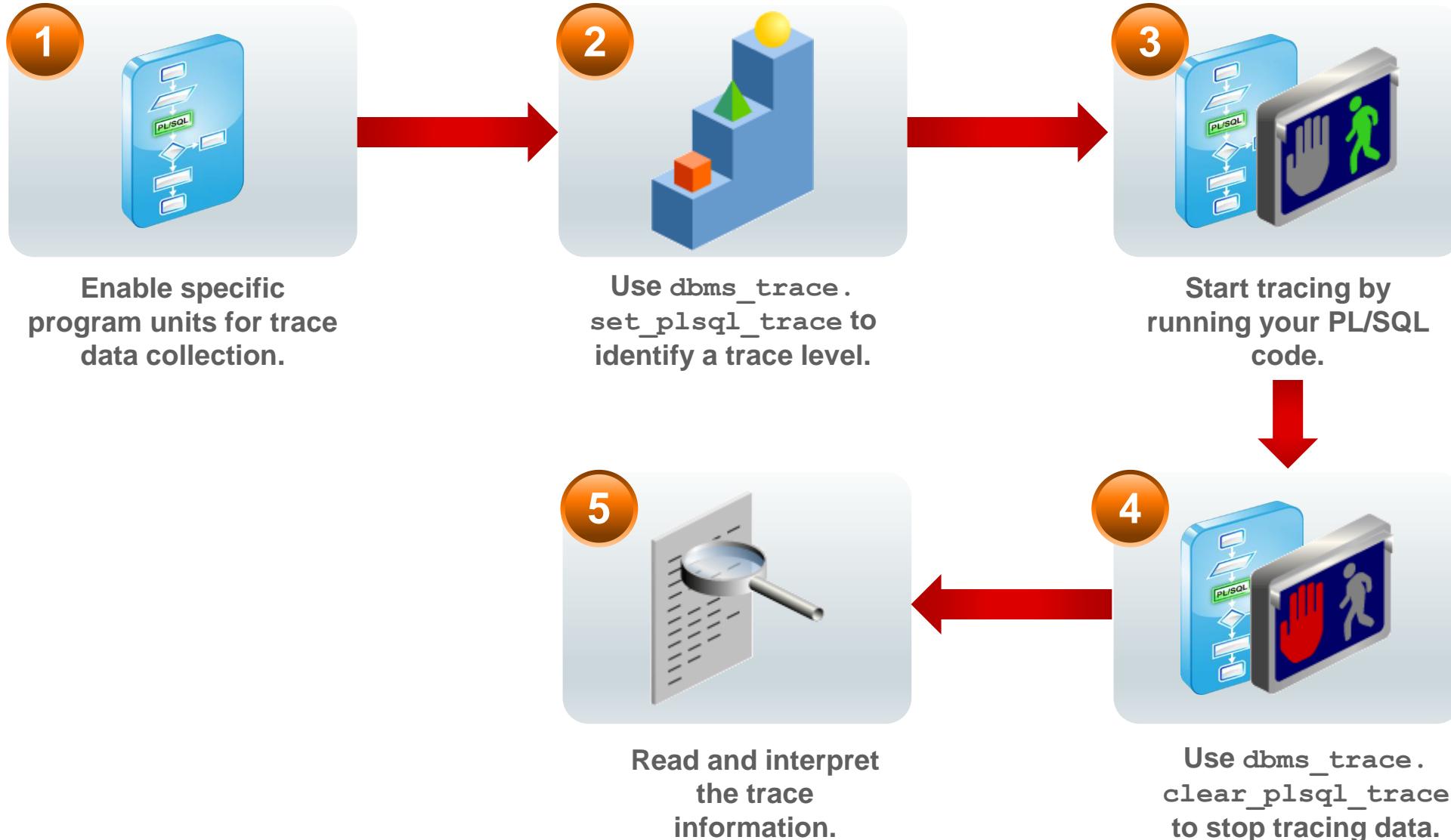
The DBMS_TRACE package contains the following PL/SQL subprograms:

Subprogram	Description
CLEAR_PLSQL_TRACE procedure	Stops trace data dumping in session
GET_PLSQL_TRACE_LEVEL function	Gets the trace level
PLSQL_TRACE_VERSION procedure	Gets the version number of the trace package
SET_PLSQL_TRACE procedure	Starts tracing in the current session

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.

Tracing PL/SQL: Steps



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];
```

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
```

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.

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.

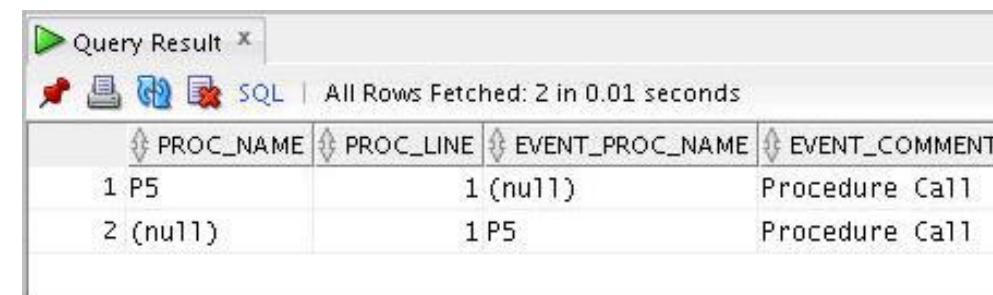


plsql_trace_runs and plsql_trace_events

```
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';
```



PROC_NAME	PROC_LINE	EVENT_PROC_NAME	EVENT_COMMENT
1 P5		1 (null)	Procedure Call
2 (null)		1 P5	Procedure Call

Lesson Agenda

- Tracing PL/SQL program execution
- Profiling PL/SQL applications



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.

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

Hierarchical Profiling Concepts

The PL/SQL hierarchical profiler consists of the:

- Data collection component
 - `start_profiling` procedure
 - `Stop_profiling` procedure
- Analyzer component
 - `analyze` function

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?

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
```

Using the PL/SQL Profiler

```
1 BEGIN  
-- start profiling  
DBMS_HPROF.START_PROFILING('PROFILE_DATA', 'pd_cc_pkg.txt');  
END;
```

```
2 DECLARE  
    v_card_info typ_cr_card_nst;  
BEGIN  
-- run application  
    credit_card_pkg.update_card_info  
        (154, 'Discover', '123456789');  
END;
```

```
3 BEGIN  
    DBMS_HPROF.STOP_PROFILING;  
END;
```

Understanding Raw Profiler Data

The pd_cc_pkg.txt file has the raw profiler data in it.

```
pd_cc_pkg.txt ×
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"#3a2b06cfa1322e42 #3
P#X 69
P#C SQL."OE"."CREDIT_CARD_PKG":11."_static_sql_exec_line9" #9."3zjmnccb1j06t6"
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#R
P#! PL/SQL Timer Stopped
```

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_INFO	Contains parent-child profiler information

Using DBMS_HPROF.ANALYZE

DBMS_HPROF.analyze:

- Analyzes the raw profiler data
- Generates hierarchical profiler information in the profiler database tables
- Definition:

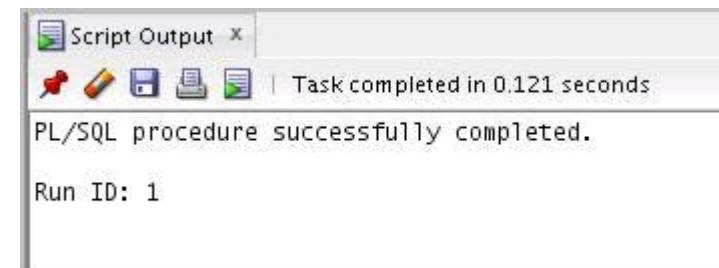
```
DBMS_HPROF.analyze(
    location      IN VARCHAR2,
    filename      IN VARCHAR2,
    summary_mode  IN BOOLEAN      DEFAULT FALSE,
    trace         IN VARCHAR2    DEFAULT NULL,
    skip          IN PLS_INTEGER DEFAULT 0,
    collect        IN PLS_INTEGER DEFAULT NULL,
    run_comment   IN VARCHAR2    DEFAULT NULL)
RETURN NUMBER;
```

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;
```

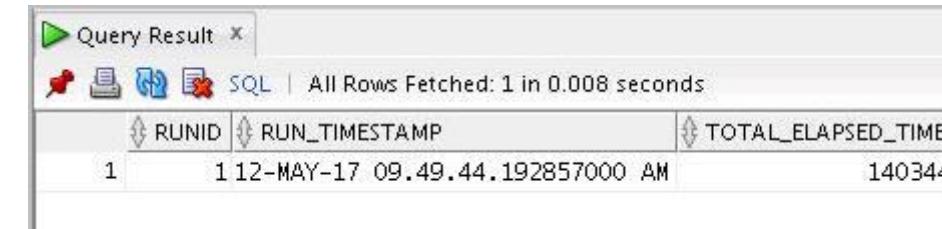
- This function returns a unique run identifier for the run. You can use this identifier to look up results corresponding to this run from the hierarchical profiler tables.



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;
```



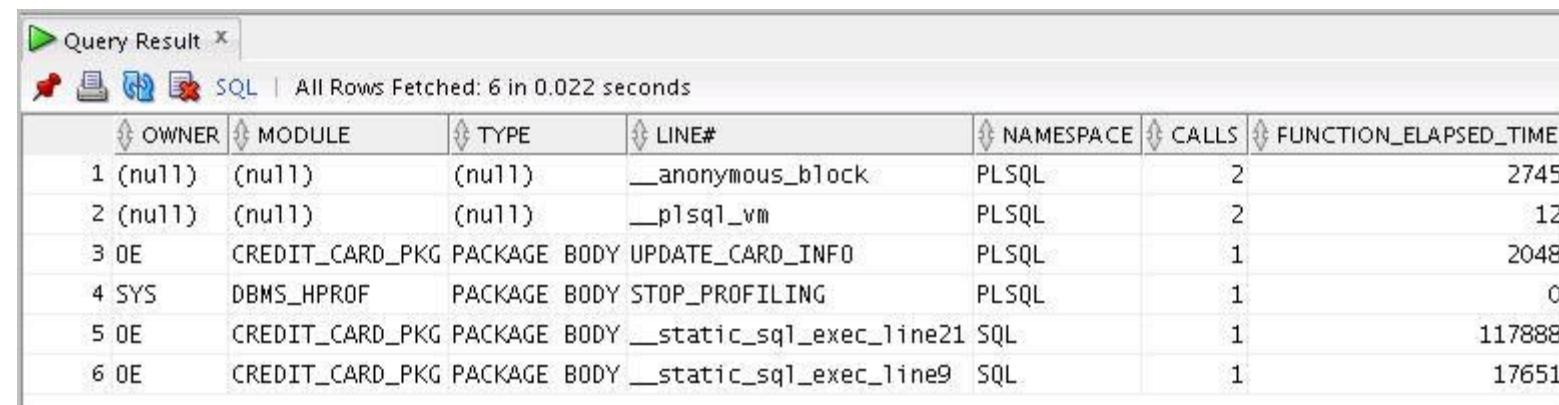
The screenshot shows a 'Query Result' window from Oracle SQL Developer. The window title is 'Query Result'. It contains a toolbar with icons for Run, Stop, Refresh, Paste, and SQL. Below the toolbar, it says 'All Rows Fetched: 1 in 0.008 seconds'. The result set is displayed in a table with three columns: RUNID, RUN_TIMESTAMP, and TOTAL_ELAPSED_TIME. The data row is: 1, 12-MAY-17 09.49.44.192857000 AM, 140344.

RUNID	RUN_TIMESTAMP	TOTAL_ELAPSED_TIME
1	12-MAY-17 09.49.44.192857000 AM	140344

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 open. The window title is 'Query Result x'. Below the title, there are icons for Refresh, Undo, Redo, and SQL, followed by the text 'All Rows Fetched: 6 in 0.022 seconds'. The main area displays 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

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.



Using plshprof

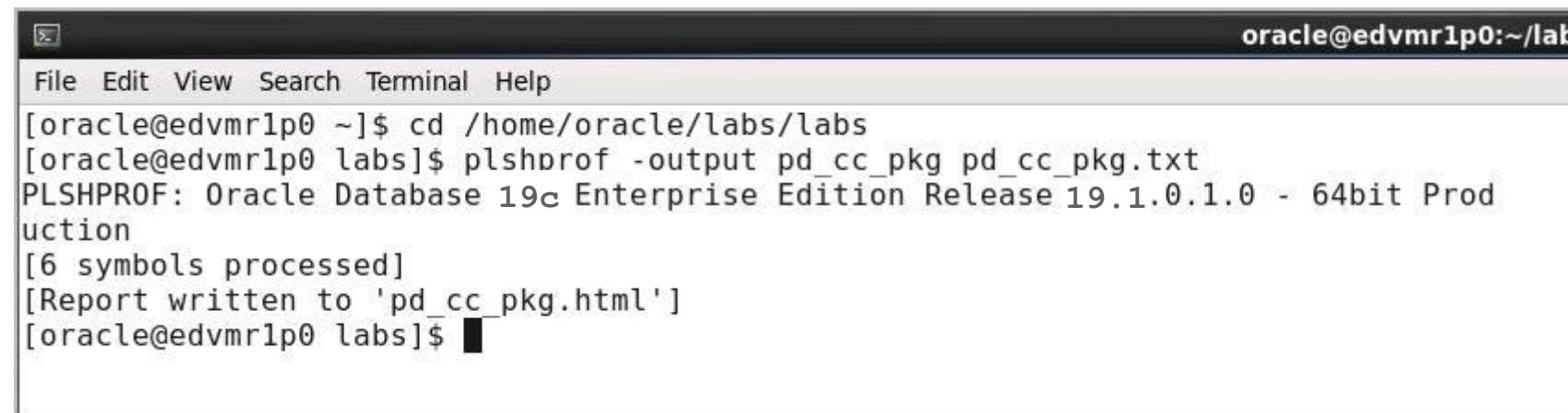
After generating the raw profiler output file:

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:

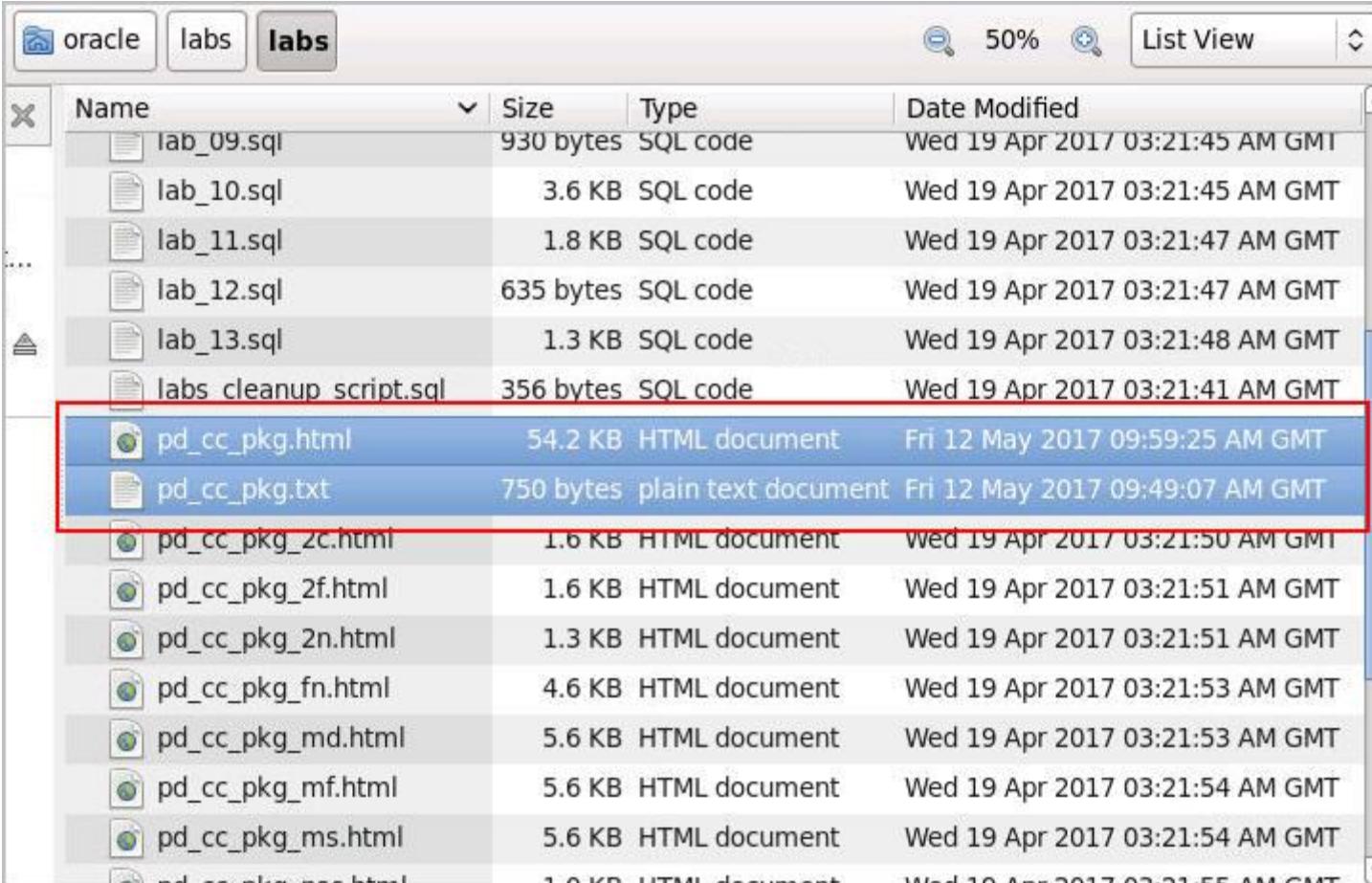


A screenshot of a terminal window titled "oracle@edvmr1p0:~/lab". The window shows the following command being run and its output:

```
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 19c Enterprise Edition Release 19.1.0.1.0 - 64bit Production
[6 symbols processed]
[Report written to 'pd_cc_pkg.html']
[oracle@edvmr1p0 labs]$ █
```

Using plshprof

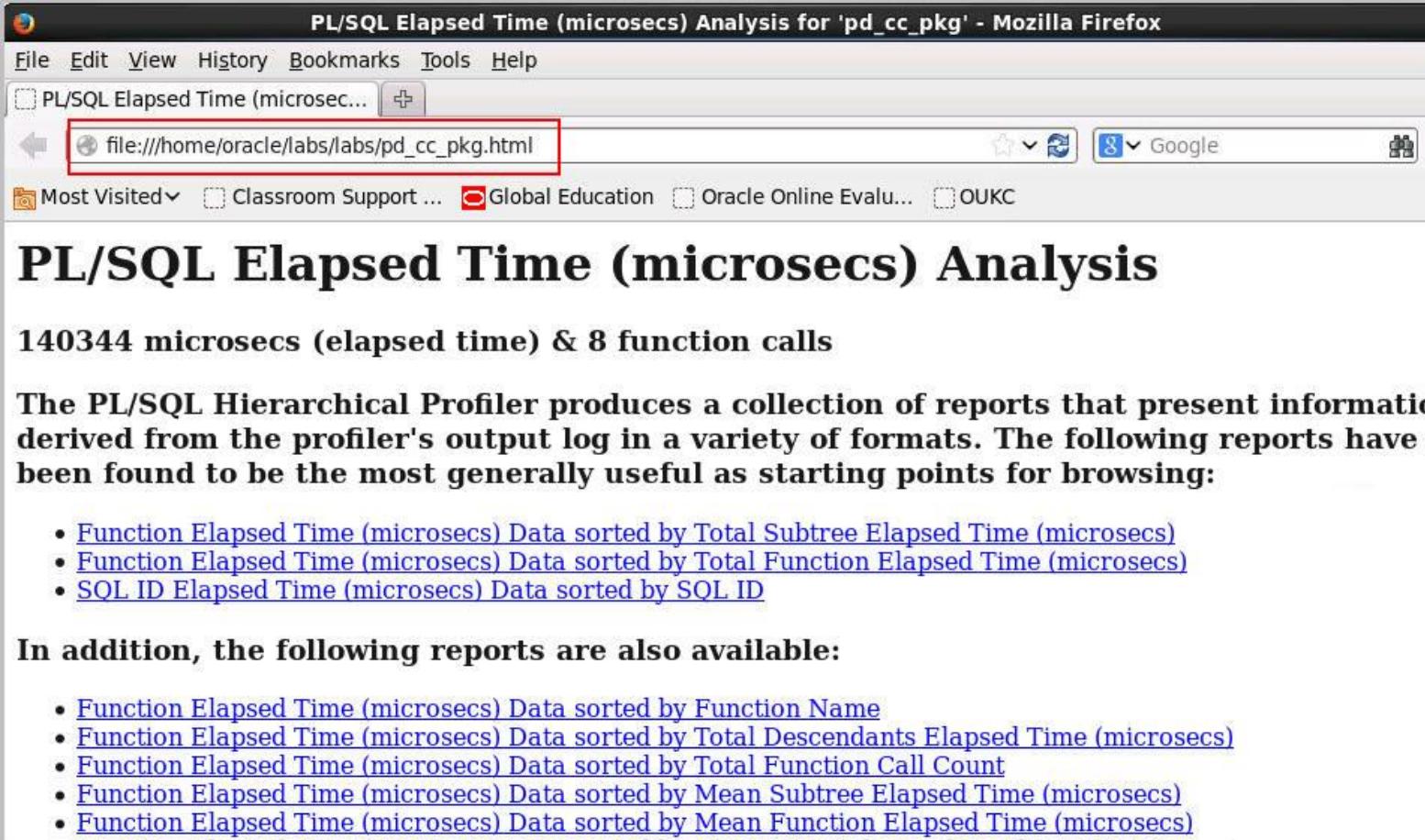
Generated files:



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
pd_cc_pkg.html	54.2 KB	HTML document	Fri 12 May 2017 09:59:25 AM GMT
pd_cc_pkg.txt	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

Using plshprof

3. Open pd_cc_pkg.html in a browser:



PL/SQL Elapsed Time (microsecs) Analysis for 'pd_cc_pkg' - Mozilla Firefox

File Edit View History Bookmarks Tools Help

file:///home/oracle/labs/labs/pd_cc_pkg.html

Most Visited Classroom Support ... Global Education Oracle Online Evalu... OUKC

PL/SQL Elapsed Time (microsecs) Analysis

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 \(microsecs\)](#)

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](#)

PL/SQL Elapsed Time (microsecs) Analysis for 'pd_cc_pkg' - Mozilla Firefox

File Edit View History Bookmarks Tools Help

PL/SQL Elapsed Time (microsec... +

file:///home/oracle/labs/labs/pd_cc_pkg.html#hprof_ts

Most Visited Classroom Support ... Global Education Oracle Online Evalu... OUKC

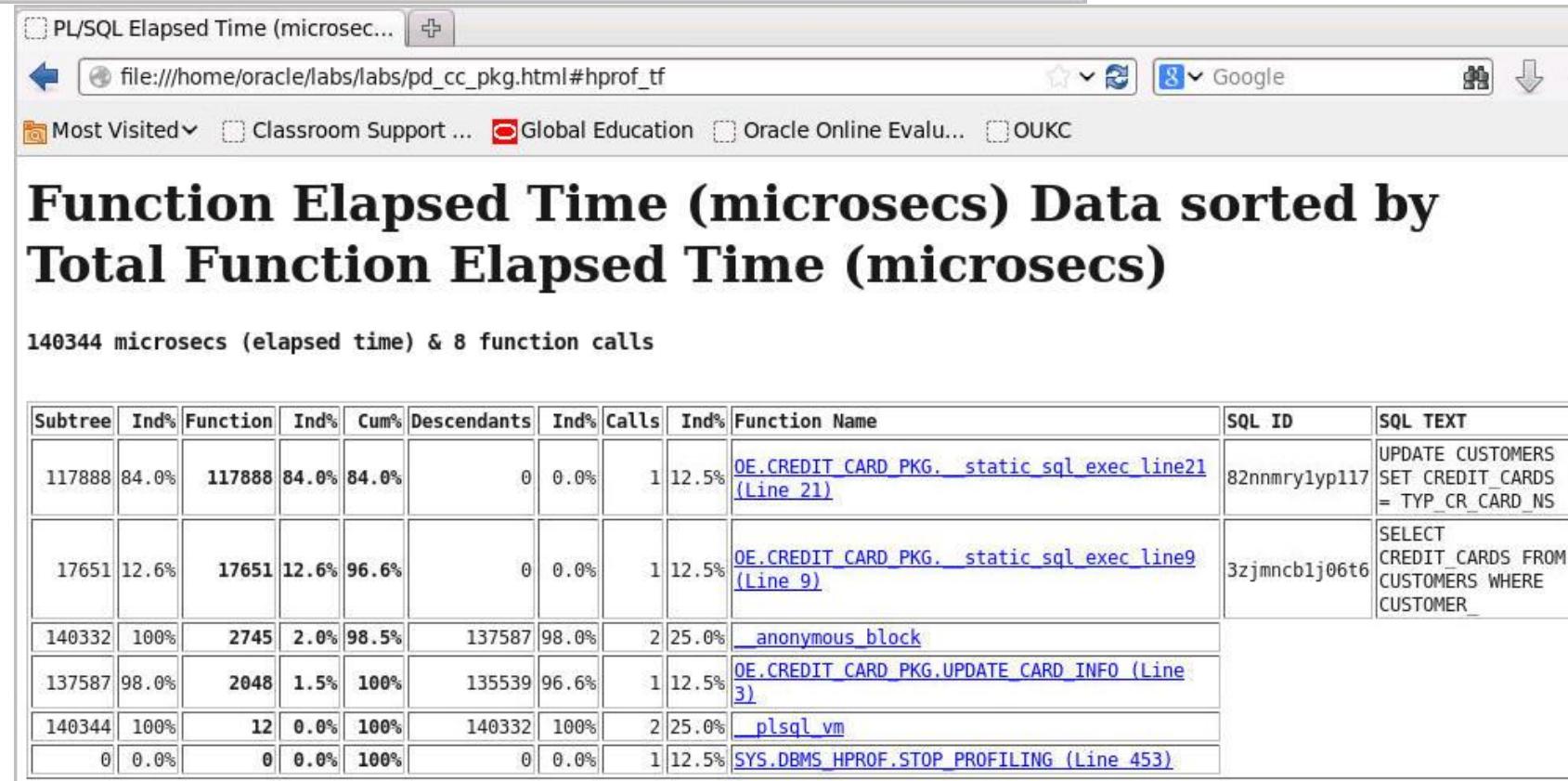
Function Elapsed Time (microsecs) Data sorted by Total Subtree Elapsed Time (microsecs)

140344 microsecs (elapsed time) & 8 function calls

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name	SQL ID	SQL TEXT
140344	100%		12	0.0%	140332	100%	2	25.0%	plsql_vm	
140332	100%		2745	2.0%	137587	98.0%	2	25.0%	anonymous_block	
137587	98.0%		2048	1.5%	135539	96.6%	1	12.5%	OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO (Line 3)	
117888	84.0%		117888	84.0%		0	0.0%	OE.CREDIT_CARD_PKG. static_sql_exec_line21 (Line 21)	82nnmry1yp117	UPDATE CUSTOMERS SET CREDIT_CARDS = TYP_CR_CARD_NS
17651	12.6%		17651	12.6%		0	0.0%	OE.CREDIT_CARD_PKG. static_sql_exec_line9 (Line 9)	3zjmncb1j06t6	SELECT CREDIT_CARDS FROM CUSTOMERS WHERE CUSTOMER_
0	0.0%		0	0.0%	0	0.0%	1	12.5%	SYS.DBMS_HPROF.STOP_PROFILING (Line 453)	

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



The screenshot shows a web browser window with the title bar "PL/SQL Elapsed Time (microsec...)" and the address bar showing "file:///home/oracle/labs/labs/pd_cc_pkg.html#hprof_tf". Below the address bar is a toolbar with icons for Back, Forward, Stop, Refresh, and Google search. The main content area has a heading "Function Elapsed Time (microsecs) Data sorted by Total Function Elapsed Time (microsecs)". Below the heading, it says "140344 microsecs (elapsed time) & 8 function calls". A table follows, showing the following data:

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)	82nnmry1yp117	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)	3zjmncb1j06t6	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 3)		
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)		

Using the HTML Reports

- [Namespace Elapsed Time \(microsecs\) Data sorted by Total Function Elapsed Time \(microsecs\)](#)
- [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](#)



PL/SQL Elapsed Time (microsec... +

file:///home/oracle/labs/labs/pd_cc_pkg.html#hprof_nsp

Most Visited Classroom Support ... Global Education Oracle Online Evalu... OUKC

Namespace Elapsed Time (microsecs) Data sorted by Namespace

140344 microsecs (elapsed time) & 8 function calls

Function	Ind%	Calls	Ind%	Namespace
4805	3.4%	6	75.0%	PLSQL
135539	96.6%	2	25.0%	SQL

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



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



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**



Summary

In this lesson, you should have learned how to:

- Trace PL/SQL program execution
- Profile PL/SQL applications

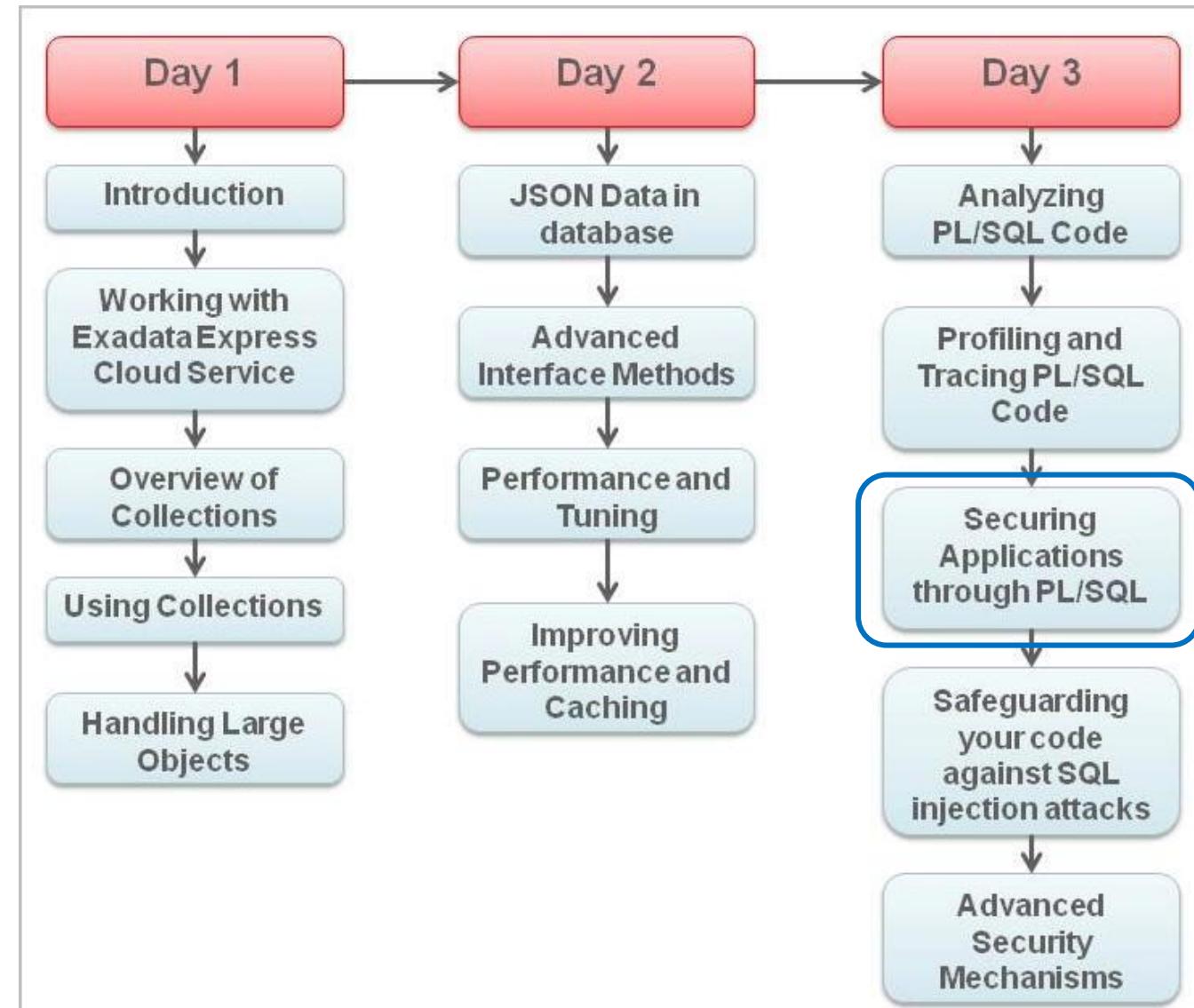


Practice 11: Overview

This practice covers hierarchical profiling of PL/SQL code.

Securing Applications through PL/SQL

Course Agenda



Objectives

After completing this lesson, you should be able to:

- Implement invoker's rights on program units
- Understand various access control mechanisms
- Define security policies on applications
- Implement Virtual Private Databases



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



Invoker's Rights and Definer's Rights



Who is an invoker?

A user trying to execute a program unit created by some other user



Who is a definer?

A user who created the program unit

What are invoker's rights?

Access privileges of a user who is invoking a program unit created by other users

What are definer's rights?

Access privileges of a user who has created the program unit

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.

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
. . .
```

