

1

Introduction

ORACLE

Copyright © 2008, Oracle. All rights reserved.

Course Objectives

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

- Design PL/SQL packages and program units that execute efficiently
- Write code to interface with external applications and the operating system
- Create PL/SQL applications that use collections
- Write and tune PL/SQL code effectively to maximize performance
- Implement a virtual private database with fine-grained access control
- Write code to interface with large objects and use SecureFile LOBs
- Perform code analysis to find program ambiguities, test, trace, and profile PL/SQL code

ORACLE

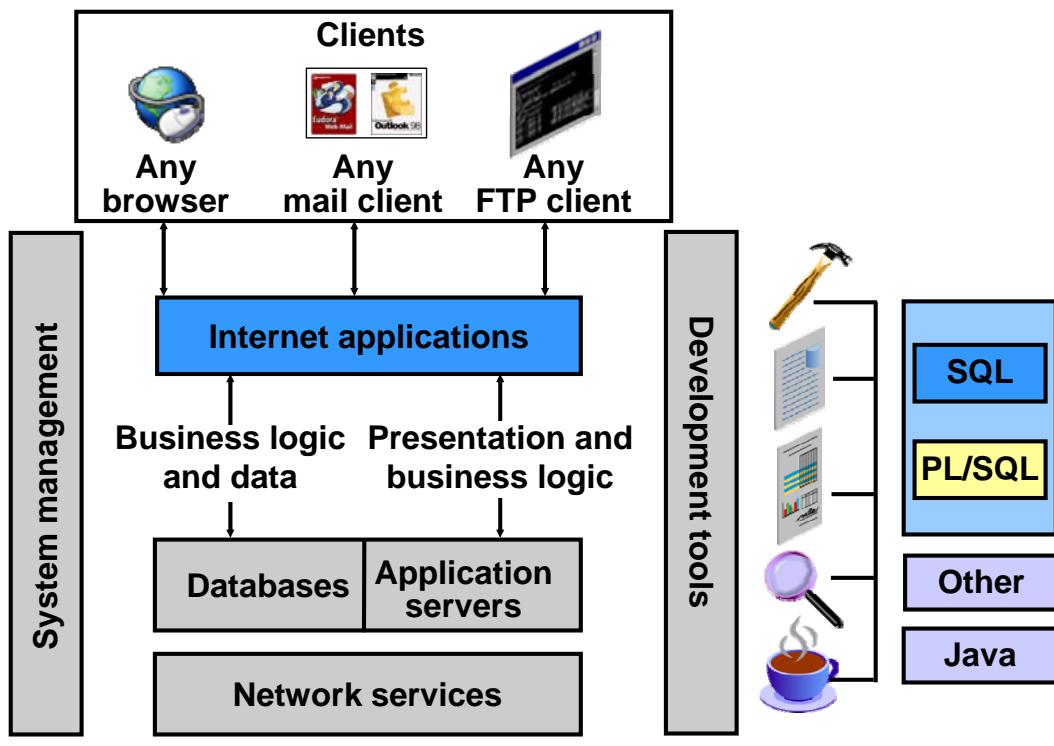
1 - 2

Copyright © 2008, Oracle. All rights reserved.

Course Objectives

In this course, you learn how to use the advanced features of PL/SQL in order to design and tune PL/SQL to interface with the database and other applications in the most efficient manner. Using the advanced features of program design, packages, cursors, extended interface methods, and collections, you learn how to write powerful PL/SQL programs. Programming efficiency, use of external C and Java routines, and fine-grained access are covered in this course.

Oracle Complete Solution



1 - 3

Copyright © 2008, Oracle. All rights reserved.

ORACLE

Oracle Complete Solution

The Oracle Internet Platform is built on three core components:

- Browser-based clients to process presentation
- Application servers to execute business logic and serve presentation logic to browser-based clients
- Databases to execute database-intensive business logic and serve data

Oracle offers a wide variety of the most advanced graphical user interface (GUI)-driven development tools to build business applications, as well as a large suite of software applications for many areas of business and industry. Stored procedures, functions, and packages can be written by using SQL, PL/SQL, Java, C, and Net languages. This course concentrates on the advanced features of PL/SQL.

Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Using SQL Developer
- Using SQL*Plus
- Identifying the tables, data, and tools used in this course



Course Agenda

- Day 1
 - Lesson 1: Introduction
 - Lesson 2: PL/SQL Review
 - Lesson 3: Designing PL/SQL Code
 - Lesson 4: Working with Collections
- Day 2
 - Lesson 4: Working with Collections
 - Lesson 5: Using Advanced Interface Methods
 - Lesson 6: Implementing Fine-Grained Access Control for VPD
 - Lesson 7: Manipulating Large Objects
 - Lesson 8: Administering SecureFile LOBs
 - Lesson 9: Performance and Tuning

ORACLE

1 - 5

Copyright © 2008, Oracle. All rights reserved.

Agenda

In this three-day course, you start with a review of PL/SQL concepts before progressing into the new and advanced topics. By the end of day one, you should have covered design considerations for your program units, and how to use collections effectively.

On day two, you learn how to use advanced interface methods to call C and Java code from your PL/SQL programs, how to implement and test fine-grained access control for virtual private databases, how to manipulate large objects programmatically through PL/SQL, how to administer the features of the new SecureFile LOB format of Database 11g, and how to tune PL/SQL code and deal with memory issues.

Course Agenda

- Day 3
 - Lesson 10: Improving Performance with Caching
 - Lesson 11: Analyzing PL/SQL Code
 - Lesson 12: Profiling and Tracing PL/SQL Code
 - Lesson 13: Safeguarding Your Code Against SQL Injection Attacks

ORACLE

1 - 6

Copyright © 2008, Oracle. All rights reserved.

Agenda (continued)

On day three, you learn how to improve performance by using Oracle database 11g caching techniques, how to write PL/SQL routines that analyze PL/SQL applications, how to profile and trace PL/SQL code, and how to protect your code from SQL injection security attacks.

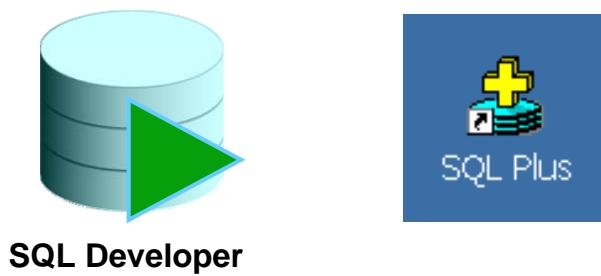
Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Using SQL Developer
- Using SQL*Plus
- Identifying the tables, data, and tools used in this course



Development Environments: Overview

- Introduction to SQL Developer
- SQL*Plus



ORACLE

1 - 8

Copyright © 2008, Oracle. All rights reserved.

PL/SQL Development Environments

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

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

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

Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- **Using SQL Developer**
- Using SQL*Plus
- Identifying the tables, data, and tools used in this course



What Is Oracle SQL Developer?

- Oracle SQL Developer is a free graphical tool that enhances productivity and simplifies database development tasks.
- You can connect to any target Oracle database schema by using the standard Oracle database authentication.
- You can use either SQL Developer or SQL*Plus in this course.



ORACLE

1 - 10

Copyright © 2008, Oracle. All rights reserved.

What Is Oracle SQL Developer?