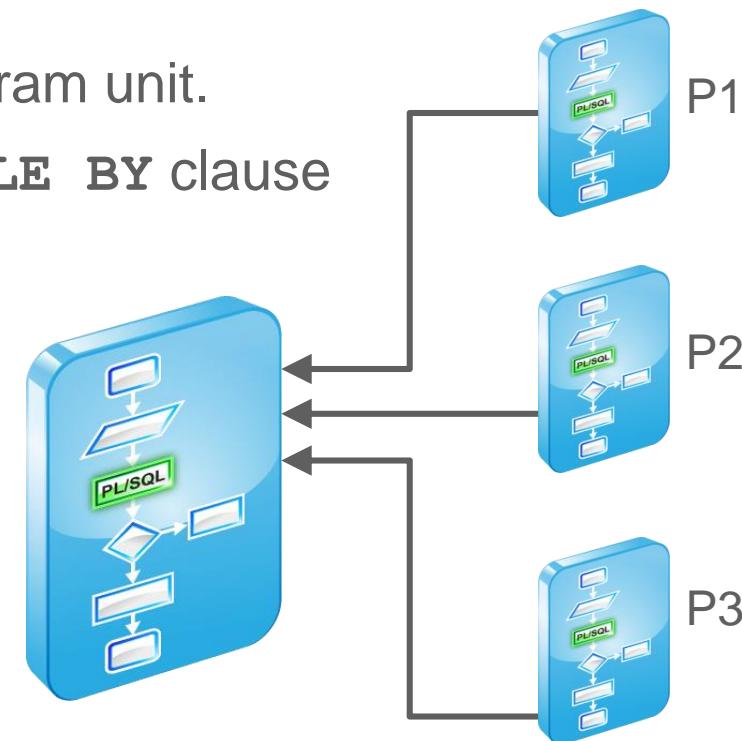
Lesson Agenda

- Invoker's rights and definer's rights on program units
- Controlling access to program units
- Application Security Policy
- Application Context
- Virtual Private Database



White Lists

- White list is a list of program units that can access a 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.



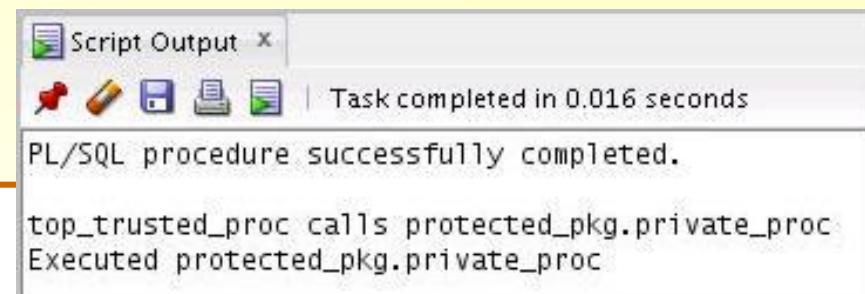
ACCESSIBLE BY Clause

- 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;
/
```



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;
```

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



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, which can allow access to the right users and deny access to intruders.



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.



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.

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 a user such as SYS.
- You should define a policy function that details the functionality of the policy.
- Define the policy using the DBMS_RLS.ADD_POLICY procedure.

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;
/
```

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;
/
```

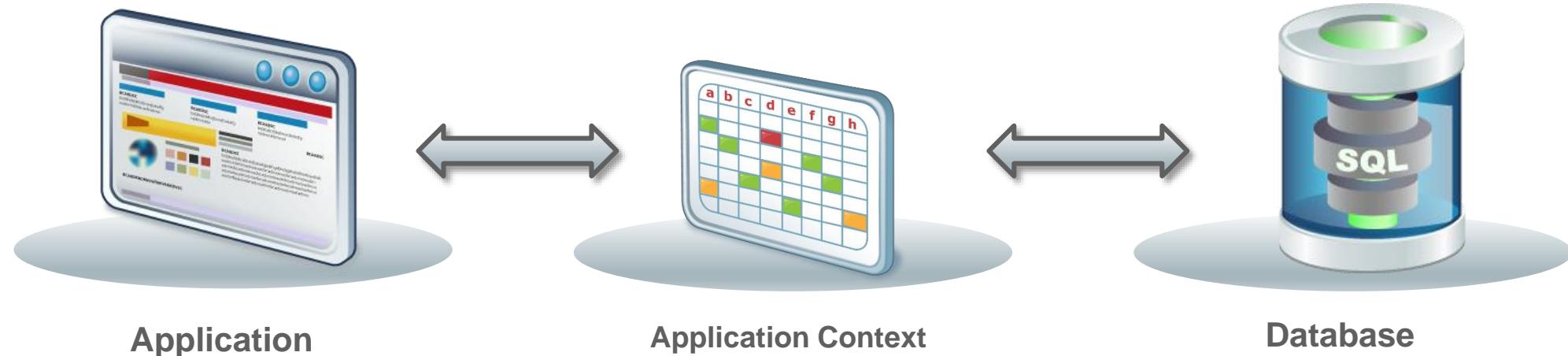
Lesson Agenda

- Invoker's rights and definer's rights on program units
- Controlling access to program units
- Application Security Policy
- Application Context
- Virtual Private Database



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 that holds the `customer_id` and use it while retrieving data from the database during the user session.



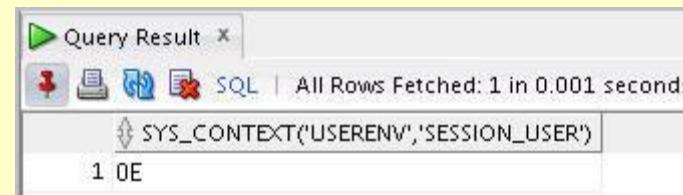
Application Context - Implementation

- 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.

USERENV Application Context

- USERENV is a namespace that holds the data of the current session.
- USERENV holds the context of the current session.

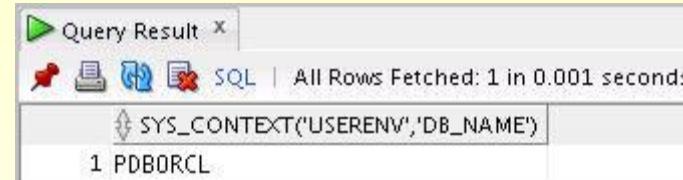
```
SELECT SYS_CONTEXT ('USERENV', 'SESSION_USER')
FROM DUAL;
```



A screenshot of the Oracle SQL Developer interface showing a 'Query Result' window. The window title is 'Query Result'. It contains a single row of data with one column labeled 'SYS_CONTEXT('USERENV','SESSION_USER')' and the value '1 OE'. Below the table, it says 'All Rows Fetched: 1 in 0.001 seconds'.

SYS_CONTEXT('USERENV','SESSION_USER')
1 OE

```
SELECT SYS_CONTEXT ('USERENV', 'DB_NAME')
FROM DUAL;
```



A screenshot of the Oracle SQL Developer interface showing a 'Query Result' window. The window title is 'Query Result'. It contains a single row of data with one column labeled 'SYS_CONTEXT('USERENV','DB_NAME')' and the value '1 PDBORCL'. Below the table, it says 'All Rows Fetched: 1 in 0.001 seconds'.

SYS_CONTEXT('USERENV','DB_NAME')
1 PDBORCL

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;
```

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)
...

```

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



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 a user id and password credentials to log in to the database and have a look at your orders.
- You can view only your orders when you log in to 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**.



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.



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



Setting Up a Context

Set up a driving context.

```
CREATE OR REPLACE CONTEXT order_ctx  
  USING orders_app_pkg;
```

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
/
```

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' ,
```

```
        'CUSTOMERS' ,
```

```
        'OE_ACCESS_POLICY' ,
```

```
        'SYS' ,
```

```
        'ORDERS_APP_PKG.THE_PREDICATE' ,
```

```
        'SELECT, UPDATE, DELETE' ,
```

```
        FALSE ,
```

```
        TRUE) ;
```

```
END ;
```

```
/
```

Object schema

Table name

Policy name

Function schema

Policy function

Statement types

Update check

Enabled

Setting Up the Logon Trigger

Create a database trigger that executes whenever anyone logs in to the database:

```
CREATE OR REPLACE TRIGGER set_id_on_logon
AFTER logon on DATABASE
BEGIN
    oe.orders_app_pkg.set_app_context;
END ;
/
```

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
```

Data Dictionary Views

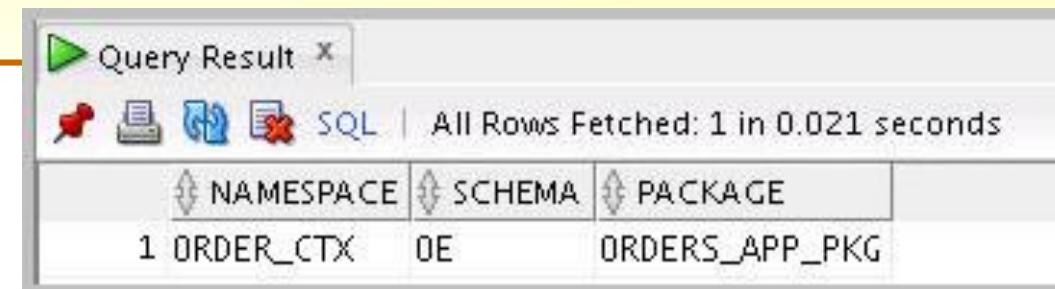
View	Description
USER_POLICIES	All policies owned by the current schema
ALL_POLICIES	All policies owned or accessible by the current schema
DBA_POLICIES	All policies in the database (its columns are the same as those in ALL_POLICIES.)
ALL_CONTEXT	All active context namespaces defined in the session
DBA_CONTEXT	All context namespace information (active and inactive)



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;
```



The screenshot shows a 'Query Result' window from Oracle SQL Developer. The window title is 'Query Result'. Below the title, there are icons for Run, Stop, Refresh, and SQL. The status bar at the bottom of the window says 'All Rows Fetched: 1 in 0.021 seconds'. The main area displays a table with three columns: 'NAMESPACE', 'SCHEMA', and 'PACKAGE'. There is one row of data: 'ORDER_CTX' in the NAMESPACE column, 'OE' in the SCHEMA column, and 'ORDERS_APP_PKG' in the PACKAGE column.

NAMESPACE	SCHEMA	PACKAGE
ORDER_CTX	OE	ORDERS_APP_PKG

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;
```



The screenshot shows a 'Query Result' window from Oracle SQL Developer. The window title is 'Query Result'. It contains a toolbar with icons for Run, Stop, Refresh, and Save, followed by a status message 'All Rows Fetched: 1 in 2.416 seconds'. Below the toolbar is a grid of data. The columns are labeled: OBJECT_NAME, POLICY_NAME, PF_OWNER, PACKAGE, FUNCTION, SEL, INS, UPD, and DEL. There is one row of data: CUSTOMERS, OE_ACCESS_POLICY, SYS, ORDERS_APP_PKG, THE_PREDICATE, YES, NO, YES, YES.

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

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 that can be used to create policy groups.
- `SYS_DEFAULT` is the default policy group:

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.



Quiz



Application context for a user is fixed and does not change with change of session.

- a. True
- b. False



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



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



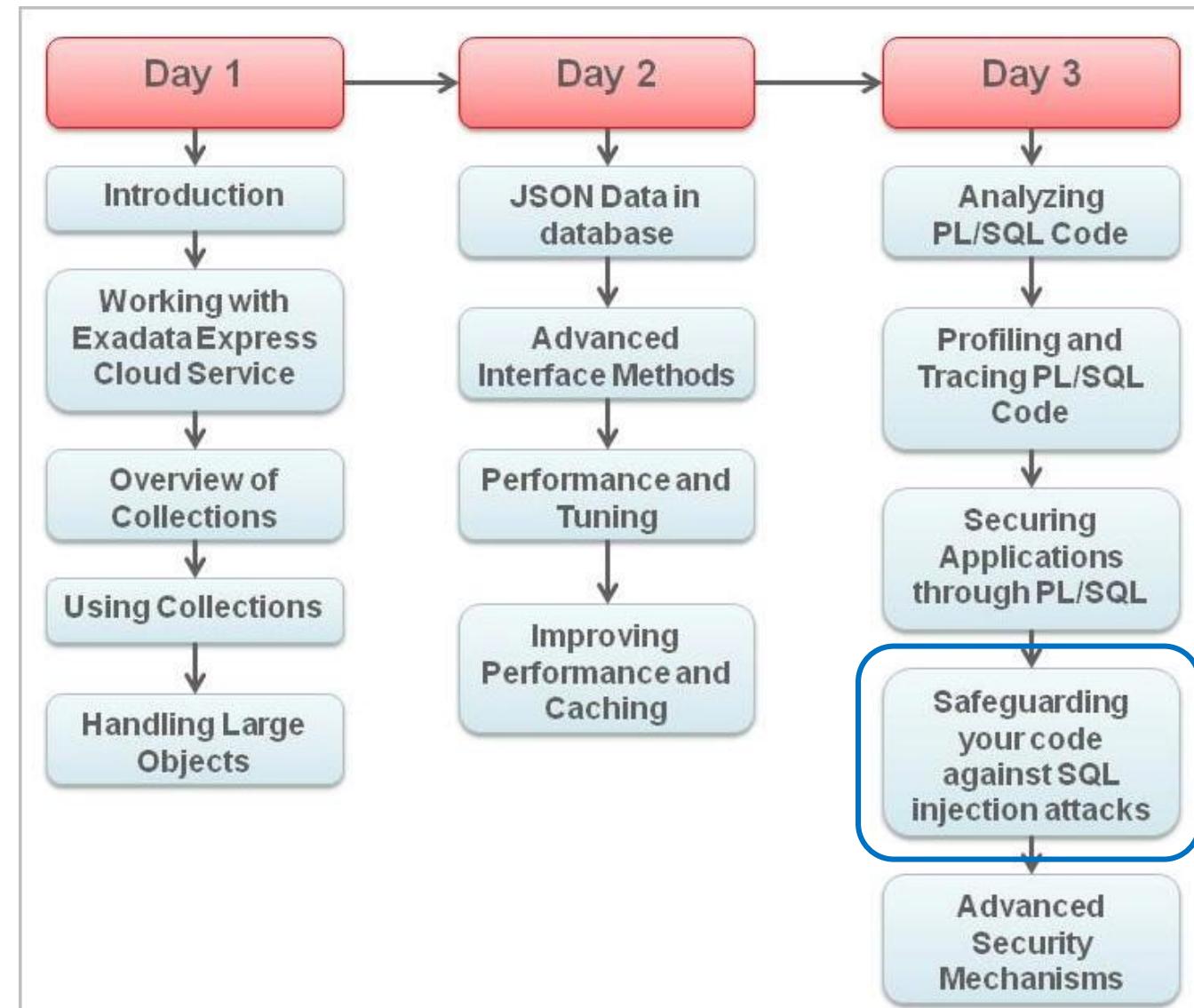
Practice 12: Overview

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

Safeguarding Your Code against SQL Injection Attacks

Course Agenda



Objectives

After completing this lesson, you should be able to:

- Describe SQL injection
- Reduce attack surfaces
- Use DBMS_ASSERT



Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Static SQL vs Dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT



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.
- It 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 the 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.

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



Scenario

```
EXECUTE get_mail('Andrews')
```



```
EXECUTE get_email('x' union  
select username from  
all_users where  
'x'='x')
```



PL/SQL procedure successfully completed.

SQL statement: SELECT cust_email FROM customers
WHERE cust_last_name = 'Andrews'
Email: Ajay.Andrews@YELLOWTHROAT.EXAMPLE.COM
Email: Dianne.Andrews@TURNSTONE.EXAMPLE.COM

PL/SQL procedure successfully completed.

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

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.

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.

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.

Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Static SQL vs Dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT



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.



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`.

Using Invoker's Rights

- Using invoker's rights helps 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

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

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.



Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Static SQL Vs Dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT



Using Static SQL

- Eliminates SQL injection vulnerability
- Creates schema object dependencies upon successful compilation
- Can improve performance, when compared with DBMS_SQL

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.



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;
```

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.

Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Static SQL vs Dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT



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;
```

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;
```

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.

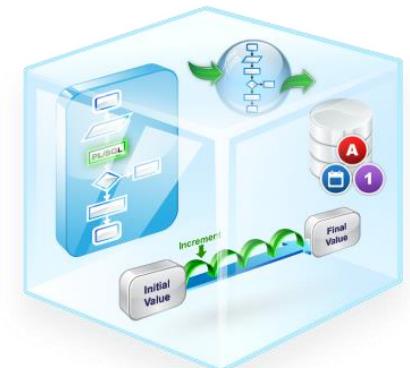
Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Static SQL vs Dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT



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.



Understanding DBMS_ASSERT

DBMS_ASSERT functions:

Function	Description
ENQUOTE_LITERAL	Encloses string literal in single quotes
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	Verifies that the 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

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' ;
```

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.

```
SELECT count(*) records FROM "orders";
```

- Such identifiers are vulnerable to SQL injection.

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

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`.

Validate Input Using DBMS_ASSERT

- Use DBMS_ASSERT package, which 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.

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.

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,'''','''')||');'||...
```

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.



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;
```

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



Quiz



By default, a stored procedure executes with the privileges of its owner (definer's rights).

- a. True
- b. False



Quiz



If you must use dynamic SQL, avoid using input concatenation to build the dynamic SQL.

- a. True
- b. False



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



Summary

In this lesson, you should have learned how to:

- Detect SQL injection vulnerabilities
- Reduce attack surfaces
- Use DBMS_ASSERT



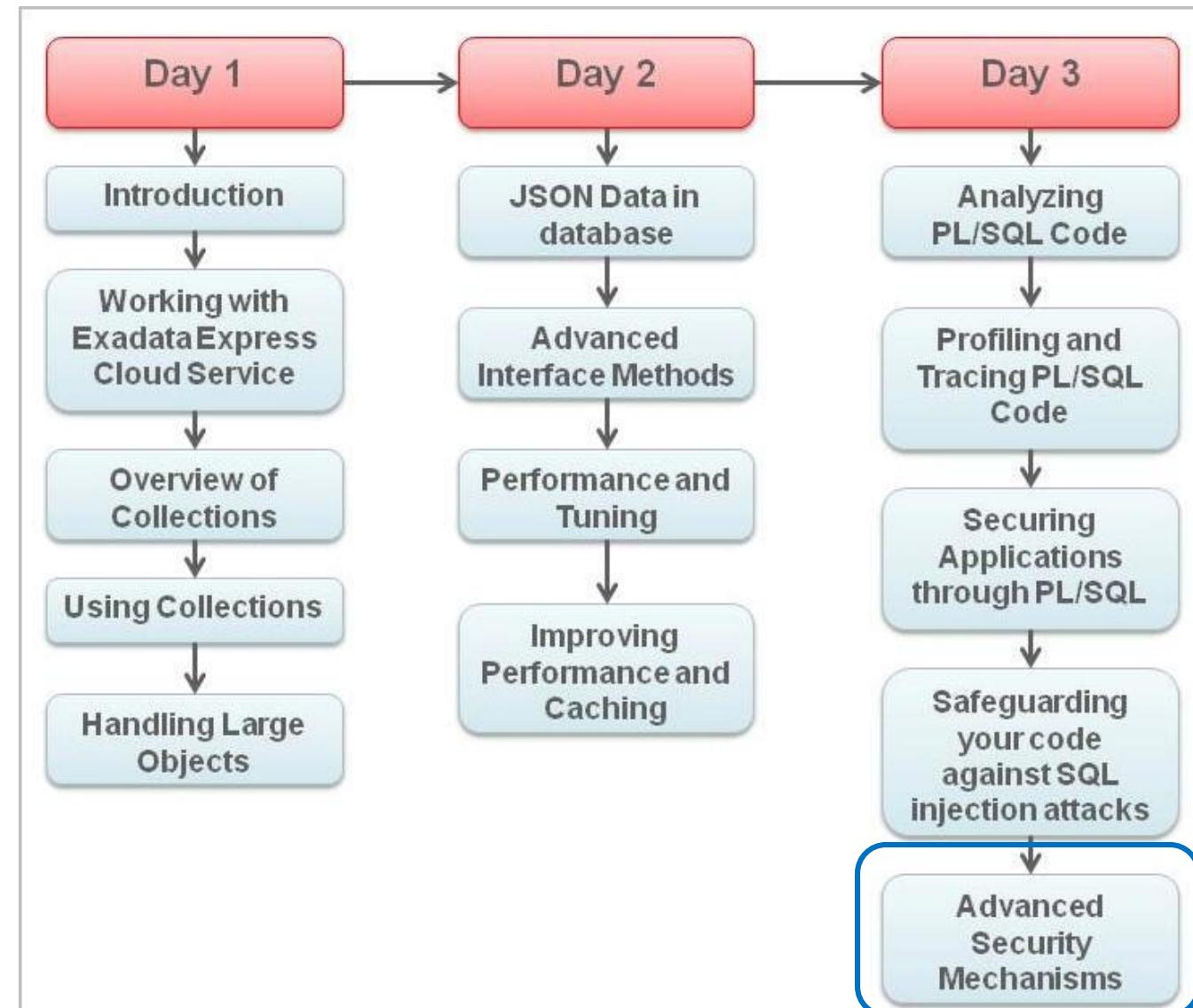
Practice 13: Overview

This practice covers the following topics:

- Testing your knowledge of SQL injection
- Rewriting code to protect against SQL injection

Advanced Security Mechanisms

Course Agenda



Objectives

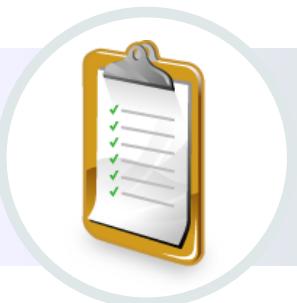
After completing this lesson, you should be able to understand:

- Real Application Security
- Transparent Data Encryption mechanism
- Oracle Data Redaction mechanism



Lesson Agenda

- Real Application Security
- Transparent Data Encryption
- Data Redaction



Real Application Security

- Real Application Security is implemented to ensure security in 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.

How It Works Without RAS?

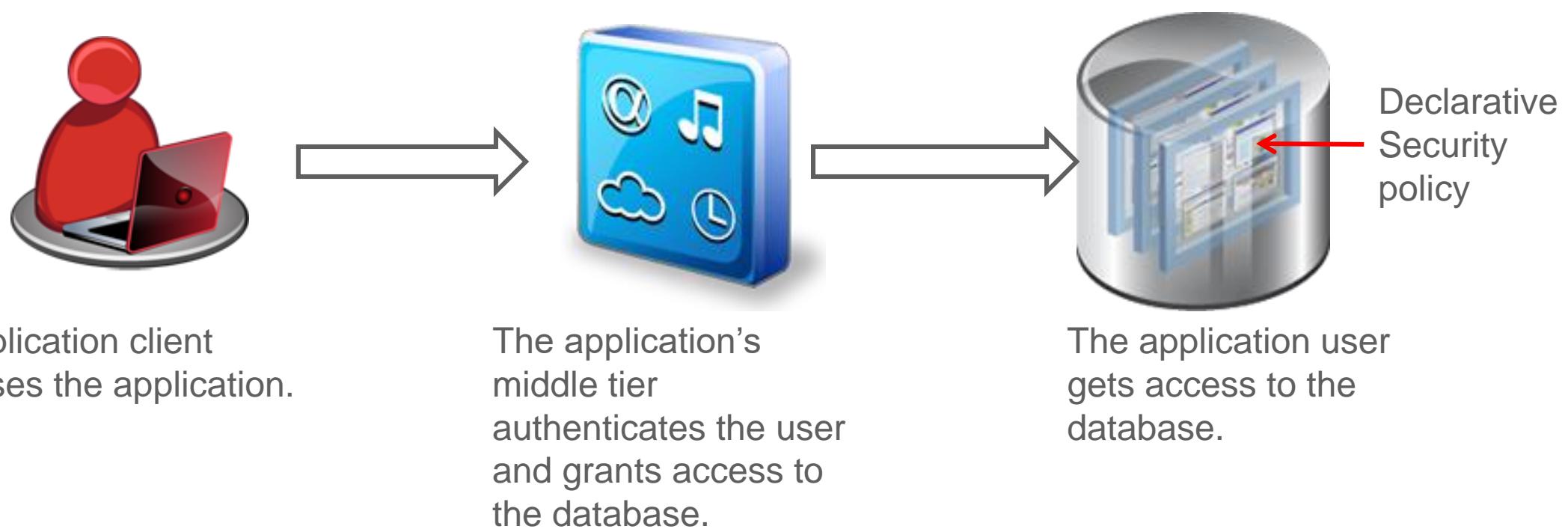


An application client
accesses the application.

The application's
middle tier
authenticates the user
and grants access to
the database.

The application user
gets access to the
database.

How It Works with RAS?

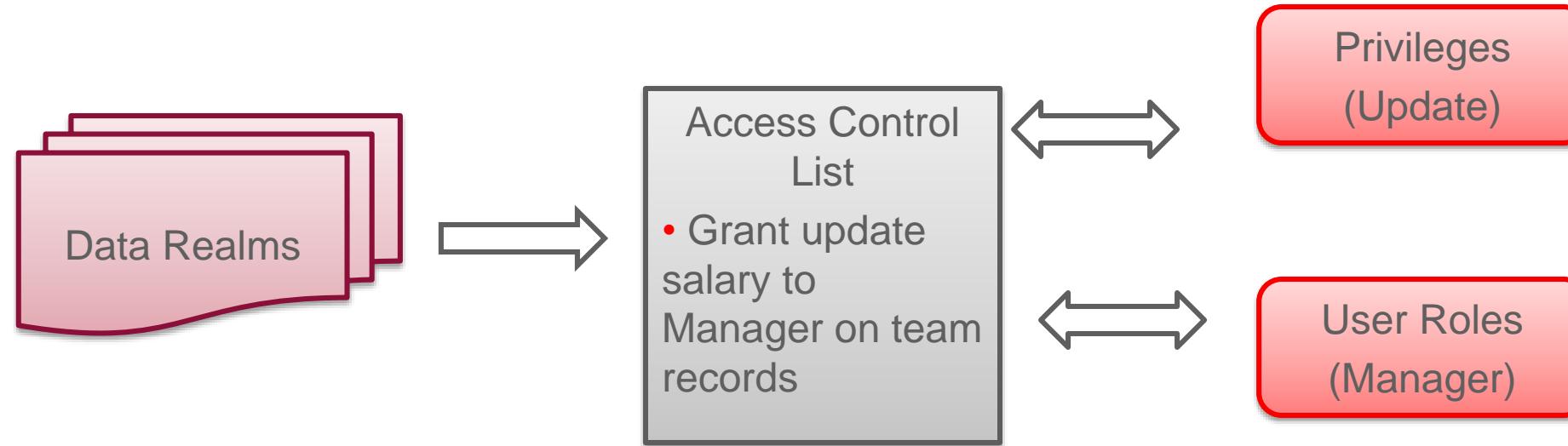


Real Application Security - Components

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

Implementing a RAS Data Security Policy

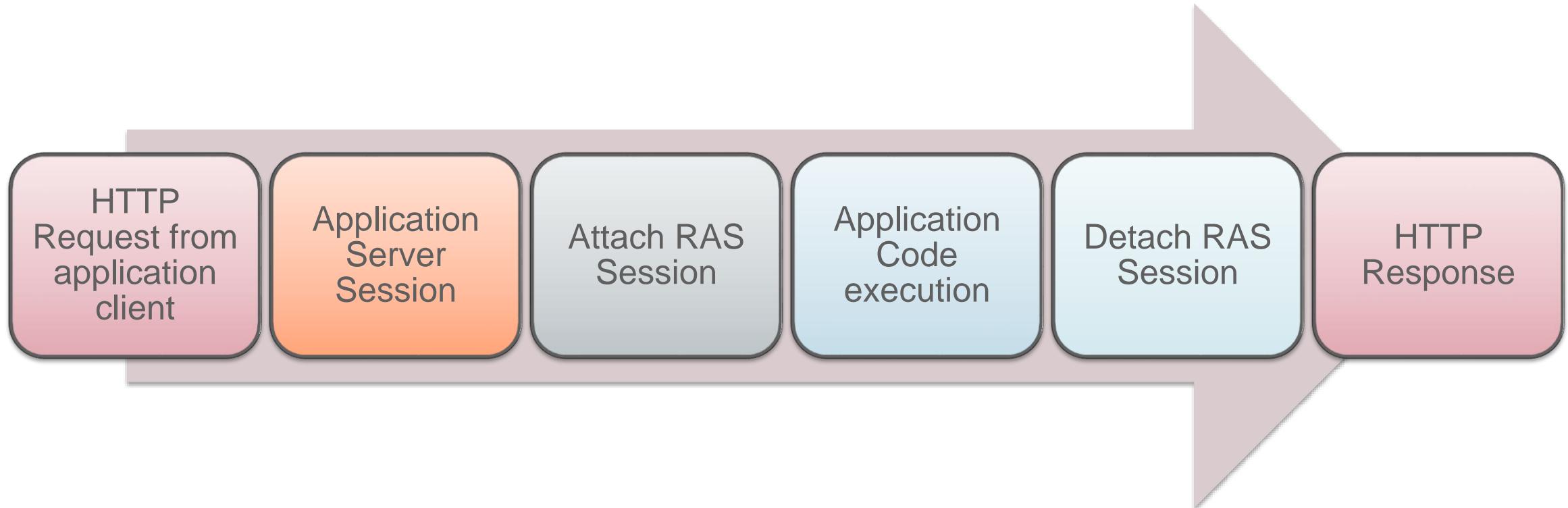


Application Sessions in RAS

- 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.



RAS Sessions



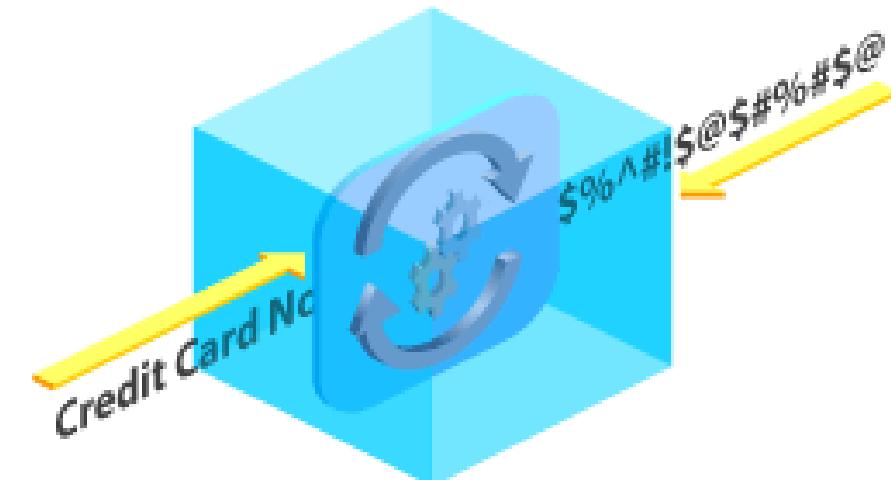
Lesson Agenda

- Real Application Security
- Transparent Data Encryption
- Data Redaction



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.



Encrypting a Table Column Using TDE

- 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.

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.
- TDE tablespace encryption allows index range scans on data in encrypted tablespaces unlike TDE column encryption.

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.

Lesson Agenda

- Real Application Security
- Transparent Data Encryption
- Data Redaction



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.

Data Redaction Methods

You can implement data redaction using one of the following methods:

- Full redaction
- Partial redaction
- Regular expressions
- Random redaction
- No redaction

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.

Summary

In this lesson, you should have learned how:

- 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



A

Table Descriptions and Data

B

Using SQL Developer

Objectives

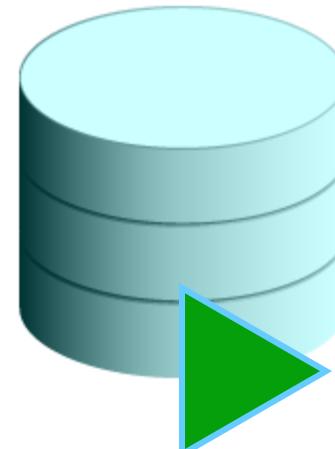
After completing this appendix, you should be able to do the following:

- List the key features of Oracle SQL Developer
- Identify the menu items of Oracle SQL Developer
- Create a database connection
- Manage database objects
- Use SQL Worksheet
- Save and run SQL scripts
- Create and save reports
- Browse the Data Modeling options in SQL Developer



What Is Oracle SQL Developer?

- Oracle SQL Developer is a graphical tool that enhances productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema by using standard Oracle database authentication.

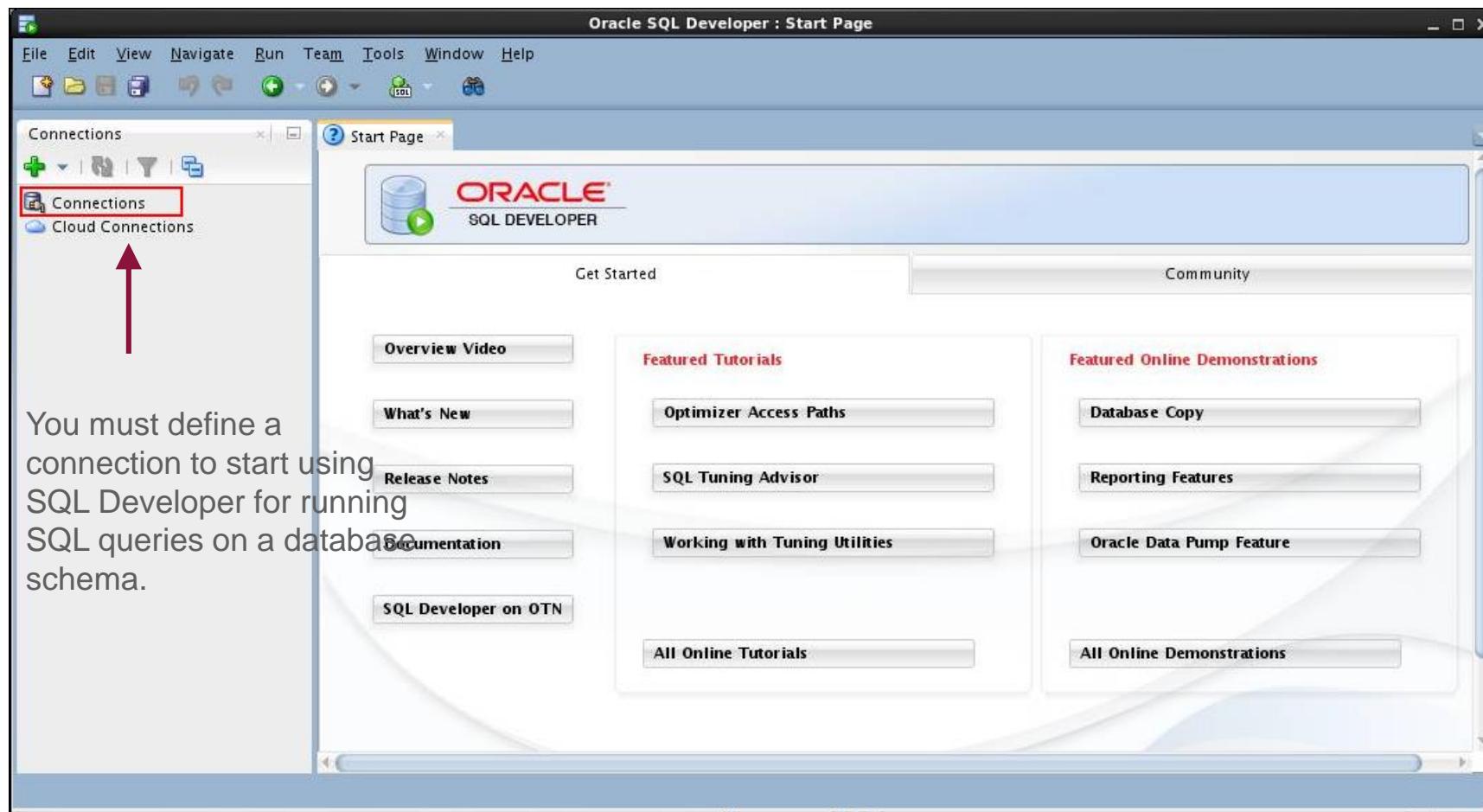


SQL Developer

Specifications of SQL Developer

- Is not shipped with Oracle Database Release 19c
- Is developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Enables default connectivity using the JDBC Thin driver
- Connects to Oracle Database 9*i* and later

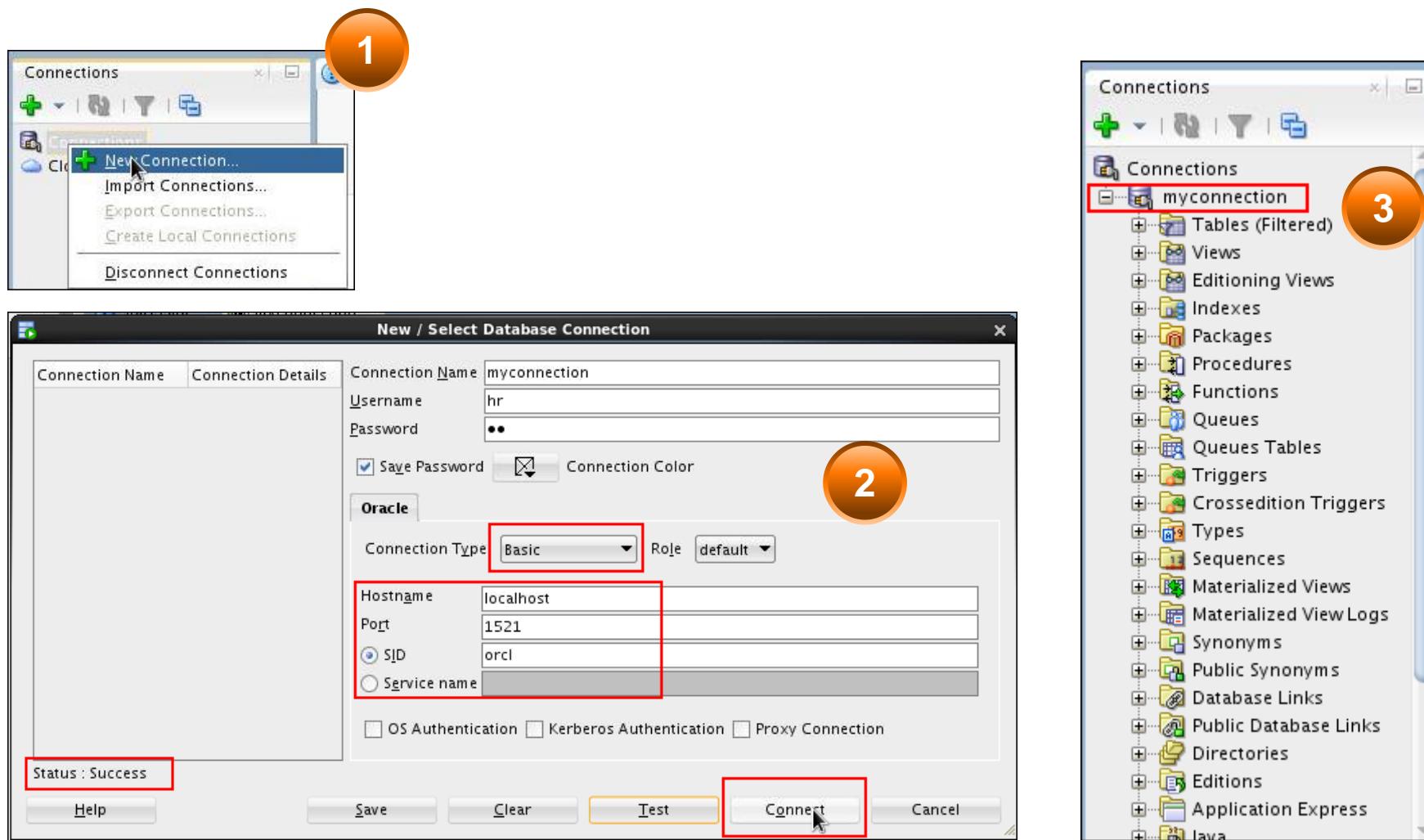
SQL Developer Interface



Creating a Database Connection

- You must have at least one database connection to use SQL Developer.
- You can create and test connections for multiple:
 - Databases
 - Schemas
- SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an Extensible Markup Language (XML) file.
- Each additional database connection created is listed in the Connections Navigator hierarchy.

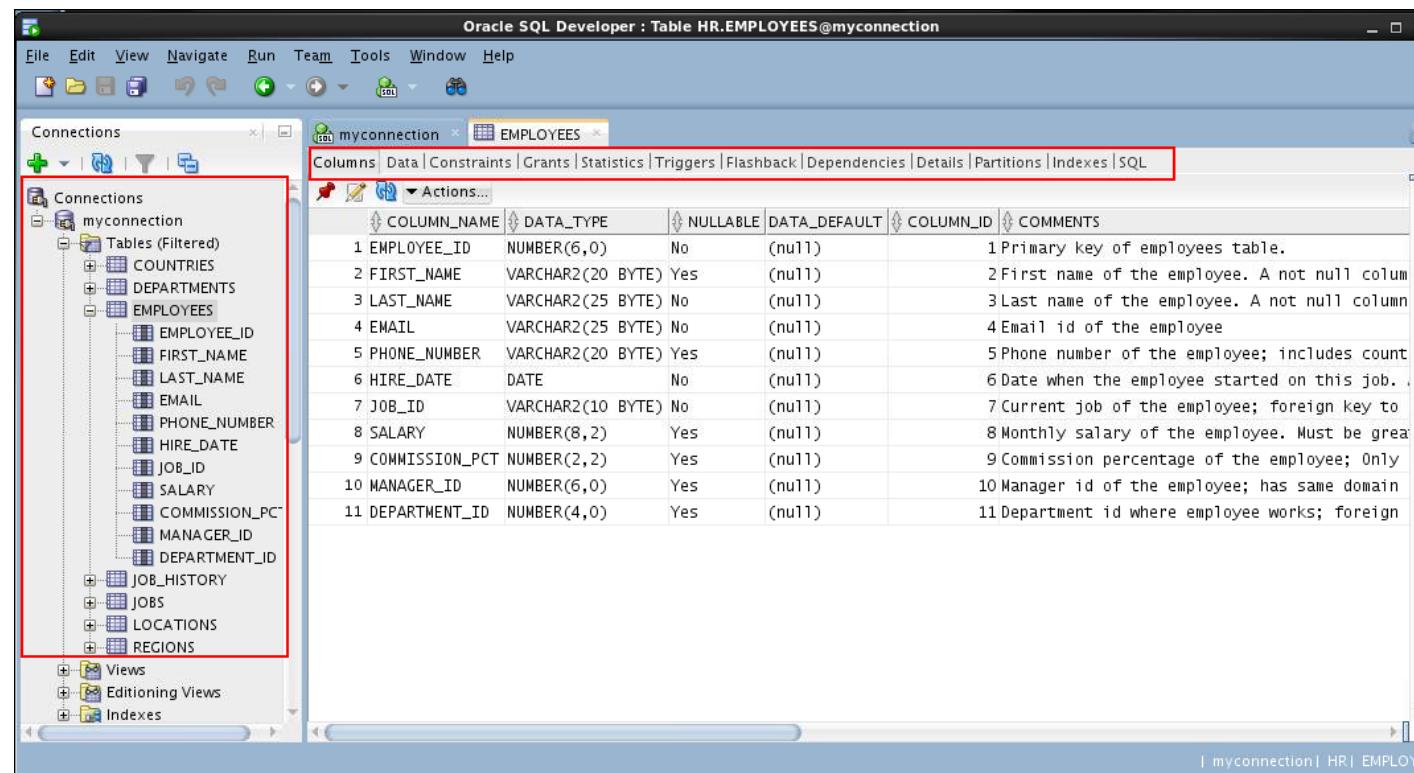
Creating a Database Connection



Browsing Database Objects

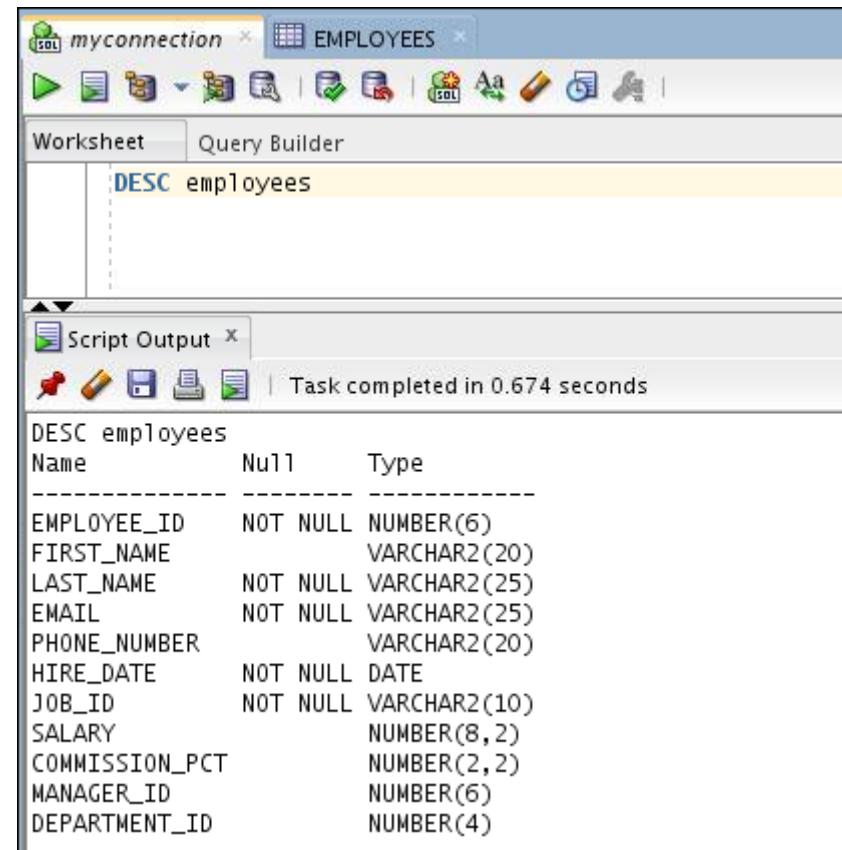
Use the Connections Navigator to:

- Browse through many objects in a database schema
- Review the definitions of objects at a glance



Displaying the Table Structure

Use the DESCRIBE command to display the structure of a table:

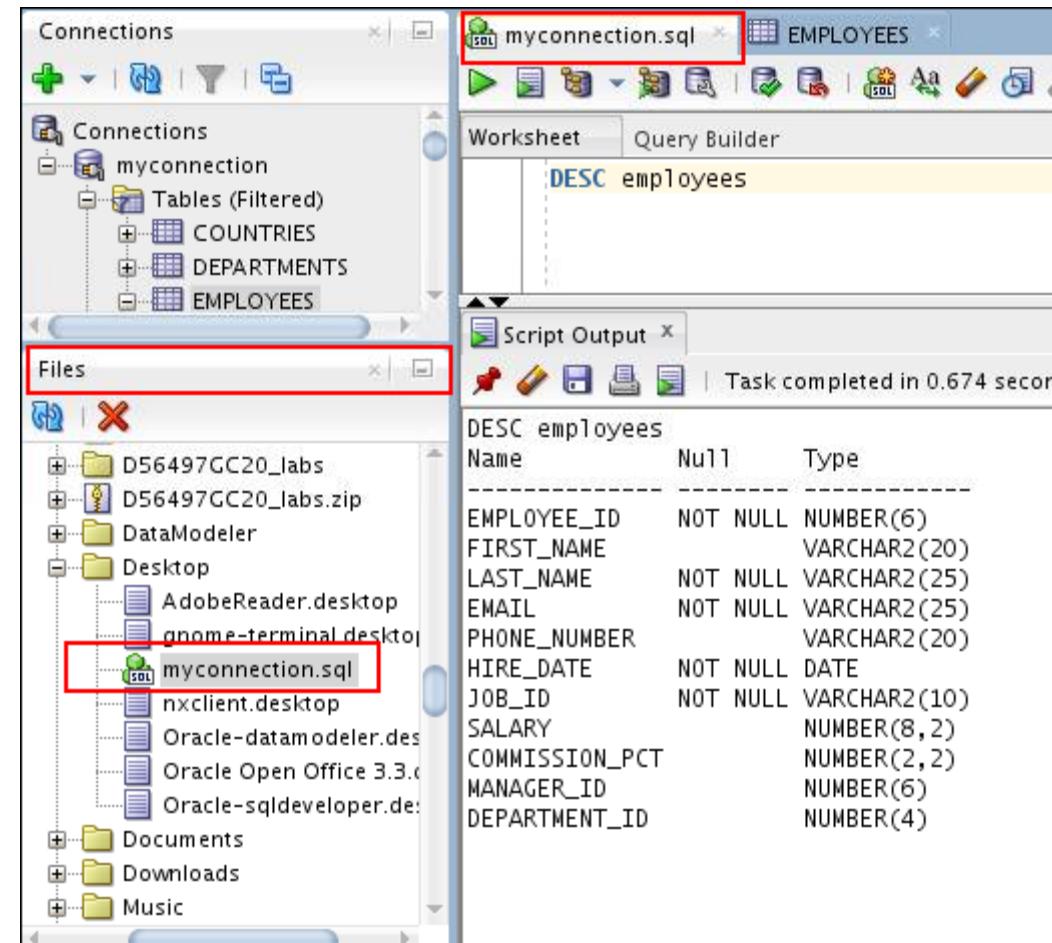


The screenshot shows the Oracle SQL Developer interface. The title bar says "myconnection > EMPLOYEES". The main area has tabs for "Worksheet" and "Query Builder", with "Worksheet" selected. Below the tabs, the text "DESC employees" is entered. A "Script Output" window below shows the results of the DESCRIBE command. The output starts with a header row "DESC employees" followed by a line separator and then a table of column details. The columns listed are: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, and DEPARTMENT_ID. Each column is shown with its name, whether it can be null ("Null"), and its data type.

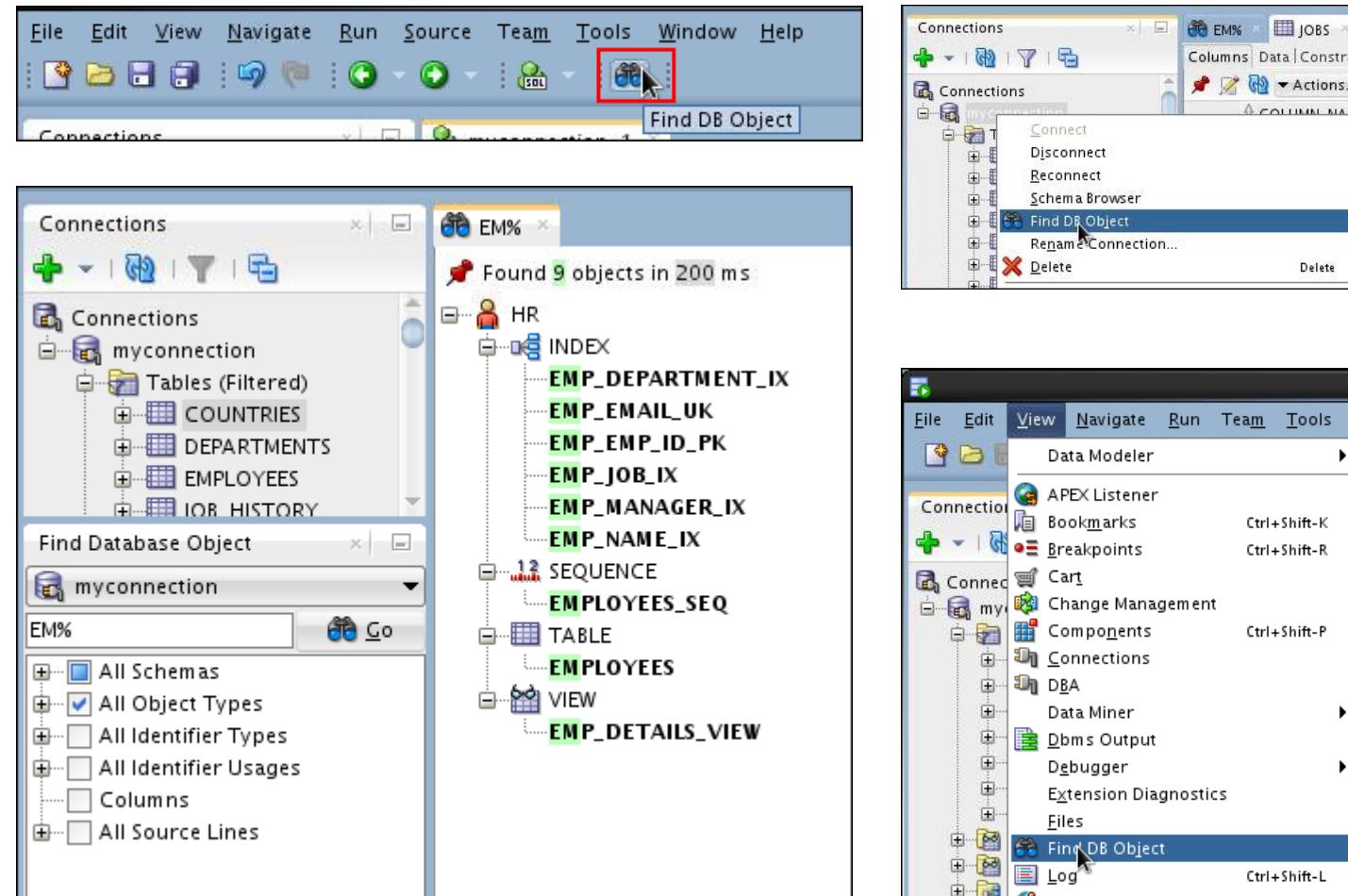
Name	Null	Type
EMPLOYEE_ID	NOT NULL	NUMBER(6)
FIRST_NAME		VARCHAR2(20)
LAST_NAME	NOT NULL	VARCHAR2(25)
EMAIL	NOT NULL	VARCHAR2(25)
PHONE_NUMBER		VARCHAR2(20)
HIRE_DATE	NOT NULL	DATE
JOB_ID	NOT NULL	VARCHAR2(10)
SALARY		NUMBER(8,2)
COMMISSION_PCT		NUMBER(2,2)
MANAGER_ID		NUMBER(6)
DEPARTMENT_ID		NUMBER(4)

Browsing Files

Use the File Navigator to explore the file system and open system files.

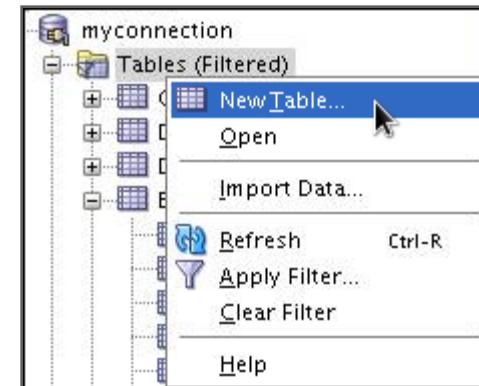


Finding Database Objects

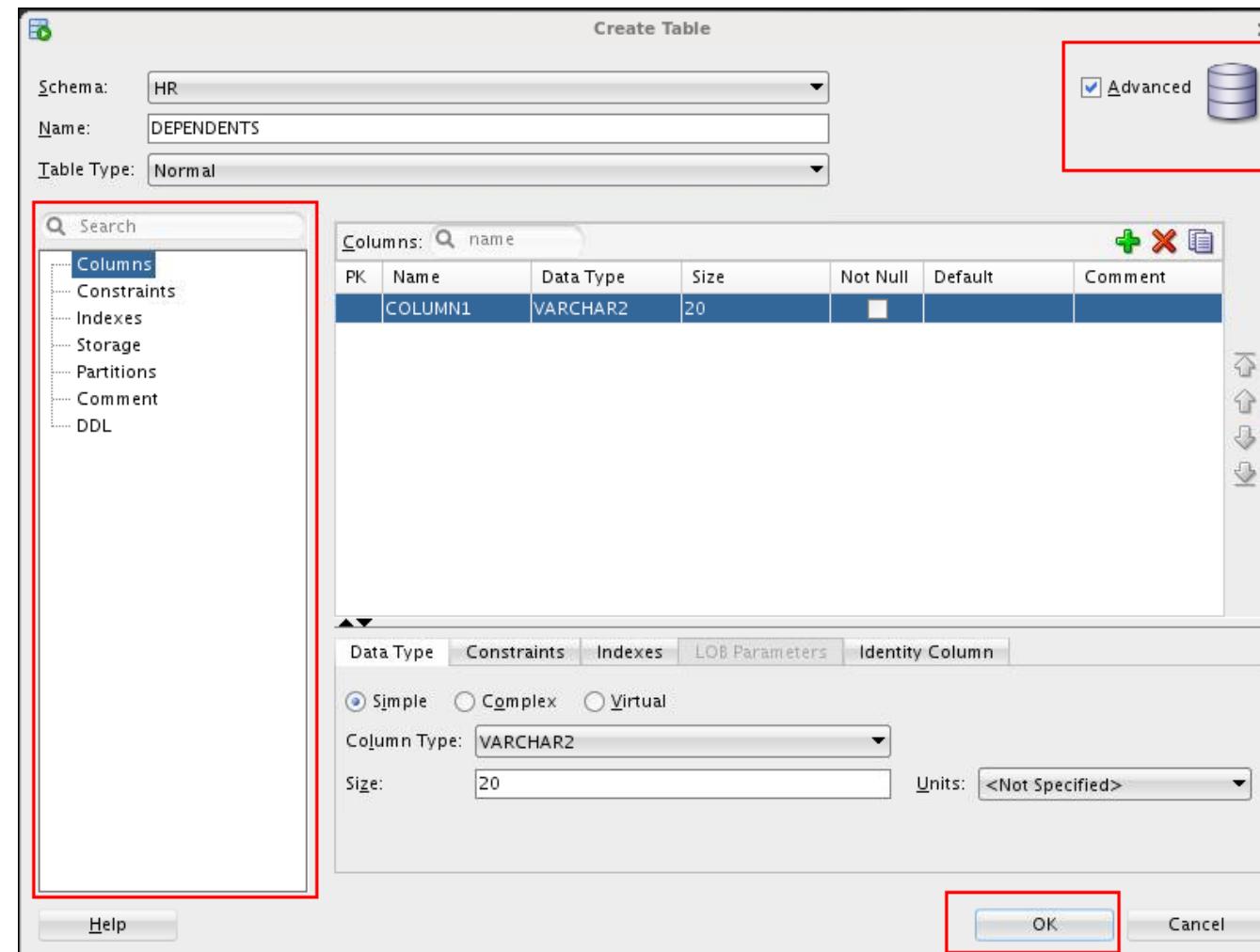


Creating a Schema Object

- SQL Developer supports the creation of any schema object by:
 - Executing a SQL statement in SQL Worksheet
 - Using the context menu
- Edit the objects by using an edit dialog box or one of the many context-sensitive menus.
- View the data definition language (DDL) for adjustments such as creating a new object or editing an existing schema object.

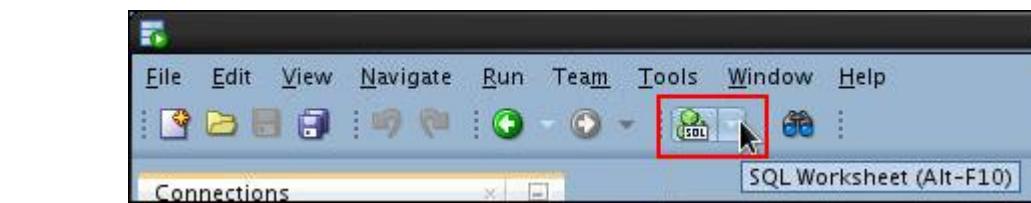
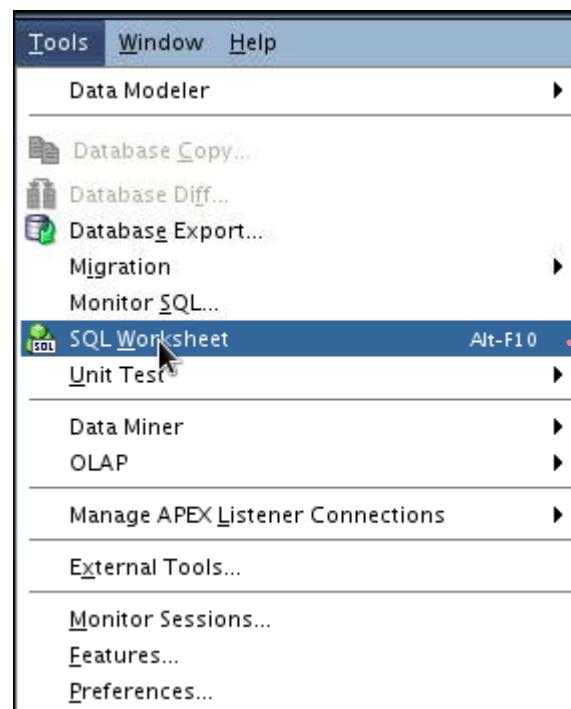


Creating a New Table: Example



Using the SQL Worksheet

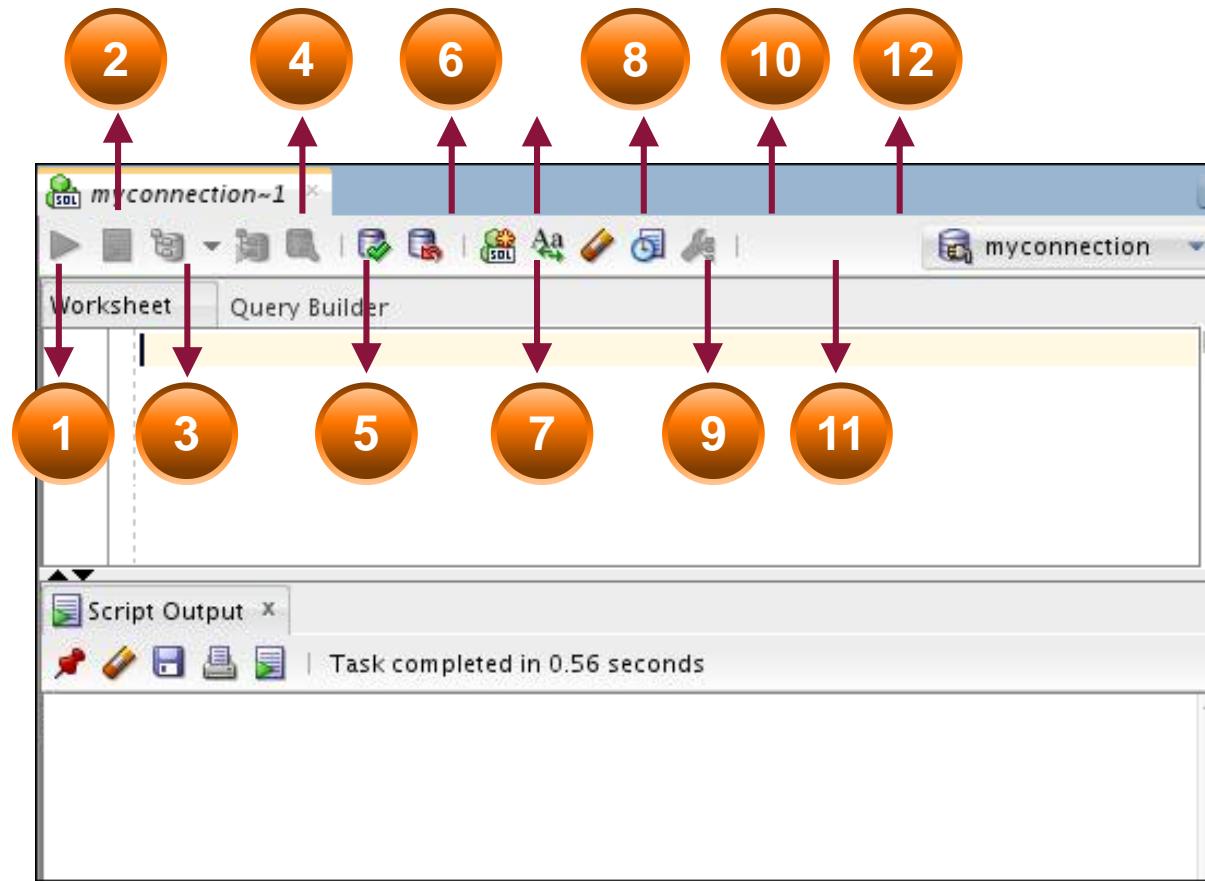
- Use the SQL Worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements.
- Specify any actions that can be processed by the database connection associated with the worksheet.



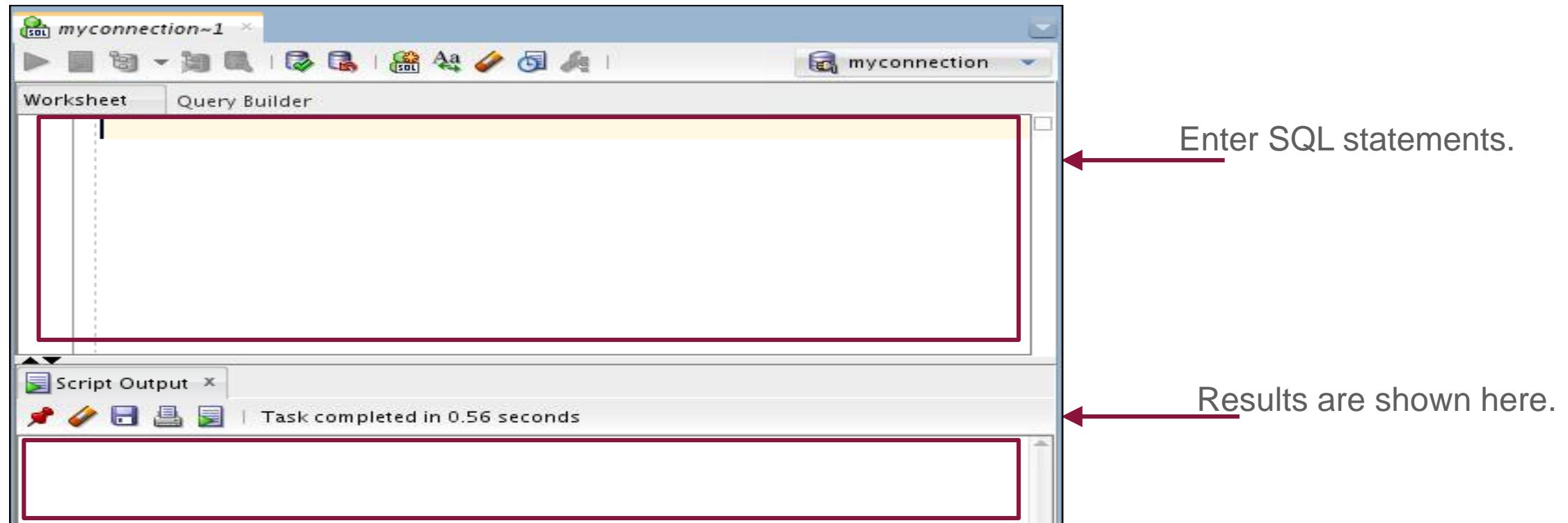
Select SQL Worksheet
from the Tools menu.

Or, click the Open SQL
Worksheet icon.

Using the SQL Worksheet

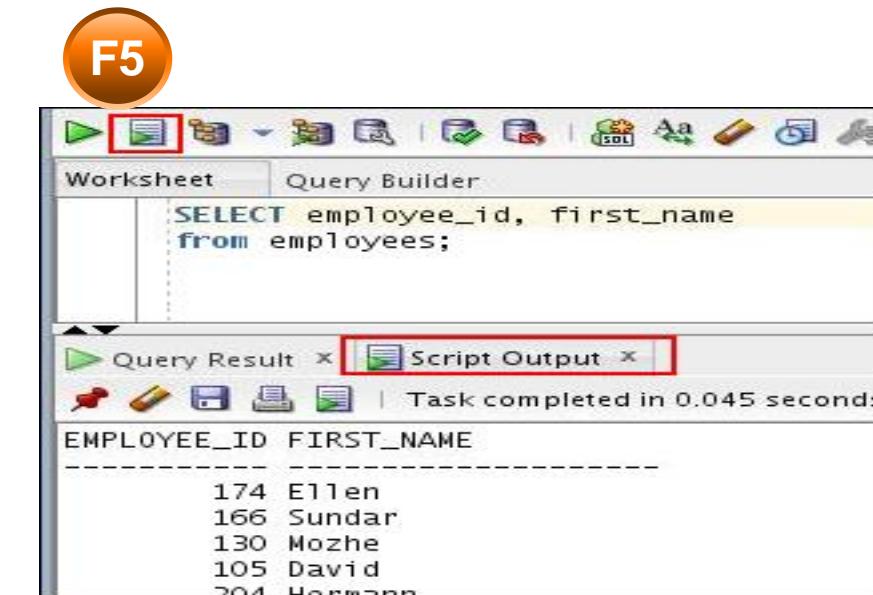
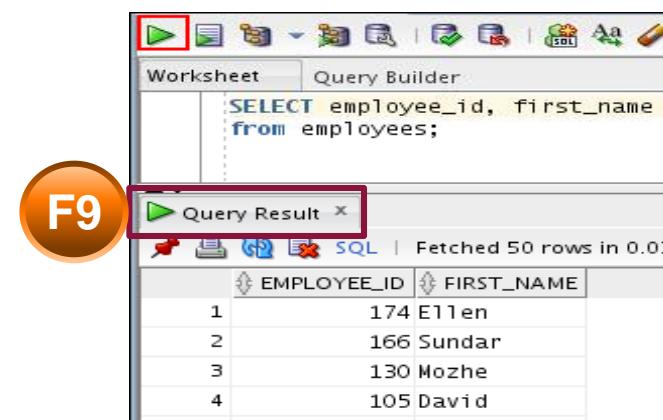
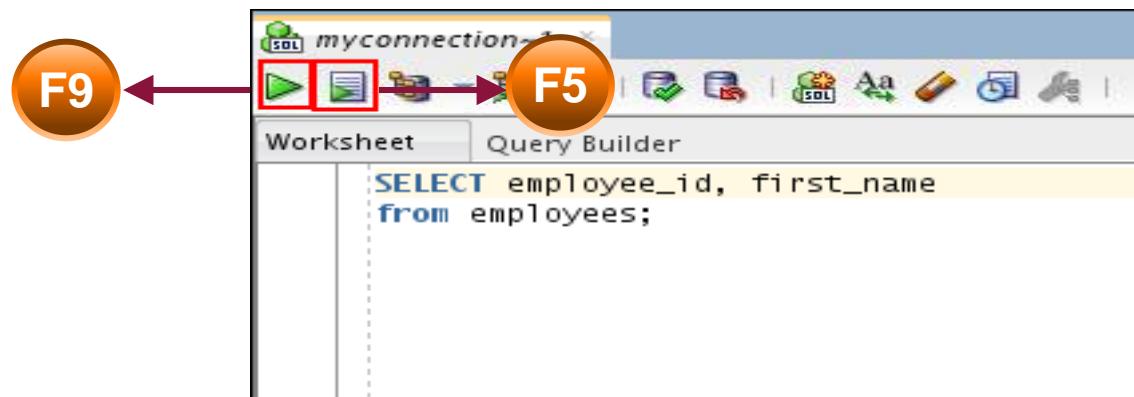


Using the SQL Worksheet



Executing SQL Statements

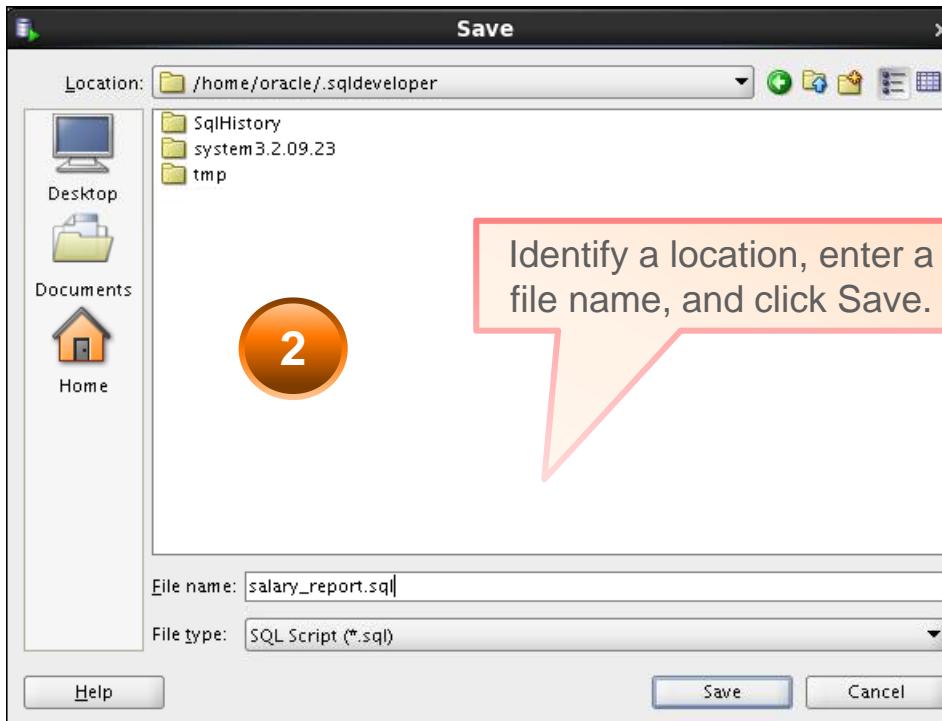
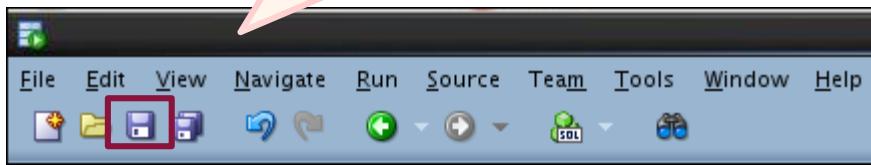
Use the Enter SQL Statement box to enter single or multiple SQL statements.



Saving SQL Scripts

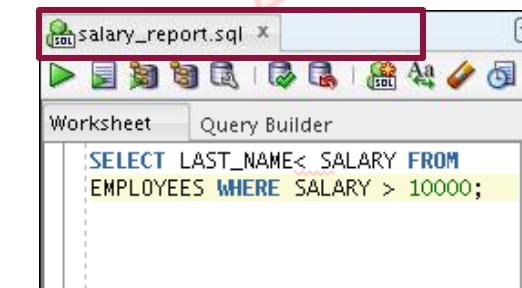
1

Click the Save icon to save your SQL statement to a file.

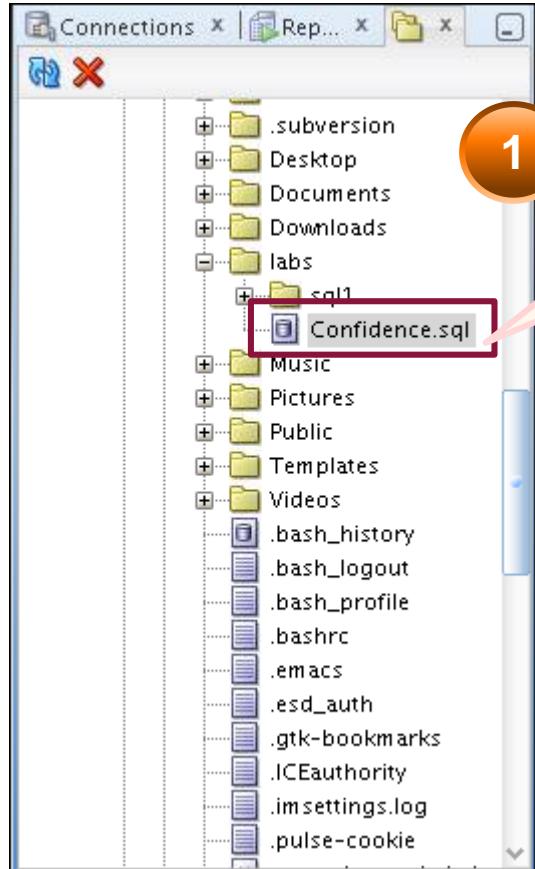


3

The contents of the saved file are visible and editable in your SQL Worksheet window.



Executing Saved Script Files: Method 1



- Use the Files tab to locate the script file that you want to open.
- Double-click the script to display the code in the SQL Worksheet.

- To run the code, click either:
- Execute Script (F9), or
 - Run Script (F5)

The screenshot shows the 'SQL Worksheet' window. The title bar says 'Confidence.sql x'. Below it, there are tabs for 'Worksheet' and 'Query Builder', with 'Worksheet' selected. The main area contains the following SQL code:

```
SELECT count(*) FROM tab;
SELECT count(*) FROM employees;
SELECT count(*) FROM countries;
SELECT count(*) FROM regions;
```

The toolbar above the worksheet has several icons. The first three icons (Execute, Run, and Stop) are highlighted with a red rectangle and circled with a large orange circle labeled '3'.

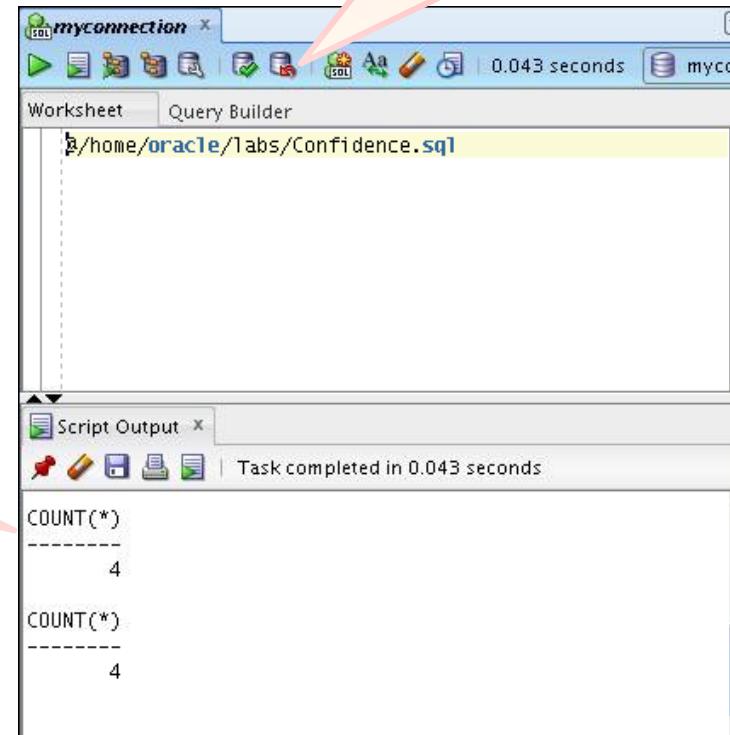
2

- Select a connection from the drop-down list.

3

Executing Saved Script Files: Method 2

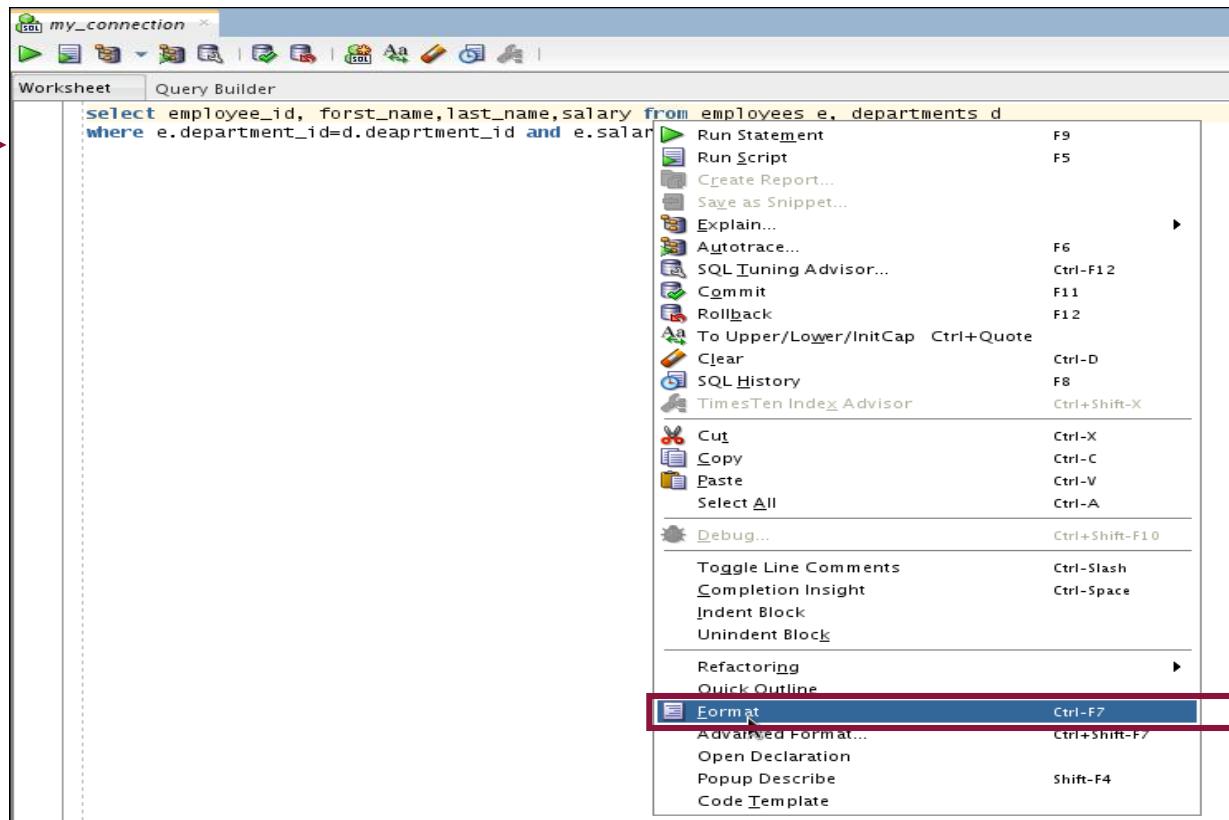
Use the @ command followed by the location and name of the file that you want to execute, and click the Run Script icon.



The output from the script is displayed on the Script Output tabbed page.

Formatting the SQL Code

Before formatting

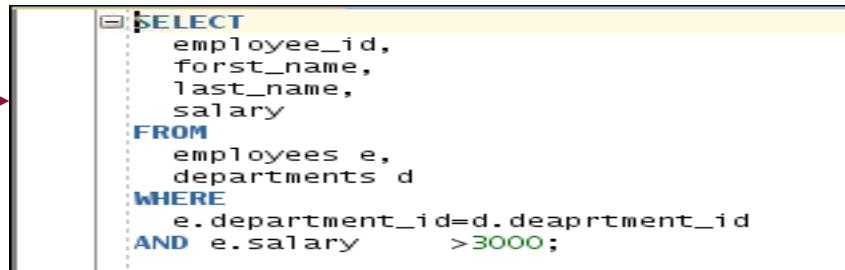


The screenshot shows the Oracle SQL Developer interface with a connection named "my_connection". In the "Worksheet" tab, there is a query editor containing the following SQL code:

```
select employee_id, first_name, last_name, salary from employees e, departments d
where e.department_id=d.deaprtment_id and e.salary > 3000;
```

A red arrow points from the "Before formatting" callout to the code in the worksheet.

After formatting



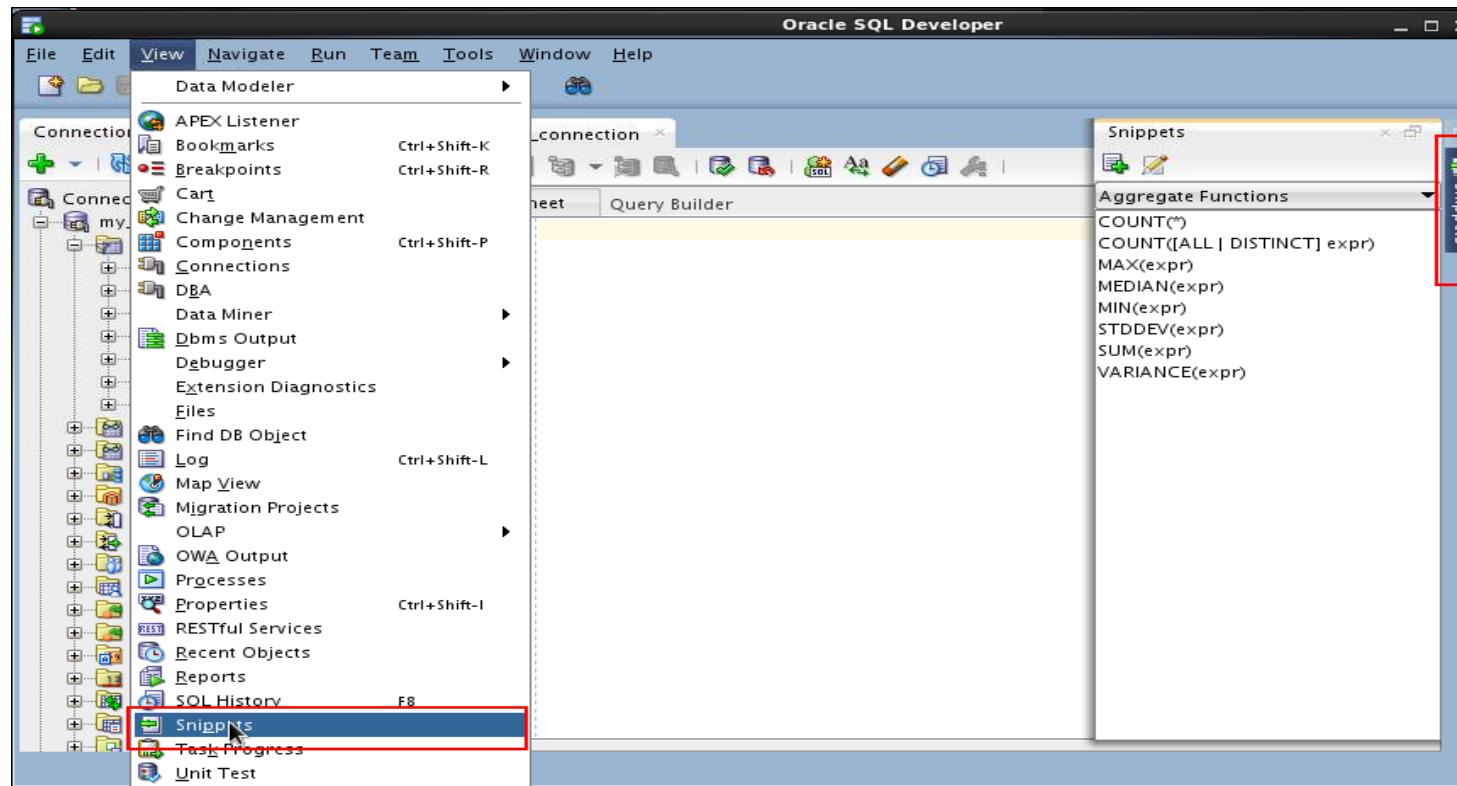
The screenshot shows the same Oracle SQL Developer interface after the code has been formatted. The SQL code now appears with proper indentation and line breaks:

```
SELECT
    employee_id,
    first_name,
    last_name,
    salary
FROM
    employees e,
    departments d
WHERE
    e.department_id=d.deaprtment_id
    AND e.salary > 3000;
```

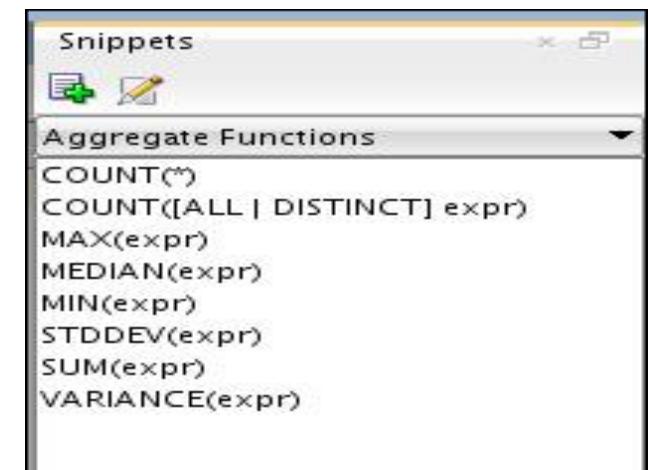
A red arrow points from the "After formatting" callout to the code in the worksheet.

Using Snippets

Snippets are code fragments that may be just syntax or examples.

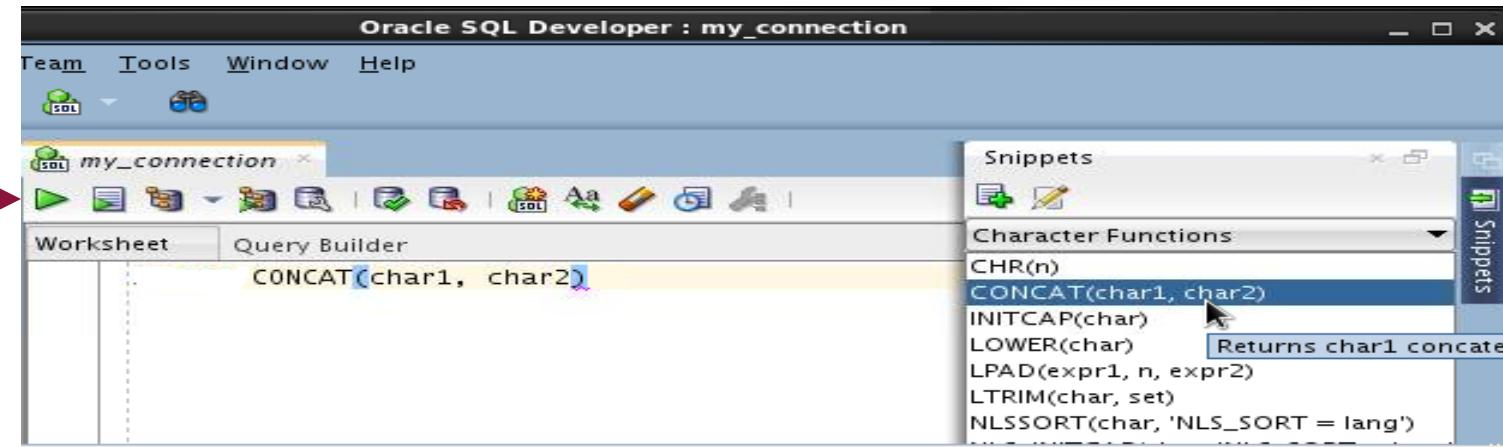


When you place your cursor here, it shows the Snippets window. From the drop-down list, you can select the functions category that you want.

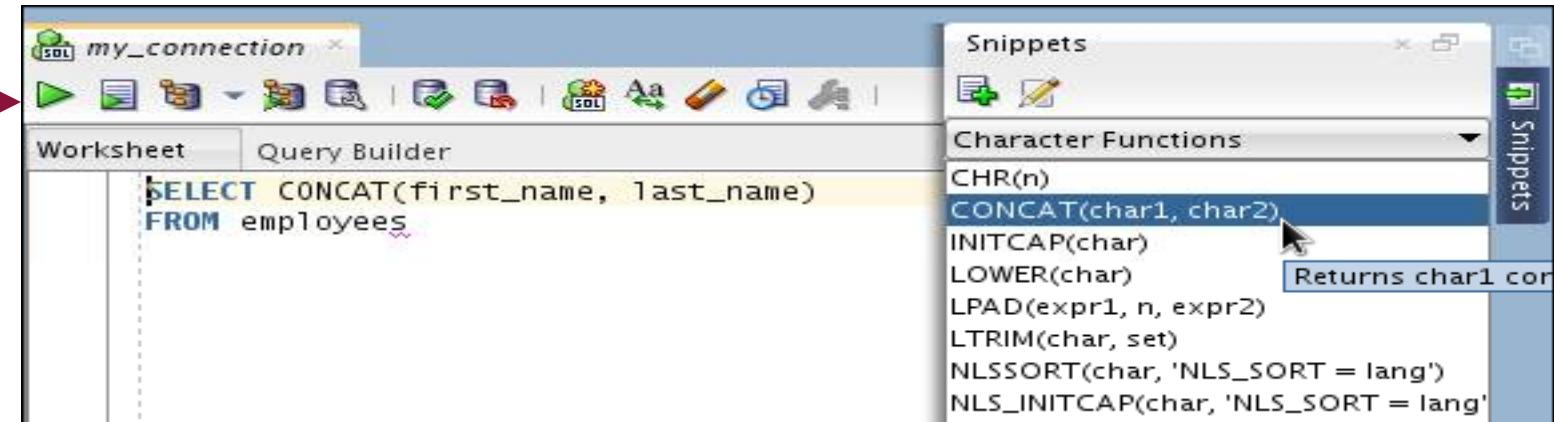


Using Snippets: Example

Inserting a snippet

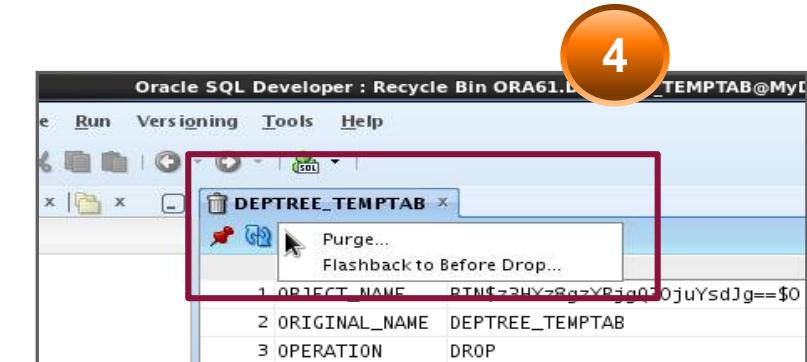
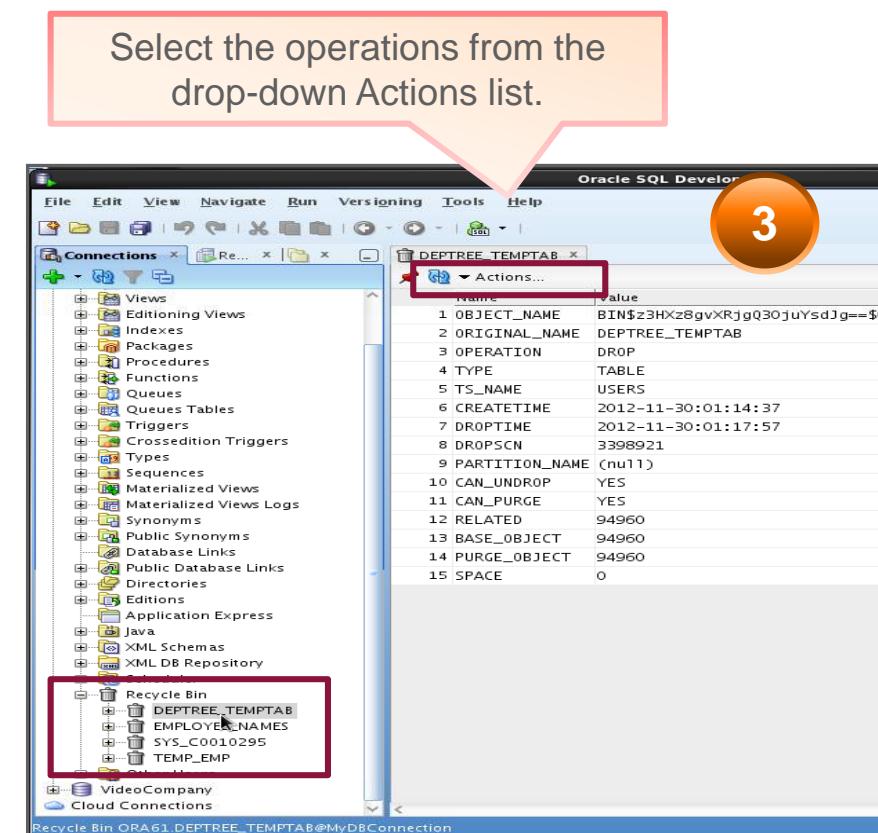
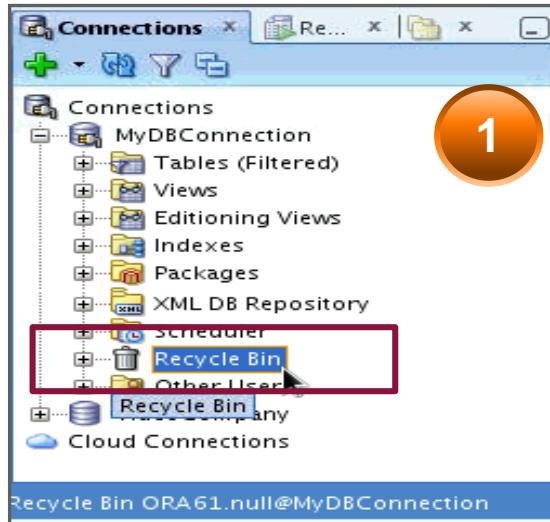


Editing the snippet



Using Recycle Bin

The Recycle Bin holds objects that have been dropped.

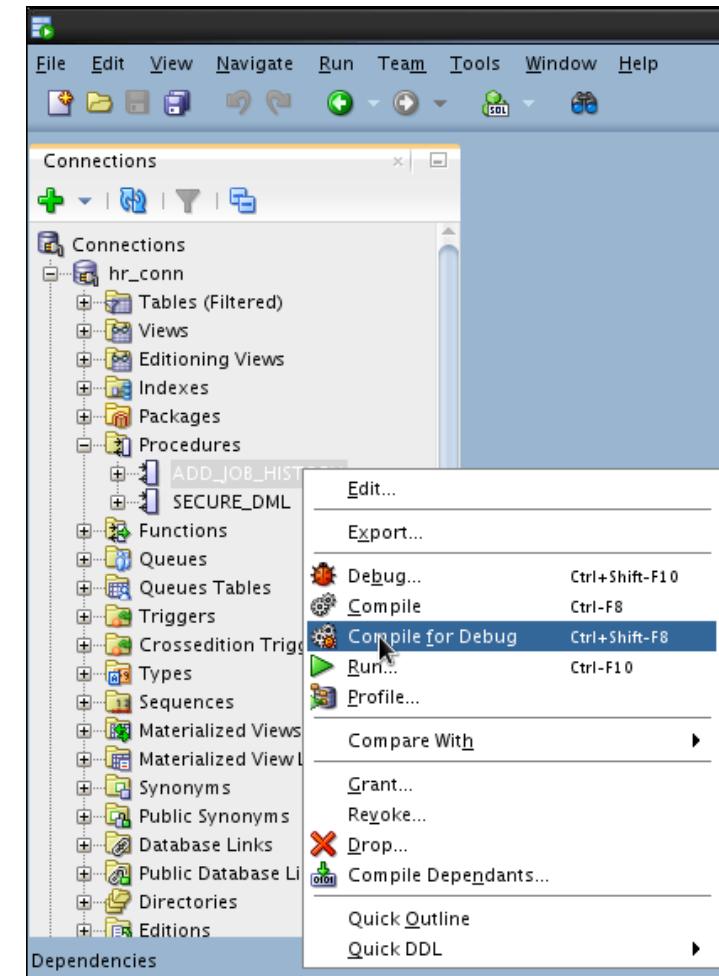


Purge: Removes the object from the Recycle Bin and deletes it

Flashback to Before Drop: Moves the object from the Recycle Bin back to its appropriate place in the Connections navigator display

Debugging Procedures and Functions

- Use SQL Developer to debug PL/SQL functions and procedures.
- Use the “Compile for Debug” option to perform a PL/SQL compilation so that the procedure can be debugged.
- Use the Debug menu options to set breakpoints, and to perform step into and step over tasks.



Database Reporting

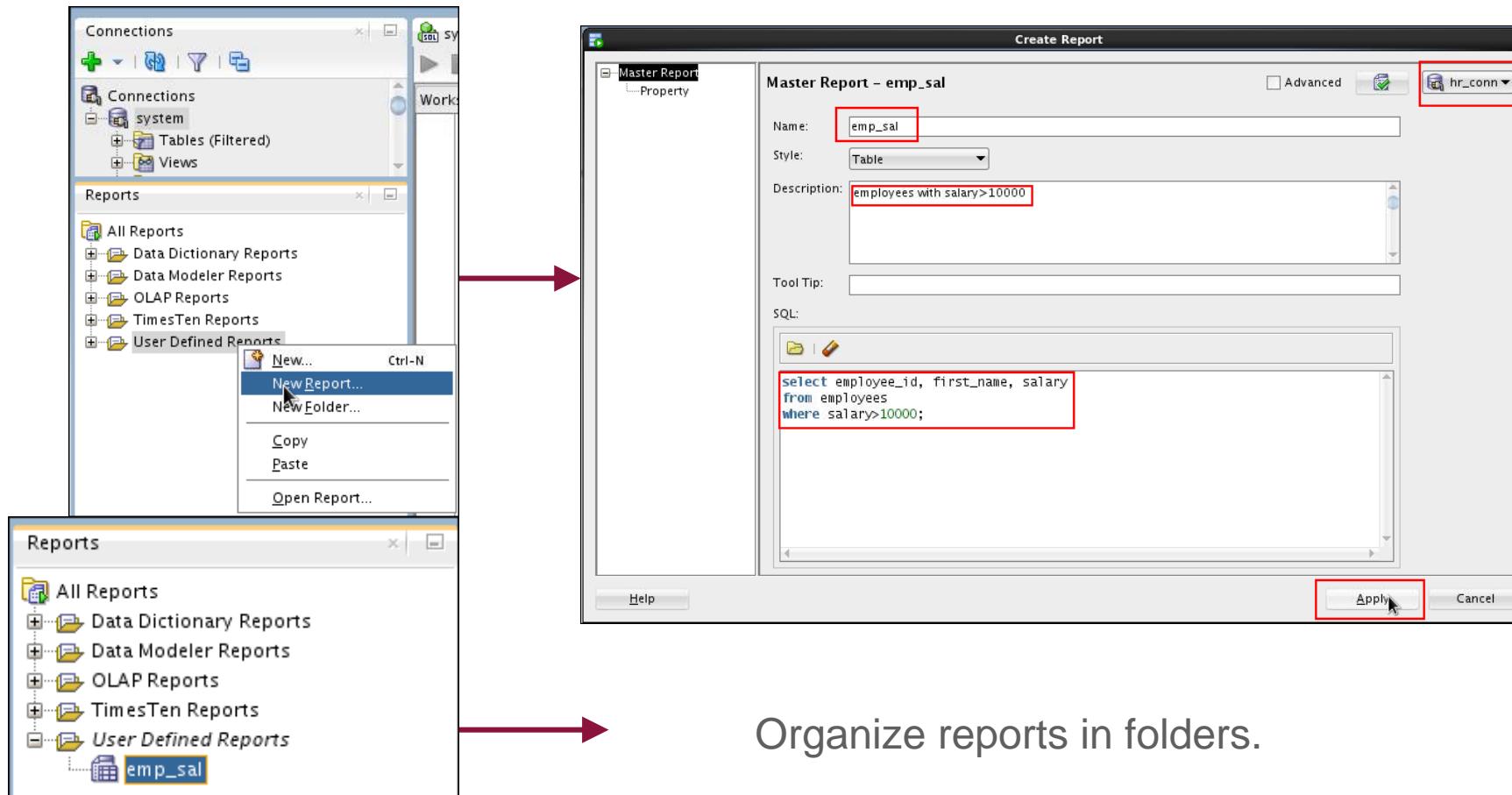
SQL Developer provides a number of predefined reports about the database and its objects.

The screenshot shows the Oracle SQL Developer interface with the 'Dependencies' report selected. The left sidebar displays various reports under the 'Reports' section, with 'Dependencies' highlighted. The main pane shows a grid of dependencies between objects in the APEX_040200 schema and other database components like WWV_FLOW, SYS, and APEX_APPLICATIONS.

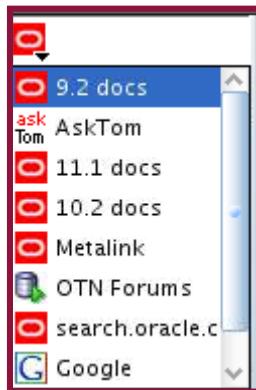
Owner	Name	Type	Referenced_Owner	Referenced_Name	Referenced_Type
1 APEX_040200	APEX	PROCEDURE	APEX_040200	WWV_FLOW	PACKAGE
2 APEX_040200	APEX	PROCEDURE	APEX_040200	WWV_FLOW_ISC	PACKAGE
3 APEX_040200	APEX	PROCEDURE	APEX_040200	WWV_FLOW_SECURITY	PACKAGE
4 APEX_040200	APEX	PROCEDURE	SYS	STANDARD	PACKAGE
5 APEX_040200	APEX	PROCEDURE	SYS	SYS_STUB_FOR_PURITY_ANALYSIS	PACKAGE
6 APEX_040200	APEXWS	PACKAGE	SYS	STANDARD	PACKAGE
7 APEX_040200	APEX_ADMIN	PROCEDURE	APEX_040200	F	PROCEDURE
8 APEX_040200	APEX_ADMIN	PROCEDURE	SYS	STANDARD	PACKAGE
9 APEX_040200	APEX_ADMIN	PROCEDURE	SYS	SYS_STUB_FOR_PURITY_ANALYSIS	PACKAGE
10 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	NV	FUNCTION
11 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WWV_FLOWS	TABLE
12 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WWV_FLOW_APPLICATION_GROUPS	TABLE
13 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WWV_FLOW_AUTHENTIFICATIONS	TABLE
14 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WWV_FLOW_COMPANIES	TABLE
15 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WWV_FLOW_COMPANY_SCHEMAS	TABLE
16 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WWV_FLOW_COMPUTATIONS	TABLE
17 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WWV_FLOW_ICON_BAR	TABLE
18 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WWV_FLOW_INSTALL_SCRIPTS	TABLE
19 APEX_040200	APEX_APPLICATIONS	VIEW	APEX_040200	WWV_FLOW_ITEMS	TABLE

Creating a User-Defined Report

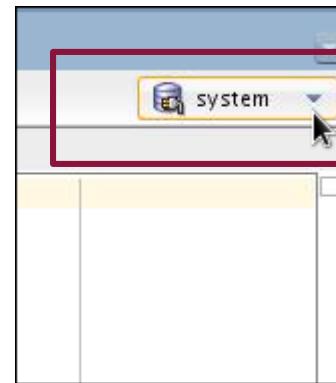
Create and save user-defined reports for repeated use.



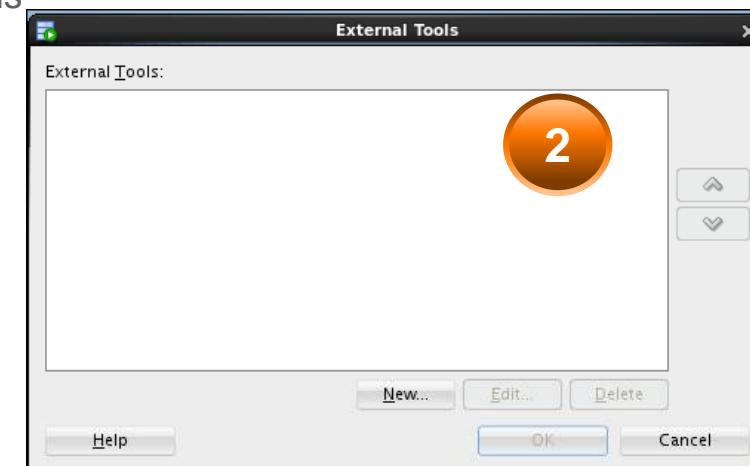
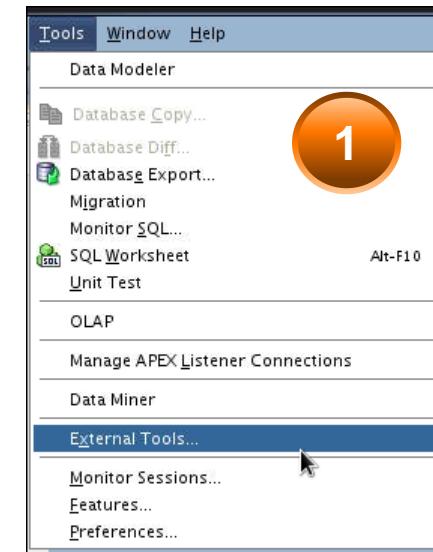
Search Engines and External Tools



Links to popular search engines and discussion forums

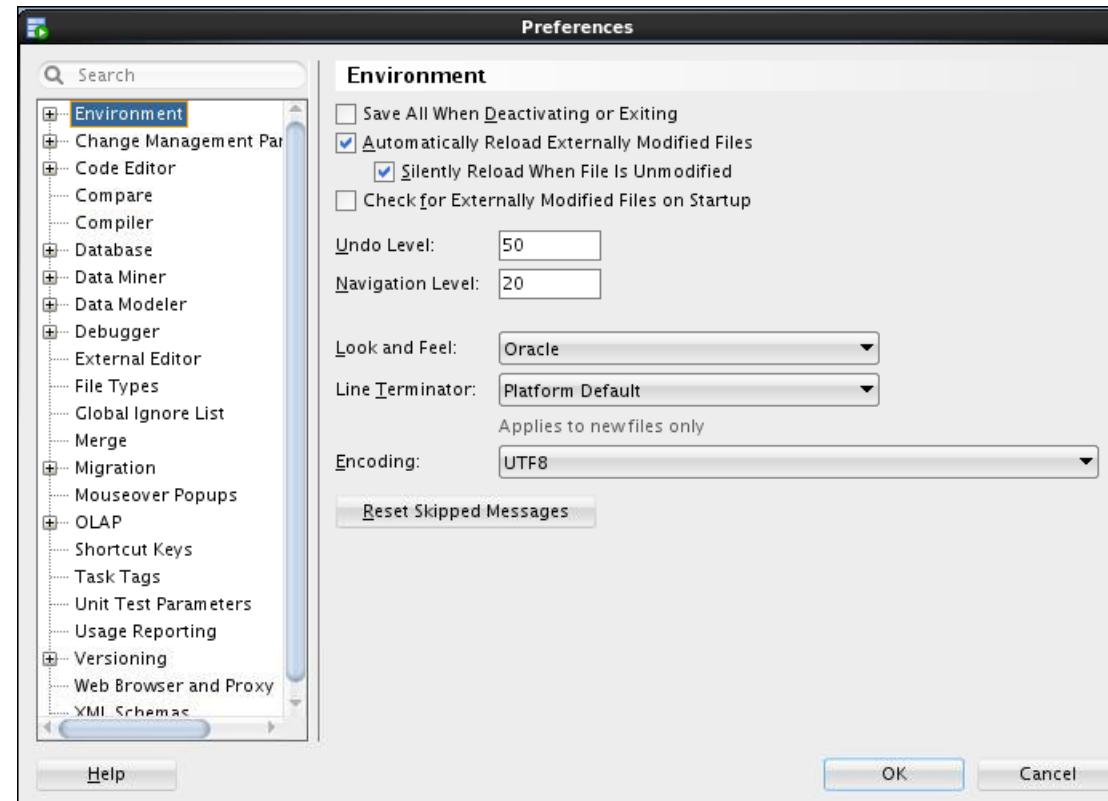


Shortcut to switch between connections



Setting Preferences

- Customize the SQL Developer interface and environment.
- In the Tools menu, select Preferences.



Resetting the SQL Developer Layout

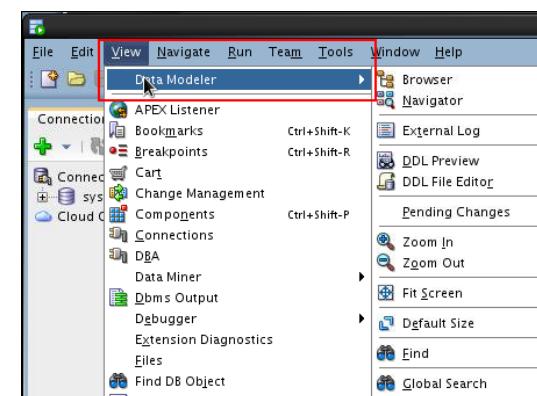
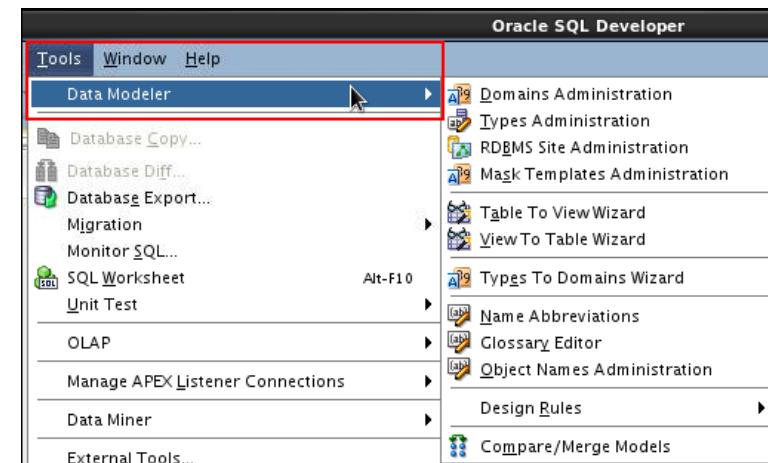
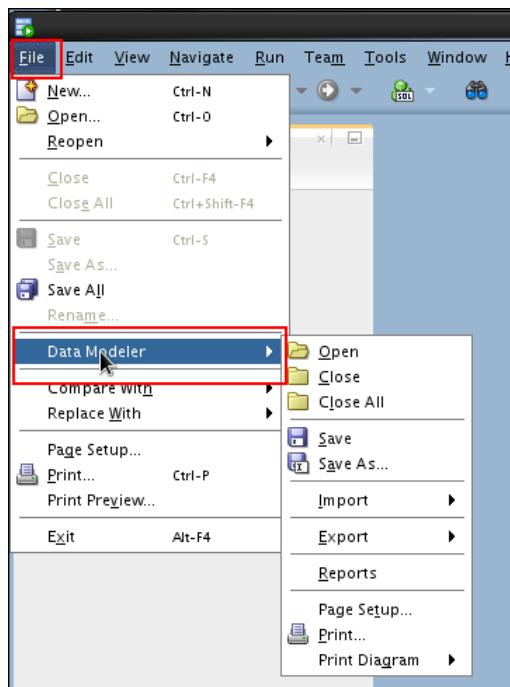


The screenshot shows a terminal window titled "oracle@EDRSR19P1:~/sqldeveloper/system4.0.0.13.80/o.ide.12.1.3.0.4". The terminal is displaying a series of Linux shell commands:

```
[oracle@EDRSR19P1 Desktop]$ locate windowlayout.xml /home/oracle/.sqldeveloper/system4.0.0.13.80/o.ide.12.1.3.0.41.131202.1730/windowinglayoutDefault.xml  
/home/oracle/.sqldeveloper/system4.0.0.13.80/o.ide.12.1.3.0.41.131202.1730/windowinglayoutDefault.xml  
[oracle@EDRSR19P1 Desktop]$ ^C  
[oracle@EDRSR19P1 Desktop]$ cd /home/oracle/.sqldeveloper/system4.0.0.13.80/o.ide.12.1.3.0.41.131202.1730  
[oracle@EDRSR19P1 o.ide.12.1.3.0.41.131202.1730]$ ls  
Debugging.layout Editing.layout runStatus.xml  
dtcache.xml projects windowinglayoutDefault.xml  
[oracle@EDRSR19P1 o.ide.12.1.3.0.41.131202.1730]$ rm windowinglayoutDefault.xml
```

Data Modeler in SQL Developer

SQL Developer includes an integrated version of SQL Developer Data Modeler.



Summary

In this appendix, you should have learned how to use SQL Developer to do the following:

- Browse, create, and edit database objects
- Execute SQL statements and scripts in SQL Worksheet
- Create and save custom reports
- Browse the Data Modeling options in SQL Developer



C

Using SQL*Plus

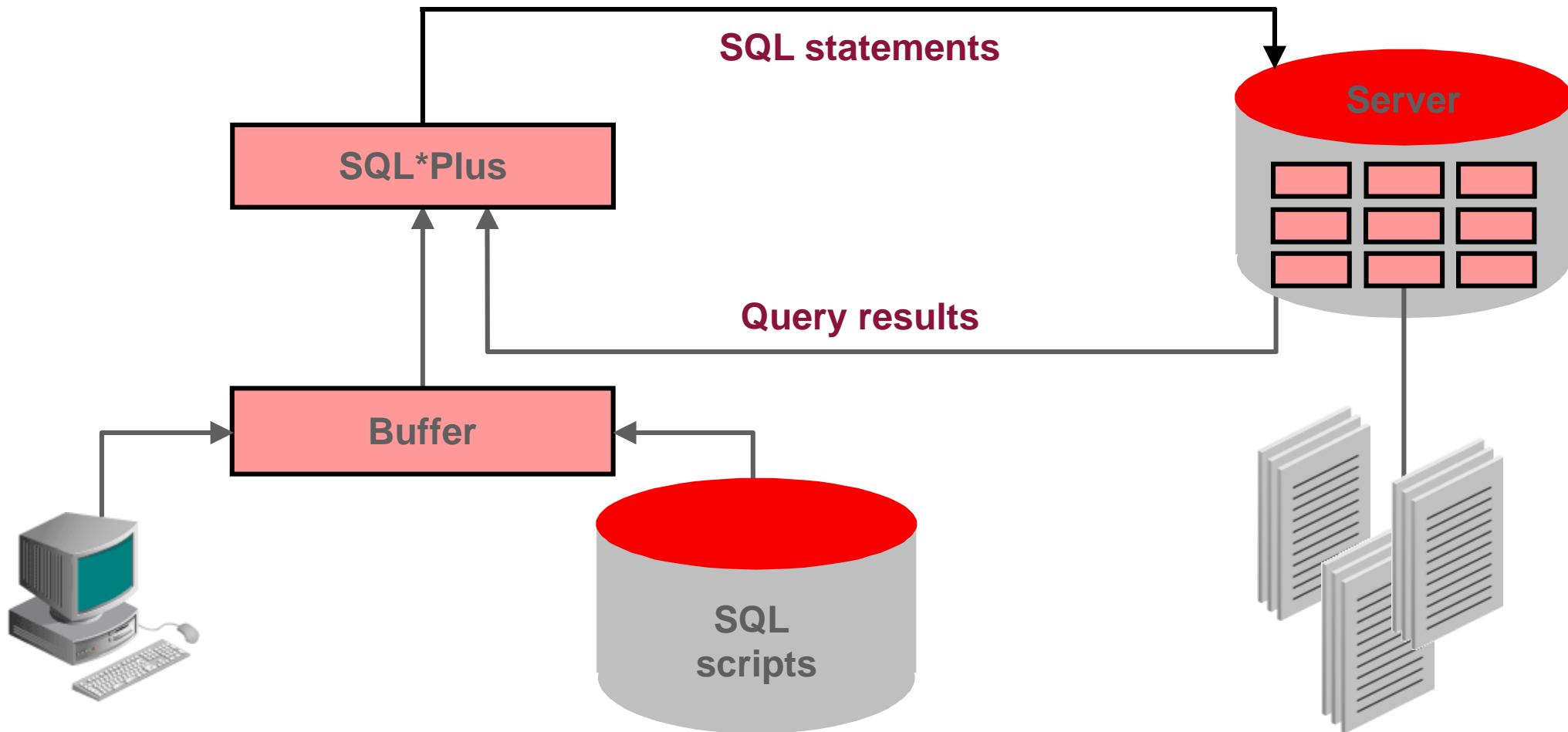
Objectives

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

- Log in to SQL*Plus
- Edit SQL commands
- Format the output by using SQL*Plus commands
- Interact with script files



SQL and SQL*Plus Interaction



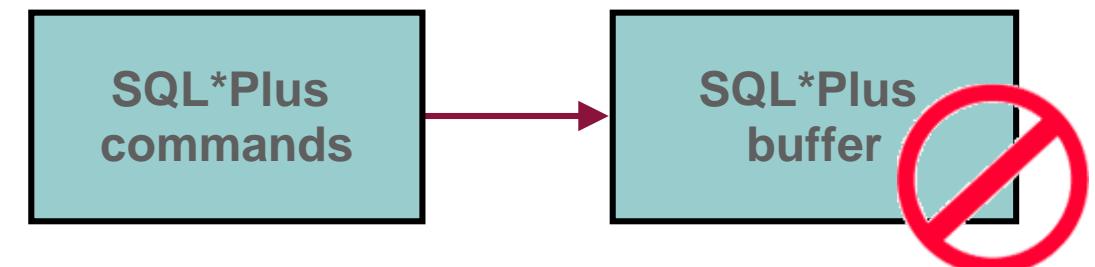
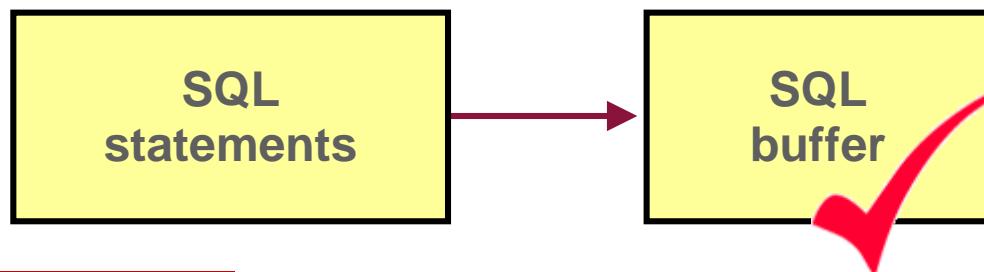
SQL Statements Versus SQL*Plus Commands

SQL

- A language
- ANSI-standard
- Keywords cannot be abbreviated.
- Statements manipulate data and table definitions in the database.

SQL*Plus

- An environment
- Oracle-proprietary
- Keywords can be abbreviated.
- Commands do not allow manipulation of values in the database.



SQL Versus SQL*Plus

SQL	SQL*Plus
Is a language for communicating with the Oracle server to access data	Recognizes SQL statements and sends them to the server
Is based on American National Standards Institute (ANSI)-standard SQL	Is the Oracle-proprietary interface for executing SQL statements
Manipulates data and table definitions in the database	Does not allow manipulation of values in the database
Is entered into the SQL buffer on one or more lines	Is entered one line at a time, not stored in the SQL buffer
Does not have a continuation character	Uses a dash (-) as a continuation character if the command is longer than one line
Cannot be abbreviated	Can be abbreviated
Uses a termination character to execute commands immediately	Does not require termination characters; executes commands immediately
Uses functions to perform some formatting	Uses commands to format data

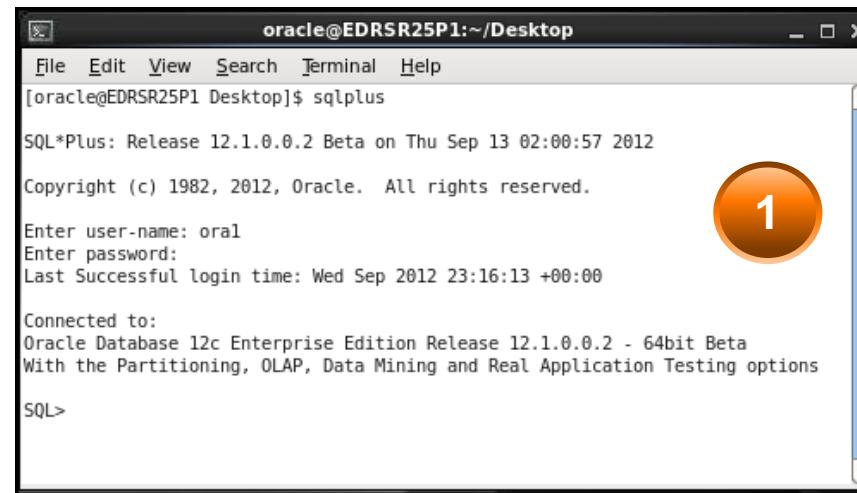
Using SQL*Plus

- Log in to SQL*Plus.
- Describe the table structure.
- Edit your SQL statement.
- Execute SQL from SQL*Plus.
- Save SQL statements to files and append SQL statements to files.
- Execute saved files.
- Load commands to be edited from the file to the buffer.

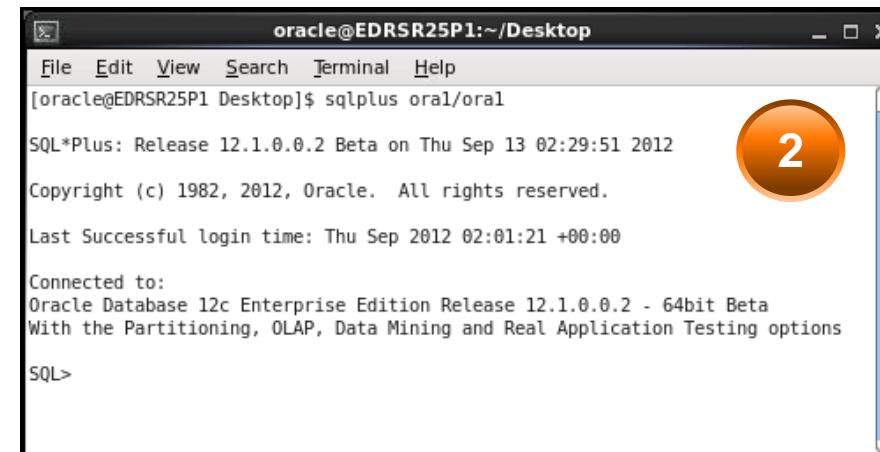
SQL Plus Commands: Categories

Category	Purpose
Environment	Affect the general behavior of SQL statements for the session.
Format	Format query results.
File manipulation	Save, load, and run script files.
Execution	Send SQL statements from the SQL buffer to the Oracle server.
Edit	Modify SQL statements in the buffer.
Interaction	Create and pass variables to SQL statements, print variable values, and print messages to the screen.
Miscellaneous	Connect to the database, manipulate the SQL*Plus environment, and display column definitions.

Logging In to SQL*Plus



sqlplus [username[/password[@database]]]



Displaying the Table Structure

Use the SQL*Plus DESCRIBE command to display the structure of a table:

```
DESC [RIBE] tablename
```

Displaying the Table Structure

```
DESCRIBE departments
```

Name	Null	Type
DEPARTMENT_ID	NOT NULL	NUMBER(4)
DEPARTMENT_NAME	NOT NULL	VARCHAR2(30)
MANAGER_ID		NUMBER(6)
LOCATION_ID		NUMBER(4)

SQL*Plus Editing Commands

Command	Description
A[PPEND] text	Adds text to the end of the current line
C[HANGE] / old / new	Changes old text to new in the current line
C[HANGE] / text /	Deletes text from the current line
CL[EAR] BUFF[ER]	Deletes all lines from the SQL buffer
DEL	Deletes current line
DEL n	Deletes line n
DEL m n	Deletes lines m to n inclusive

SQL*Plus Editing Commands

Command	Description
I[NPUT]	Inserts an indefinite number of lines
I[NPUT] text	Inserts a line consisting of text
L[IST]	Lists all lines in the SQL buffer
L[IST] n	Lists one line (specified by n)
L[IST] m n	Lists a range of lines (m to n) inclusive
R[UN]	Displays and runs the current SQL statement in the buffer
n	Specifies the line to make the current line
n text	Replaces line n with text
0 text	Inserts a line before line 1

Using LIST, n, and APPEND

LIST

```
1  SELECT last_name  
2* FROM employees
```

1

```
1* SELECT last_name
```

A , job_id

```
1* SELECT last_name, job_id
```

LIST

```
1  SELECT last_name, job_id  
2* FROM employees
```

Using the CHANGE Command

LIST

```
1* SELECT * from employees
```

c/employees/departments

```
1* SELECT * from departments
```

LIST

```
1* SELECT * from departments
```

SQL*Plus File Commands

Command	Description
SAV[E] filename [.ext] [REP[LACE]APP[END]]	Saves the current contents of the SQL buffer to a file. Use APPEND to add to an existing file; use REPLACE to overwrite an existing file. The default extension is .sql.
GET filename [.ext]	Writes the contents of a previously saved file to the SQL buffer. The default extension for the file name is .sql.
STA[RT] filename [.ext]	Runs a previously saved command file
@ filename	Runs a previously saved command file (same as START)
ED[IT]	Invokes the editor and saves the buffer contents to a file named afiedt.buf
ED[IT] [filename[.ext]]	Invokes the editor to edit the contents of a saved file
SPO[OL] [filename[.ext]] OFF OUT	Stores query results in a file. OFF closes the spool file. OUT closes the spool file and sends the file results to the printer.
Exit	Exits SQL*Plus

Using the SAVE and START Commands

LIST

```
1  SELECT last_name, manager_id, department_id  
2* FROM employees
```

SAVE my_query

Created file my_query

START my_query

LAST_NAME	MANAGER_ID	DEPARTMENT_ID
King		90
Kochhar	100	90
...		
107 rows selected.		

SERVEROUTPUT Command

- Use the SET SERVEROUT [PUT] command to control whether to display the output of stored procedures or PL/SQL blocks in SQL*Plus.
- The DBMS_OUTPUT line length limit is increased from 255 bytes to 32767 bytes.
- The default size is now unlimited.
- Resources are not preallocated when SERVEROUTPUT is set.
- Because there is no performance penalty, use UNLIMITED unless you want to conserve physical memory.

```
SET SERVEROUT[PUT] {ON | OFF} [SIZE {n | UNLIMITED} ] [FOR[MAT] {WRA[PPED] |  
WOR[D_WRAPPED] | TRU[NATED]} ]
```

Using the SQL*Plus SPOOL Command

```
SPO[OL] [file_name [.ext]] [CRE[ATE] | REP[LACE] | APP[END]] | OFF | OUT]
```

Option	Description
file_name[.ext]	Spools output to the specified file name
CRE[ATE]	Creates a new file with the name specified
REP[LACE]	Replaces the contents of an existing file. If the file does not exist, REPLACE creates the file.
APP[END]	Adds the contents of the buffer to the end of the file that you specify
OFF	Stops spooling
OUT	Stops spooling and sends the file to your computer's standard (default) printer

Using the AUTOTRACE Command

- It displays a report after the successful execution of SQL data manipulation statements (DML) statements such as SELECT, INSERT, UPDATE, or DELETE.
- The report can now include execution statistics and the query execution path.

```
SET AUTOT[RACE] {ON | OFF | TRACE[ONLY]} [EXP[LAIN]] [STAT[ISTICS]]
```

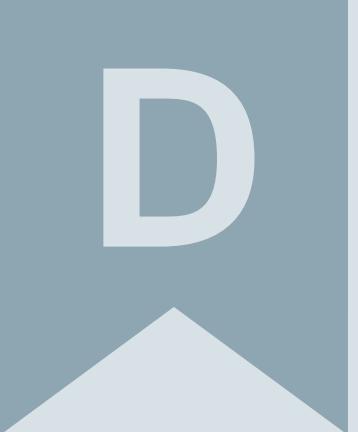
```
SET AUTOTRACE ON
-- The AUTOTRACE report includes both the optimizer
-- execution path and the SQL statement execution
-- statistics
```

Summary

In this appendix, you should have learned how to use SQL*Plus as an environment to do the following:

- Execute SQL statements
- Edit SQL statements
- Format the output
- Interact with script files



A large blue ribbon graphic with a white letter 'R' on it, positioned in the top right corner of the slide.

PL/SQL Programming Concepts: Review

Objectives

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

- Describe PL/SQL basics
- List restrictions on calling functions from SQL expressions
- Review PL/SQL packages
- Identify how explicit cursors are processed
- Handle exceptions
- Use the `RAISE_APPLICATION_ERROR` procedure
- Manage dependencies
- Use Oracle-supplied packages



Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages



PL/SQL Block Structure

```
DECLARE  
BEGIN  
  
EXCEPTION  
  
END ;
```

Anonymous
PL/SQL block

```
<header>  
IS | AS  
  
BEGIN  
  
EXCEPTION  
  
END ;
```

Stored
program unit

Naming Conventions

Advantages of proper naming conventions:

Easier to read

Understandable

Gives information about the functionality

Easier to debug

Ensures consistency

Improves performance

Naming Conventions

Identifier	Convention	Example
Variable	v_prefix	v_product_name
Constant	c_prefix	c_tax
Parameter	p_prefix	p_cust_id
Exception	e_prefix	e_check_credit_limit
Cursor	cur_prefix	cur_orders
Type	typ_prefix	typ_customer

Procedures

A procedure is:

- A named PL/SQL block that performs a sequence of actions and optionally returns a value or values
- Stored in the database as a schema object
- Used to promote reusability and maintainability

```
CREATE [OR REPLACE] PROCEDURE procedure_name
  [(parameter1 [mode] datatype1,
    parameter2 [mode] datatype2, ...)]
IS | AS
  [local_variable_declarations; ...]
BEGIN
  -- actions;
END [procedure_name];
```

Procedure: Example

```
CREATE OR REPLACE PROCEDURE get_avg_order
(p_cust_id NUMBER,
 p_cust_last_name VARCHAR2,
 p_order_tot NUMBER)
IS
    v_cust_ID customers.customer_id%type;
    v_cust_name customers.cust_last_name%type;
    v_avg_order NUMBER;
BEGIN
    SELECT customers.customer_id, customers.cust_last_name,
    AVG(orders.order_total)
    INTO v_cust_id, v_cust_name, v_avg_order
    FROM CUSTOMERS, ORDERS
    WHERE customers.customer_id=orders.customer_id
    GROUP BY customers.customer_id, customers.cust_last_name;
END;
```

Stored Functions

A function is:

- A named block that must return a value
- Stored in the database as a schema object
- Called as part of an expression or used to provide a parameter value

```
CREATE [OR REPLACE] FUNCTION function_name
  [(parameter1 [mode1] datatype1, ...)]
  RETURN datatype IS|AS
    [local_variable_declarations; ...]
  BEGIN
    -- actions;
    RETURN expression;
  END [function_name];
```

Functions: Example

- Create the function:

```
CREATE OR REPLACE FUNCTION get_credit
  (v_id customers.customer_id%TYPE) RETURN NUMBER IS
    v_credit customers.credit_limit%TYPE := 0;
BEGIN
  SELECT credit_limit
  INTO   v_credit
  FROM   customers
  WHERE  customer_id = v_id;
  RETURN (v_credit);
END get_credit;
/
```

- Invoke the function as an expression or as a parameter value:

```
EXECUTE dbms_output.put_line(get_credit(101))
```

Ways to Execute Functions

- Invoke as part of a PL/SQL expression:
 - Using a host variable to obtain the result:

```
VARIABLE v_credit NUMBER  
EXECUTE :v_credit := get_credit(101)
```

- Using a local variable to obtain the result:

```
DECLARE v_credit customers.credit_limit%type;  
BEGIN  
    v_credit := get_credit(101); ...  
END;
```

- Use as a parameter to another subprogram:

```
EXECUTE dbms_output.put_line(get_credit(101))
```

- Use in a SQL statement (subject to restrictions):

```
SELECT get_credit(customer_id) FROM customers;
```

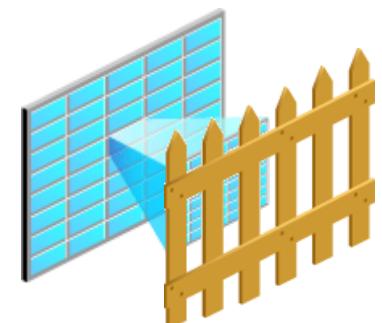
Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages



Restrictions on Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
 - Be stored in the database
 - Accept only `IN` parameters with valid SQL data types, not PL/SQL-specific types
 - Return valid SQL data types, not PL/SQL-specific types
- When calling functions in SQL statements:
 - You must own the function or have the `EXECUTE` privilege

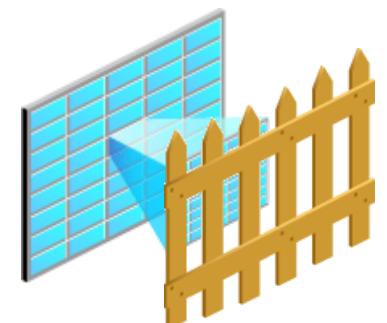


Restrictions on Calling Functions from SQL Expressions

Functions called from SQL statements cannot:

- Contain DML statements
- End transactions (that is, cannot execute COMMIT or ROLLBACK operations)

Note: Calls to subprograms that break these restrictions are also not allowed in the function.



Lesson Agenda

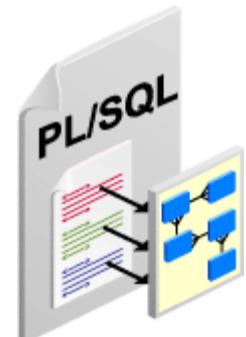
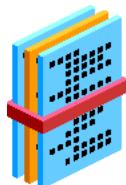
- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages



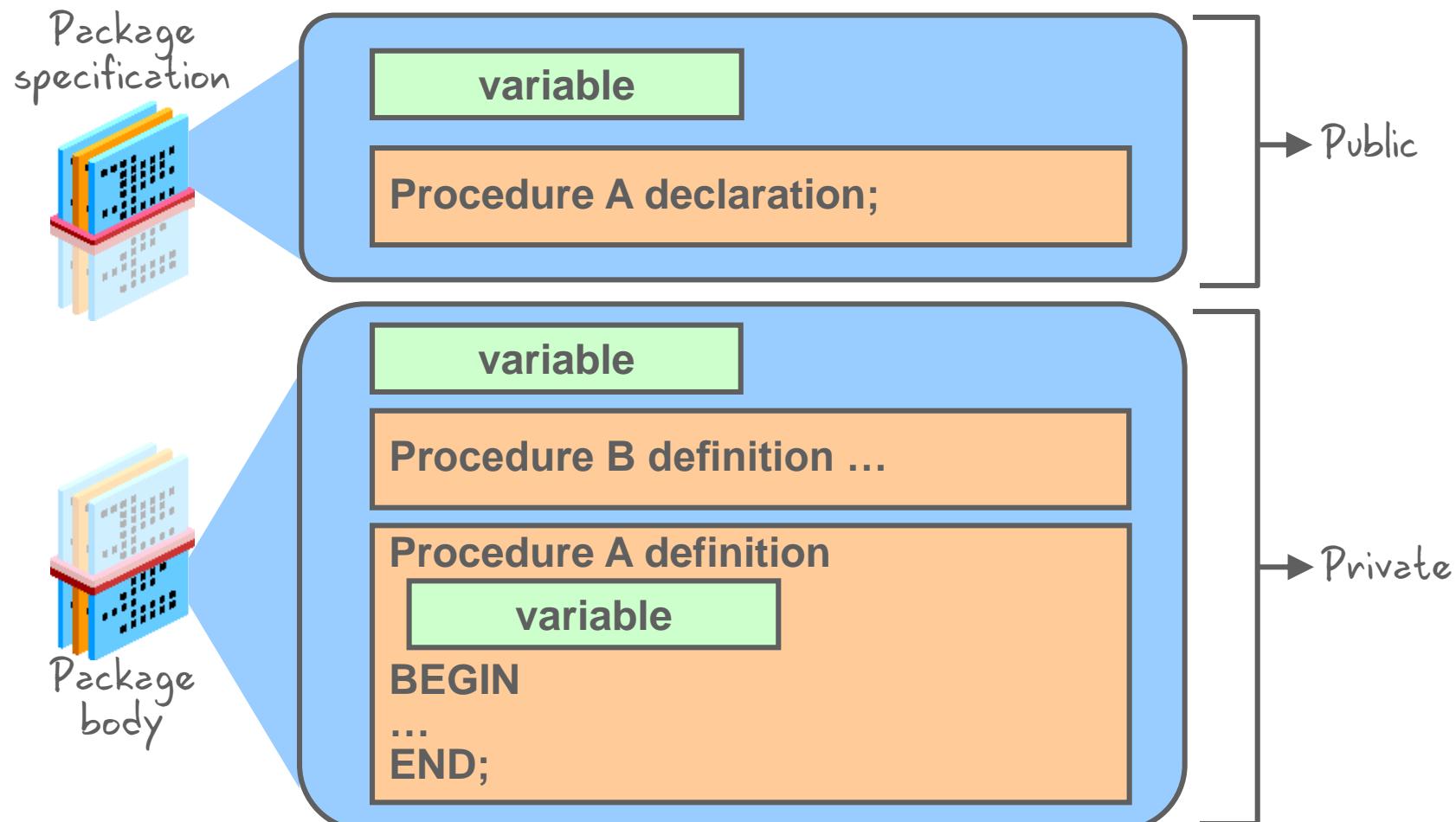
PL/SQL Packages: Review

PL/SQL packages:

- Group logically related components:
 - PL/SQL types
 - Variables, data structures, and exceptions
 - Subprograms: procedures and functions
- Consist of two parts:
 - A specification
 - A body
- Enable the Oracle server to read multiple objects into memory simultaneously



Components of a PL/SQL Package



Creating the Package Specification

Syntax:

```
CREATE [OR REPLACE] PACKAGE package_name IS | AS  
  public type and variable declarations  
  subprogram specifications  
END [package_name];
```

- The OR REPLACE option drops and re-creates the package specification.
- Variables declared in the package specification are initialized to NULL by default.
- All constructs declared in a package specification are visible to users who are granted privileges on the package.

Creating the Package Body

Syntax:

```
CREATE [OR REPLACE] PACKAGE BODY package_name IS|AS  
    private type and variable declarations  
    subprogram bodies  
    [BEGIN initialization statements]  
    END [package_name];
```

- The OR REPLACE option drops and re-creates the package body.
- Identifiers defined in the package body are private and not visible outside the package body.
- All private constructs must be declared before they are referenced.
- Public constructs are visible to the package body.

Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages



Cursor

- A cursor is a pointer to the private memory area allocated by the Oracle server.
- There are two types of cursors:

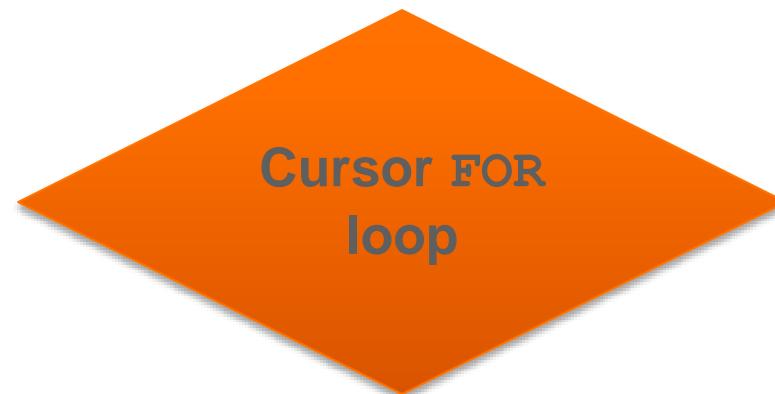
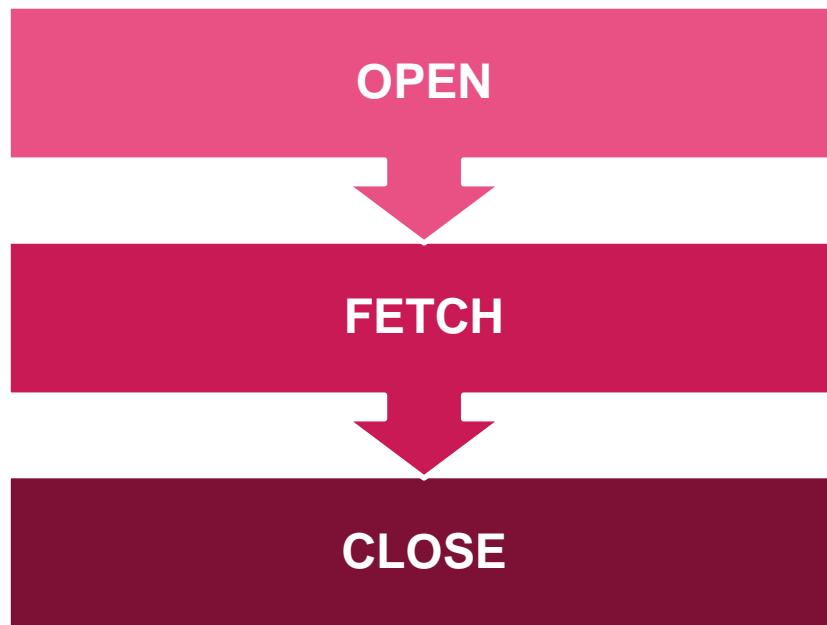
Implicit cursors

Created and managed
internally by the Oracle server
to process SQL statements

Explicit cursors

Explicitly declared by the programmer

Processing Explicit Cursors



Explicit Cursor Attributes

Every explicit cursor has the following attributes:

- *cursor_name*%FOUND
- *cursor_name*%ISOPEN
- *cursor_name*%NOTFOUND
- *cursor_name*%ROWCOUNT



Cursor FOR Loops

Syntax:

```
FOR record_name IN cursor_name LOOP  
    statement1;  
    statement2;  
    . . .  
END LOOP;
```

- The cursor FOR loop is a shortcut to process explicit cursors.
- Implicit open, fetch, exit, and close occur.
- The record is implicitly declared.

Cursor: Example