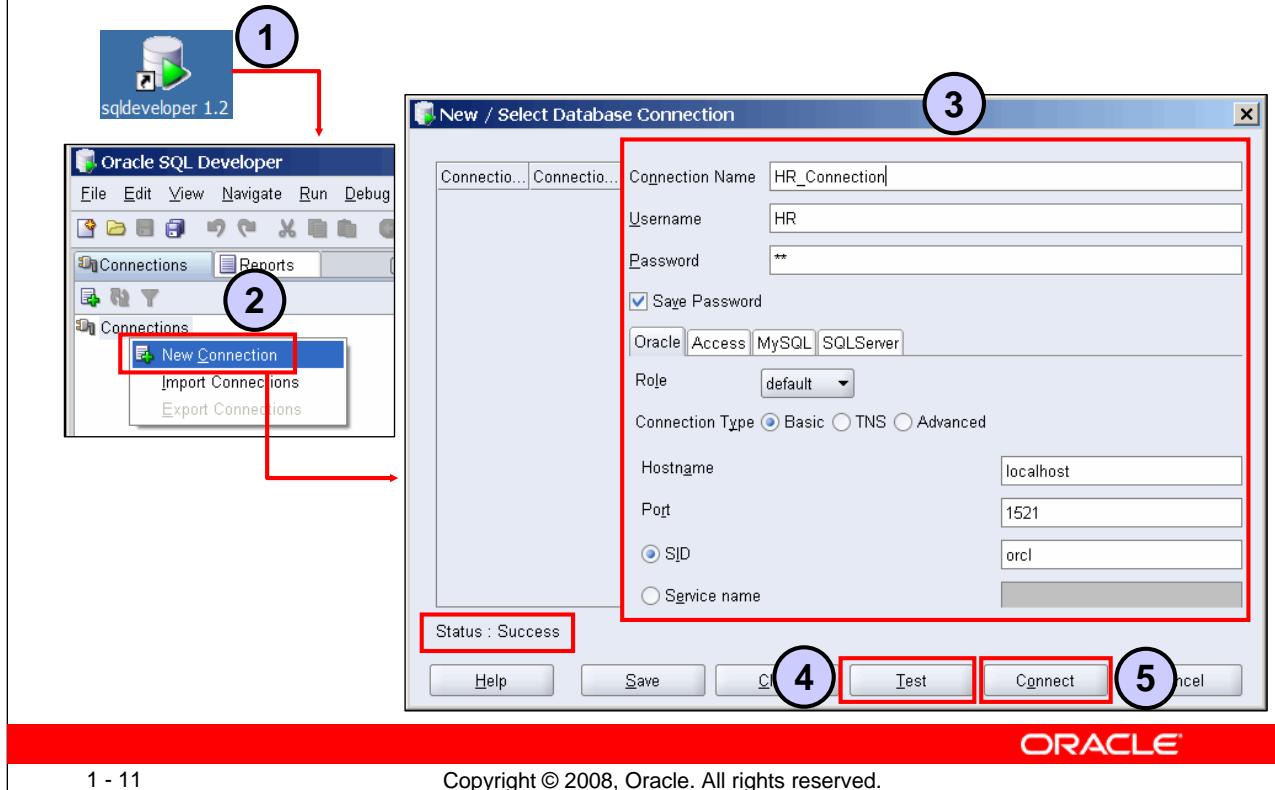
Oracle SQL Developer is a free graphical tool designed to improve your productivity and simplify the development of everyday database tasks. With just a few clicks, you can easily create and debug stored procedures, test SQL statements, and view optimizer plans.

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

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

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

Starting SQL Developer and Creating a Database Connection



Starting SQL Developer and Creating a Database Connection

To create a database connection, perform the following steps:

1. Double-click <your_path>\sqldeveloper\sqldeveloper.exe.
2. On the Connections tabbed page, right-click Connections and select **New Database Connection**.
3. Enter the connection name, username, password, host name, and SID for the database that you want to connect to.
4. Click Test to make sure that the connection is set correctly.
5. Click Connect.

On the basic tabbed page, at the bottom, enter the following options:

- **Hostname:** Host system for the Oracle database
- **Port:** Listener port
- **SID:** Database name
- **Service Name:** Network service name for a remote database connection

If you select the Save Password check box, the password is saved to an XML file. After you close the SQL Developer connection and open it again, you are not prompted for the password.

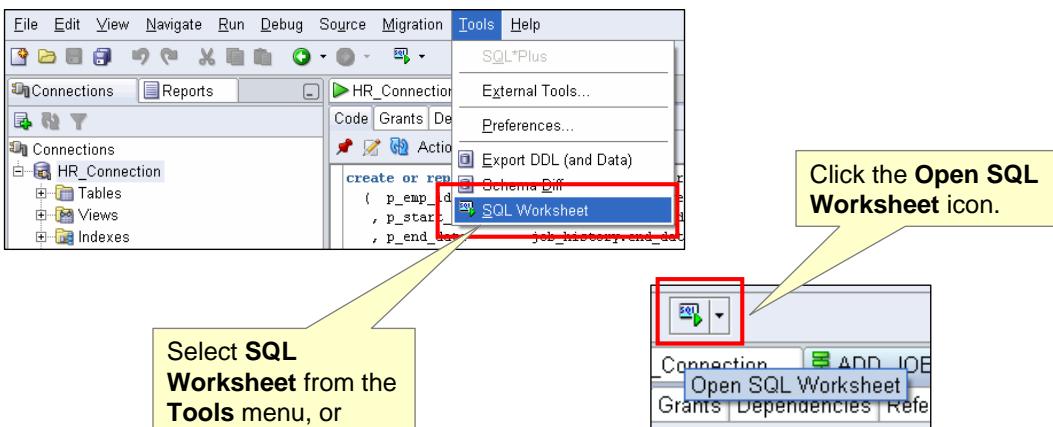
Creating Schema Objects

- You can create any schema object in SQL Developer by using one of the following methods:
 - Executing a SQL statement in the 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 DDL for adjustments such as creating a new object or editing an existing schema object

ORACLE

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.



Using the SQL Worksheet

When you connect to a database, a SQL worksheet window for that connection automatically opens. This example uses the HR_Connection. However, you use the OE_Connection and SH_Connection later in this course.

You can use the SQL worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. The SQL worksheet supports some SQL*Plus statements. However, SQL*Plus statements that are not supported by the SQL worksheet are ignored and not passed to the database.

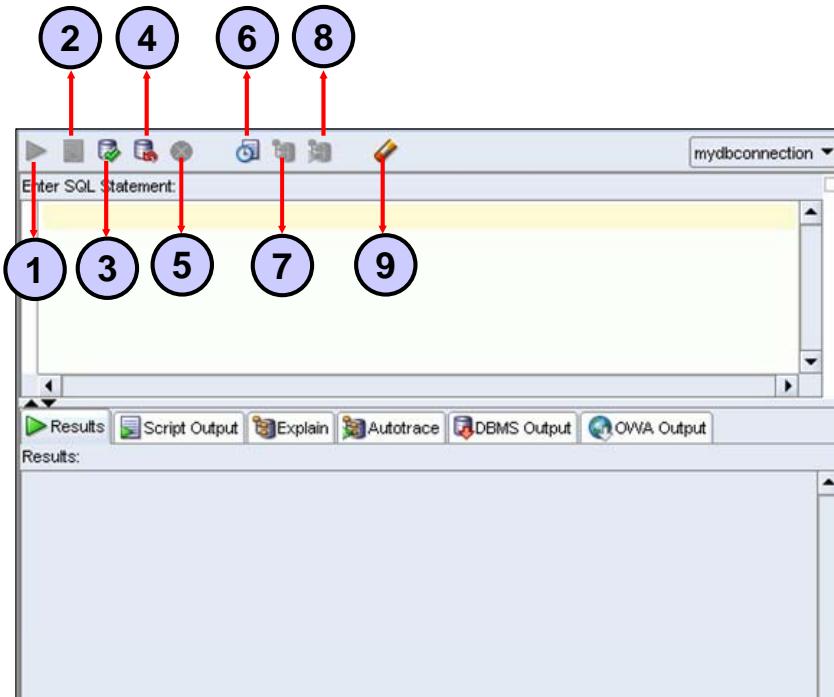
You can specify any actions that can be processed by the database connection associated with the worksheet, such as:

- Creating a table
- Inserting data
- Creating and editing a trigger
- Selecting data from a table
- Saving the selected data to a file

You can display a SQL worksheet by using one of the following options:

- Select **Tools > SQL Worksheet**.
- Click the **Open SQL Worksheet** icon.

Using the SQL Worksheet



ORACLE

1 - 14

Copyright © 2008, Oracle. All rights reserved.