```
DECLARE
    CURSOR cur_cust IS
        SELECT cust_first_name, credit_limit
        FROM customers
        WHERE credit_limit > 4000;
BEGIN
    FOR v_cust_record IN cur_cust
    LOOP
        DBMS_OUTPUT.PUT_LINE
            (v_cust_record.cust_first_name || ' ' ||
             v_cust_record.credit_limit);
    END LOOP;
END;
/
```

Lesson Agenda

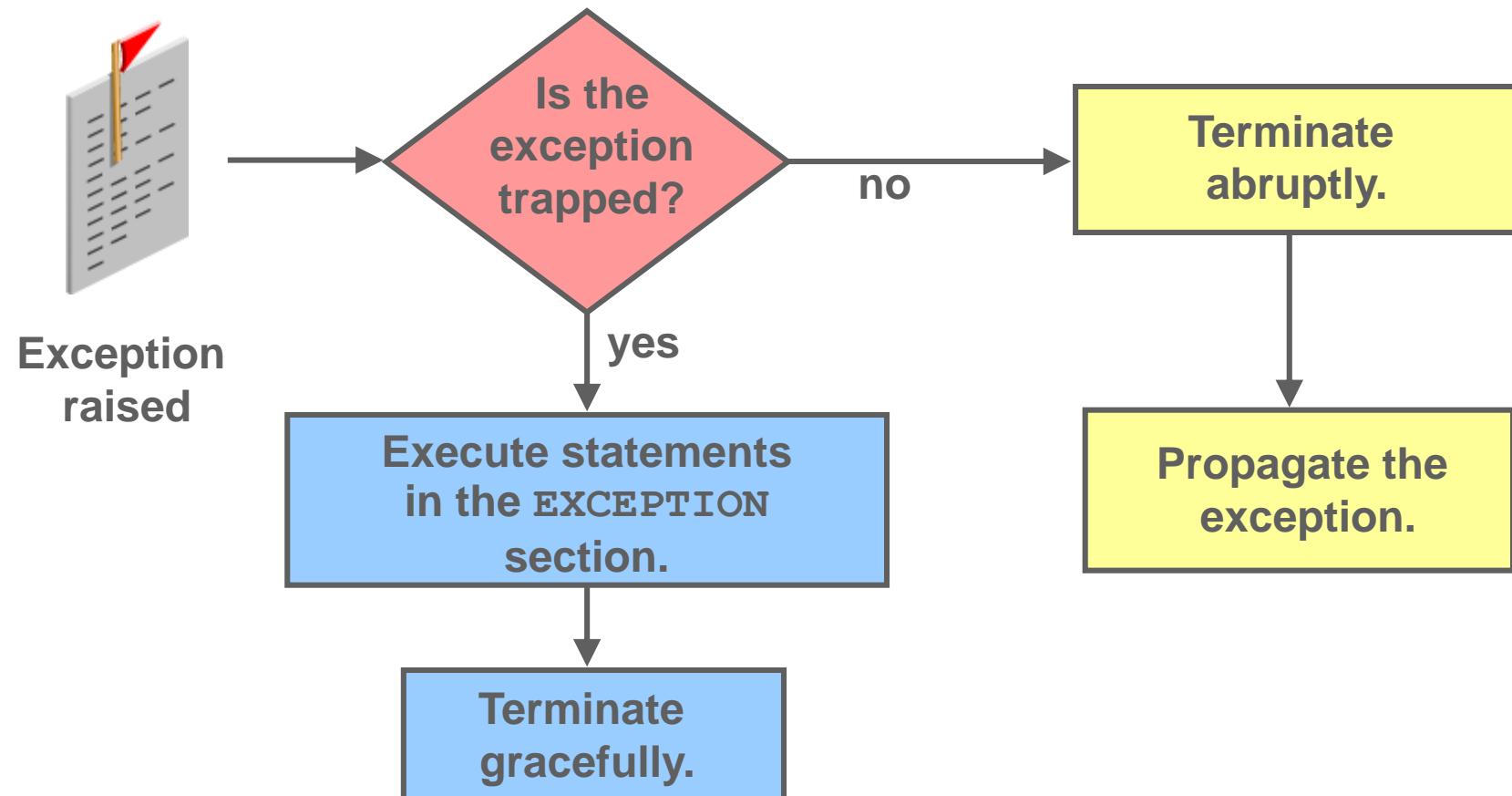
- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages



Handling Exceptions

- An exception is an error in PL/SQL that is raised during program execution.
- An exception can be raised:
 - Implicitly by the Oracle server
 - Explicitly by the program
- An exception can be handled:
 - By trapping it with a handler
 - By propagating it to the calling environment
 - By trapping and propagating it

Handling Exceptions



Exceptions: Example

```
DECLARE
    v_lname VARCHAR2(15);
BEGIN
    SELECT cust_last_name INTO v_lname FROM customers
    WHERE cust_first_name='Ally';
    DBMS_OUTPUT.PUT_LINE ('Ally''s last name is : '
                          ||v_lname);
EXCEPTION
    WHEN TOO_MANY_ROWS THEN
        DBMS_OUTPUT.PUT_LINE (' Your select statement
retrieved multiple rows. Consider using a
cursor.');
END;
/
```

Predefined Oracle Server Errors

- Reference the predefined name in the exception-handling routine.
- Sample predefined exceptions:
 - NO_DATA_FOUND
 - TOO_MANY_ROWS
 - INVALID_CURSOR
 - ZERO_DIVIDE
 - DUP_VAL_ON_INDEX



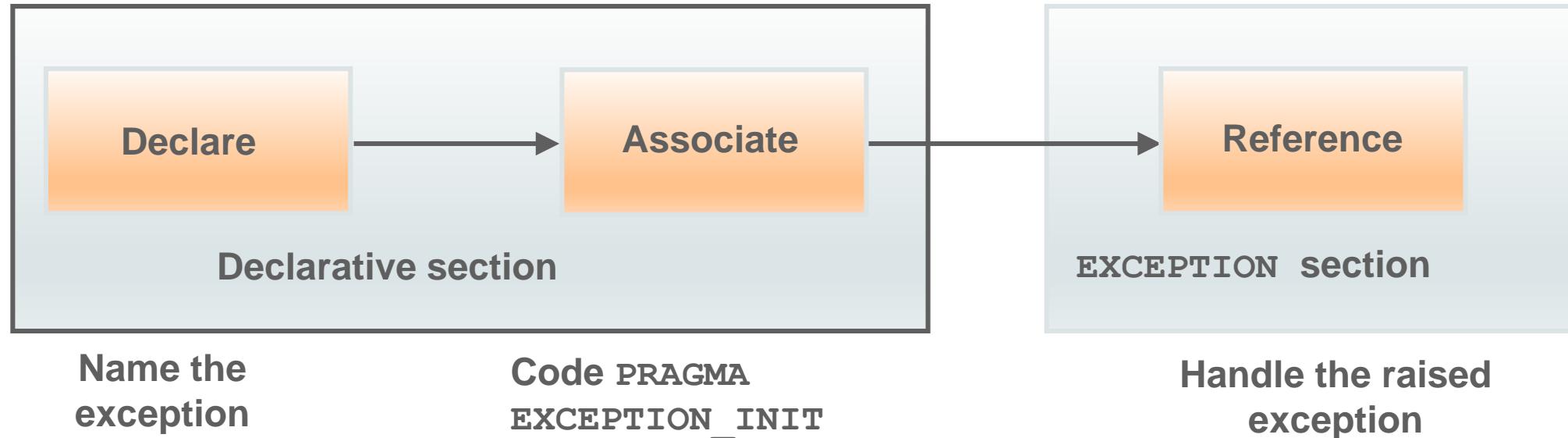
Predefined Oracle Server Exceptions

Exception Name	Oracle Server Error Number	Description
ACCESS_INTO_NULL	ORA-06530	Attempted to assign values to the attributes of an uninitialized object
CASE_NOT_FOUND	ORA-06592	None of the choices in the WHEN clauses of a CASE statement is selected, and there is no ELSE clause.
COLLECTION_IS_NULL	ORA-06531	Attempted to apply collection methods other than EXISTS to an uninitialized nested table or varray
CURSOR_ALREADY_OPEN	ORA-06511	Attempted to open an already open cursor
DUP_VAL_ON_INDEX	ORA-00001	Attempted to insert a duplicate value
INVALID_CURSOR	ORA-01001	An Illegal cursor operation occurred.
INVALID_NUMBER	ORA-01722	Conversion of character string to number failed.
LOGIN_DENIED	ORA-01017	Logging on to the Oracle server with an invalid username or password
NO_DATA_FOUND	ORA-01403	Single-row SELECT returned no data.
NOT_LOGGED_ON	ORA-01012	The PL/SQL program issued a database call without being connected to the Oracle server.

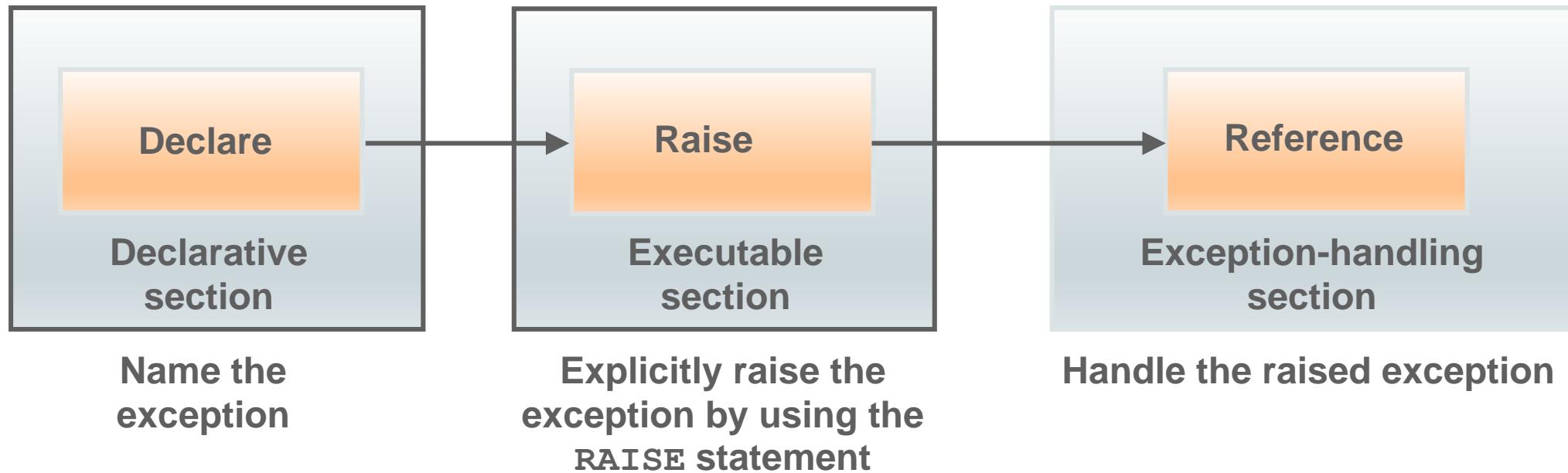
Predefined Oracle Server Exceptions

Exception Name	Oracle Server Error Number	Description
PROGRAM_ERROR	ORA-06501	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	ORA-06504	Host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types.
STORAGE_ERROR	ORA-06500	PL/SQL ran out of memory or memory is corrupted.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Referenced a nested table or varray element by using an index number larger than the number of elements in the collection
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Referenced a nested table or varray element by using an index number that is outside the legal range (for example, –1)
SYS_INVALID_ROWID	ORA-01410	The conversion of a character string into a universal ROWID failed because the character string did not represent a valid ROWID.
TIMEOUT_ON_RESOURCE	ORA-00051	Time-out occurred while the Oracle server was waiting for a resource.
TOO_MANY_ROWS	ORA-01422	Single-row SELECT returned more than one row.
VALUE_ERROR	ORA-06502	Arithmetic, conversion, truncation, or size-constraint error occurred.
ZERO_DIVIDE	ORA-01476	Attempted to divide by zero

Trapping Non-Predefined Oracle Server Errors



Trapping User-Defined Exceptions



Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- **Using the RAISE_APPLICATION_ERROR procedure**
- Managing dependencies
- Using Oracle-supplied packages



RAISE_APPLICATION_ERROR Procedure

Syntax:

```
raise_application_error (error_number,  
                      message[, {TRUE | FALSE}]);
```

- You can use this procedure to issue user-defined error messages from stored subprograms.
- You can report errors to your application and avoid returning unhandled exceptions.

RAISE_APPLICATION_ERROR Procedure

- Is used in two places:
 - Executable section
 - Exception section
- Returns error conditions to the user in a manner consistent with other Oracle server errors

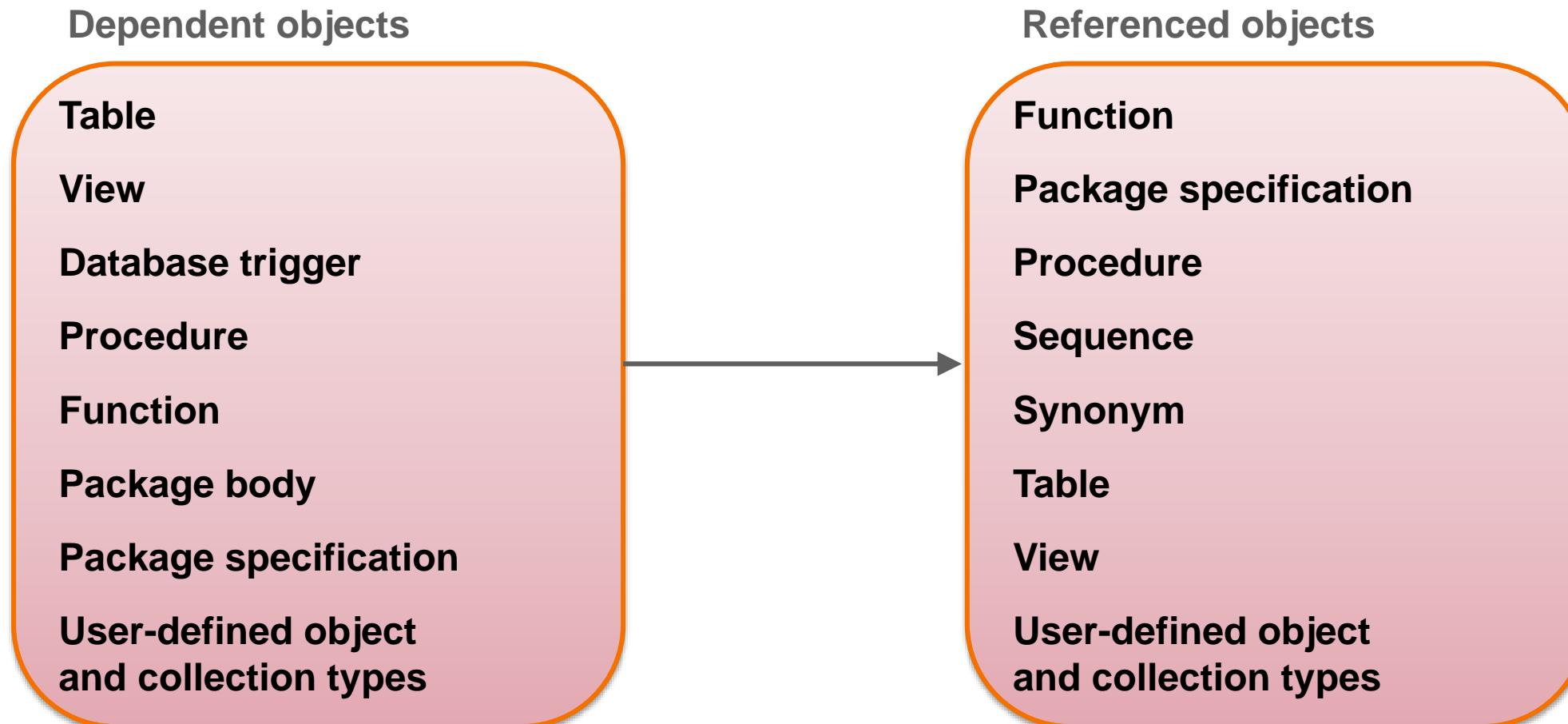


Lesson Agenda

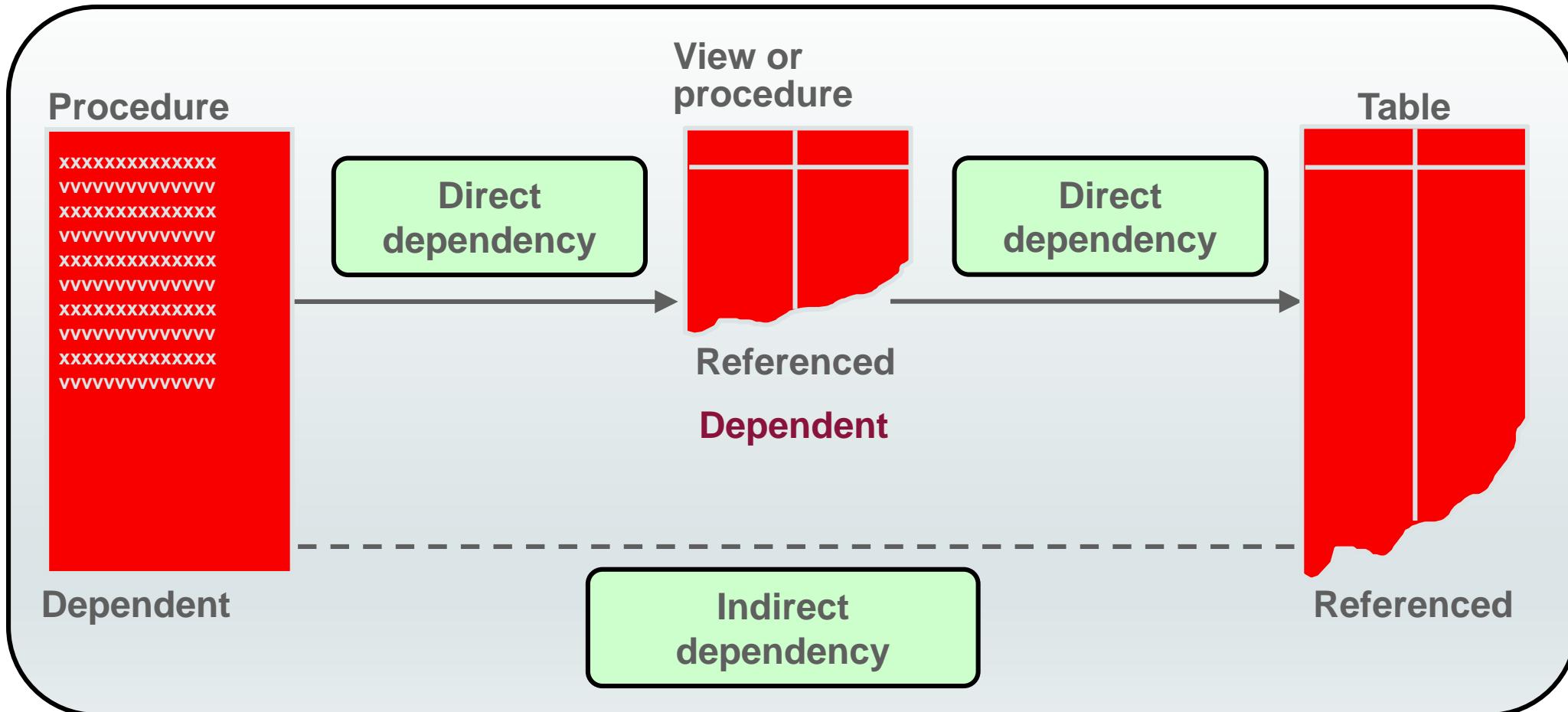
- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages



Dependencies



Dependencies



Displaying Direct and Indirect Dependencies

1. Run the `utldtree.sql` script to create the objects that enable you to display the direct and indirect dependencies.
2. Execute the `DEPTREE_FILL` procedure:

```
EXECUTE deptree_fill('TABLE', 'OE', 'CUSTOMERS')
```

Lesson Agenda

- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `RAISE_APPLICATION_ERROR` procedure
- Managing dependencies
- Using Oracle-supplied packages



Using Oracle-Supplied Packages

Oracle-supplied packages:

- Are provided with the Oracle server
- Extend the functionality of the database
- Enable access to certain SQL features that are normally restricted for PL/SQL

For example, the `DBMS_OUTPUT` package was originally designed to debug PL/SQL programs.



Some of the Oracle-Supplied Packages

Here is an abbreviated list of some Oracle-supplied packages:

DBMS_ALERT

DBMS_LOCK

DBMS_SESSION

DBMS_OUTPUT

HTP

UTL_FILE

UTL_MAIL

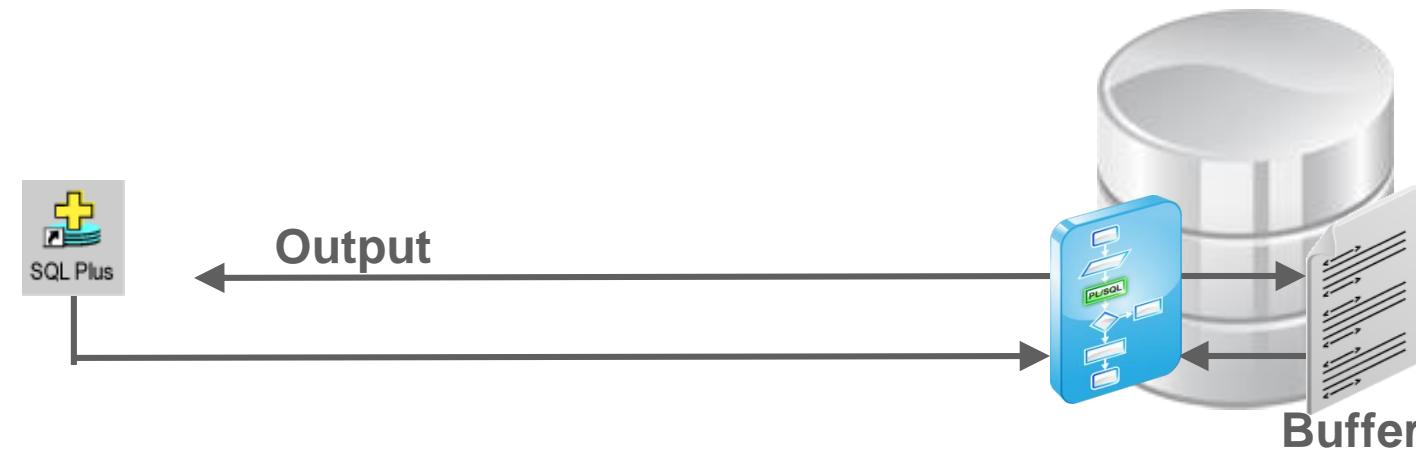
DBMS_SCHEDULER



DBMS_OUTPUT Package

The DBMS_OUTPUT package enables you to send messages from stored subprograms and triggers.

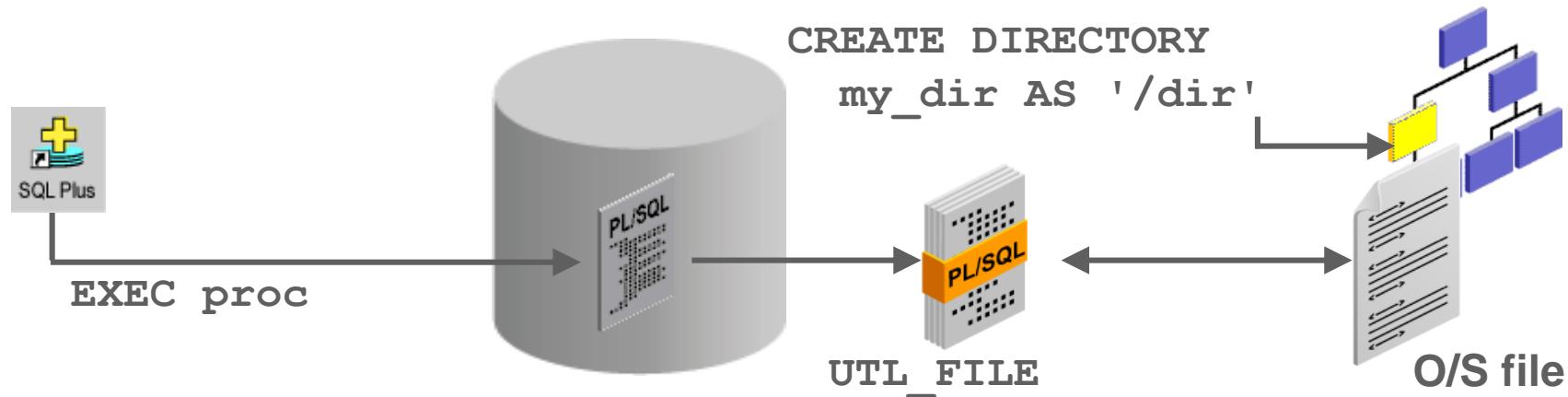
- PUT and PUT_LINE place text in the buffer.
- GET_LINE and GET_LINES read the buffer.
- Use SET SERVEROUTPUT ON to display messages in SQL*Plus. (The default is OFF.)



UTL_FILE Package

The UTL_FILE package extends PL/SQL programs to read and write operating system text files.

- It provides a restricted version of operating system stream file I/O for text files.
- It can access files in operating system directories defined by a CREATE DIRECTORY statement.



Summary

In this appendix, you should have learned how to:

- Identify a PL/SQL block
- Create subprograms
- List restrictions on calling functions from SQL expressions
- Review PL/SQL packages
- Use cursors
- Handle exceptions
- Use the `RAISE_APPLICATION_ERROR` procedure
- Identify Oracle-supplied packages