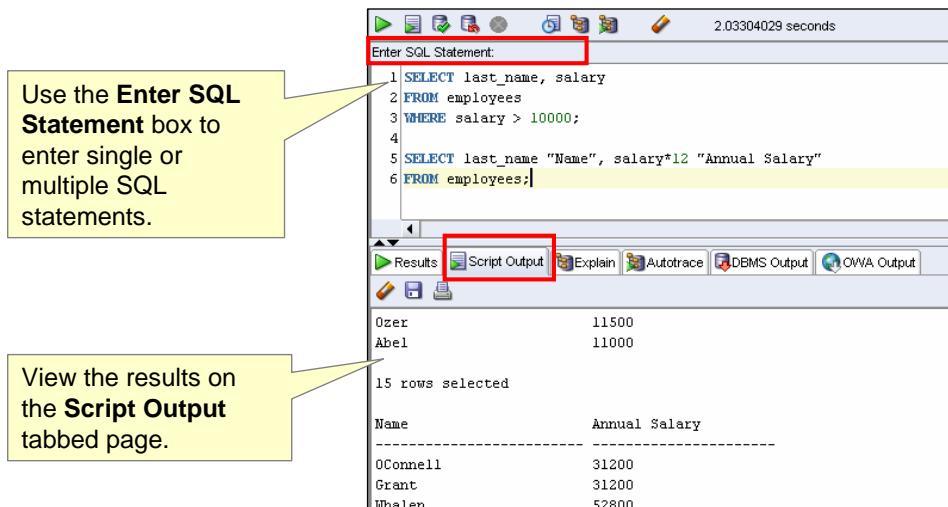
Using the SQL Worksheet (continued)

You may want to use the shortcut keys or icons to perform certain tasks, such as executing a SQL statement, running a script, or viewing the history of the SQL statements that you executed. You can use the SQL worksheet toolbar that contains icons to perform the following tasks:

1. **Execute Statement:** Executes the statement at the cursor in the Enter SQL Statement box. You can use bind variables in the SQL statements. You cannot use substitution variables.
2. **Run Script:** Executes all statements in the Enter SQL Statement box by using the Script Runner. You can use substitution variables in the SQL statements. You cannot use bind variables.
3. **Commit:** Writes changes to the database and ends the transaction.
4. **Rollback:** Discards changes to the database without writing them to the database, and ends the transaction.
5. **Cancel:** Stops the execution of statements that are being executed.
6. **SQL History:** Displays a dialog box with information about the SQL statements that you executed.
7. **Execute Explain Plan:** Generates the execution plan, which you can see by clicking the Explain tab.
8. **Autotrace:** Generates trace information for the statement.
9. **Clear:** Erases the statement or statements in the Enter SQL Statement box.

Executing SQL Statements

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



ORACLE

1 - 15

Copyright © 2008, Oracle. All rights reserved.

Executing SQL Statements

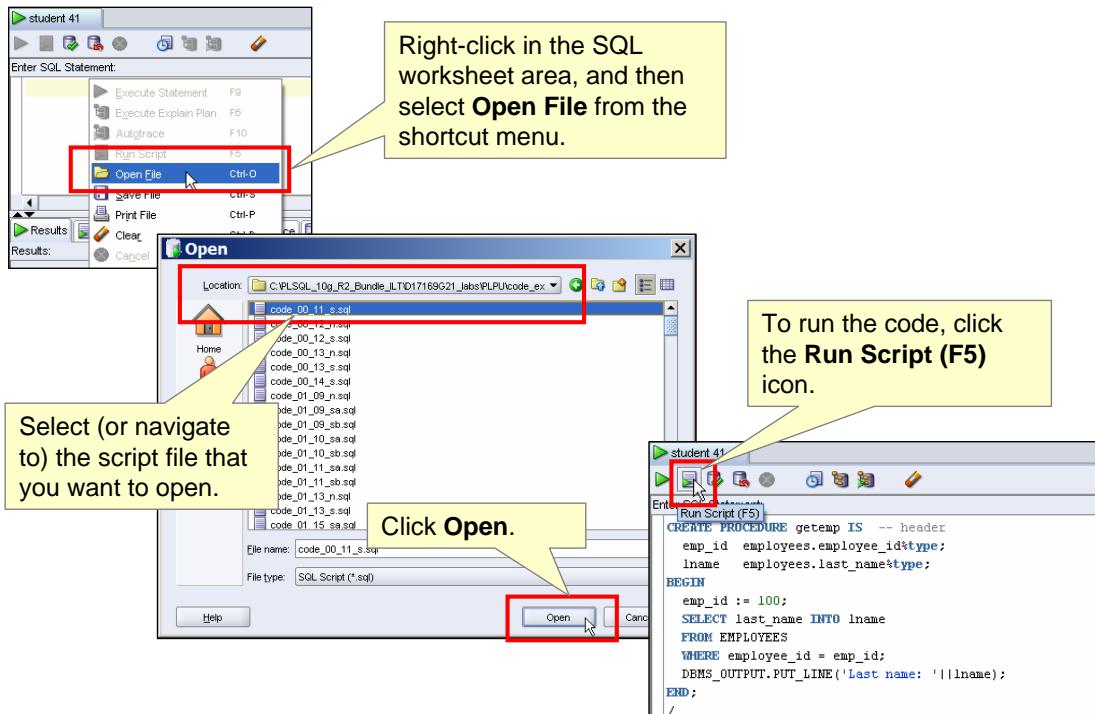
In the SQL worksheet, you can use the Enter SQL Statement box to enter a single statement or multiple SQL statements. For a single statement, the semicolon at the end is optional.

When you enter the statement, the SQL keywords are automatically highlighted. To execute a SQL statement, ensure that your cursor is within the statement and click the **Execute Statement** icon. Alternatively, you can press **F9**.

To execute multiple SQL statements and see the results, click the **Run Script** icon. Alternatively, you can press **F5**.

In the example in the slide, because there are multiple SQL statements, the first statement is terminated with a semicolon. The cursor is in the first statement, and therefore, when the statement is executed, results corresponding to the first statement are displayed in the Results box.

Executing Saved Script Files: Method 1

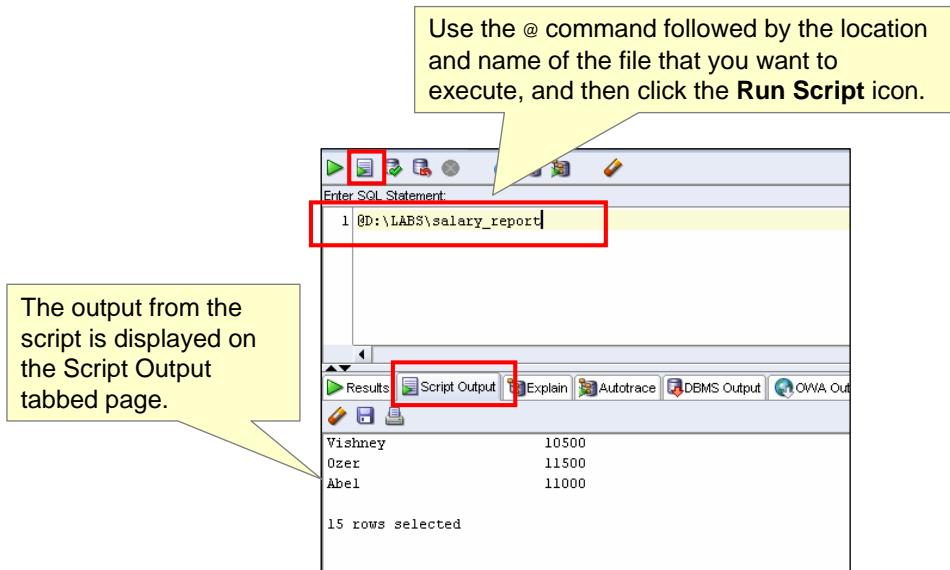


Executing Saved Script Files: Method 1

To open a script file and display the code in the SQL worksheet area, you can use one of the following methods:

1. Right-click in the SQL worksheet area, and then select **Open File** from the shortcut menu. The Open dialog box appears.
2. In the Open dialog box, select (or navigate to) the script file that you want to open.
3. Click Open. The code of the script file is displayed in the SQL worksheet area.
4. To run the code, click the **Run Script (F5)** icon on the SQL worksheet toolbar.

Executing Saved SQL Scripts: Method 2



ORACLE

Executing Saved Script Files: Method 2

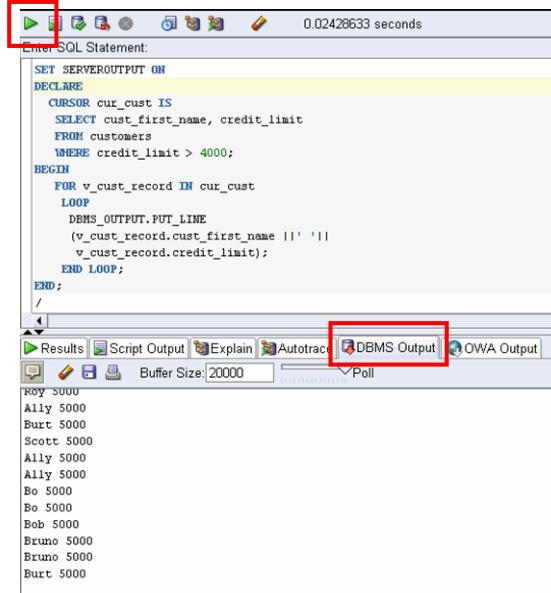
To run a saved SQL script, follow these steps:

1. In the Enter SQL Statement box, use the @ command followed by the location and name of the file that you want to run.
2. Click the **Run Script** icon.

The results from running the file are displayed on the Script Output tabbed page. You can also save the script output by clicking the Save icon on the Script Output tabbed page. The Windows File Save dialog box appears and you can identify a name and location for your file.

Creating an Anonymous Block

Create an anonymous block and display the output of the DBMS_OUTPUT package.



The screenshot shows the Oracle SQL Developer interface. A red box highlights the 'Execute Statement' icon (a green triangle) at the top left of the 'Enter SQL Statement' box. Another red box highlights the 'DBMS Output' tab at the bottom of the results pane. The code in the statement box is:

```
SET SERVEROUTPUT ON
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;
/
```

The results pane displays the output from the DBMS_OUTPUT package, with each row consisting of a first name followed by a space and a credit limit value. The output is:

Customer Name	Credit Limit
ROY SUUU	5000
Ally	5000
Burt	5000
Scott	5000
Ally	5000
Ally	5000
Bo	5000
Bo	5000
Bob	5000
Bruno	5000
Bruno	5000
Burt	5000

ORACLE

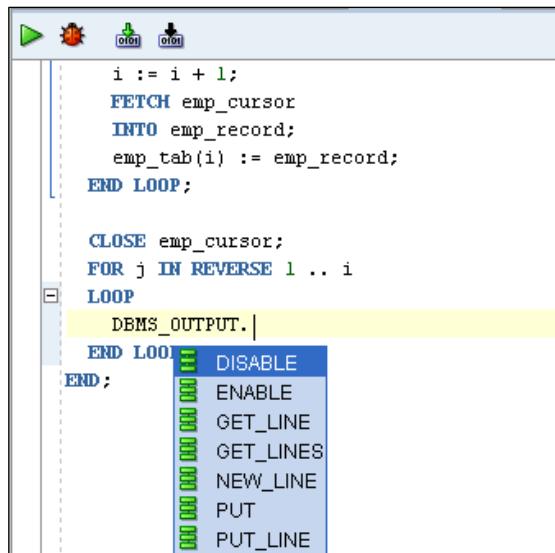
Creating an Anonymous Block

You can create an anonymous block (a unit of code without a name) and display the output of the DBMS_OUTPUT package. To create an anonymous block and view the results, perform the following steps:

1. Enter the PL/SQL code in the Enter SQL Statement box.
2. Click the **DBMS Output** tab. Click the **Enable DBMS Output** icon to set the server output ON.
3. Click the **Execute Statement** icon above the Enter SQL Statement box. Click the **DBMS Output** tab to see the results.

Editing the PL/SQL Code

Use the full-featured editor for PL/SQL program units.



ORACLE

1 - 19

Copyright © 2008, Oracle. All rights reserved.

Editing the PL/SQL Code

You may want to make changes to your PL/SQL code. SQL Developer includes a full-featured editor for PL/SQL program units. It includes customizable PL/SQL syntax highlighting in addition to common editor functions, such as:

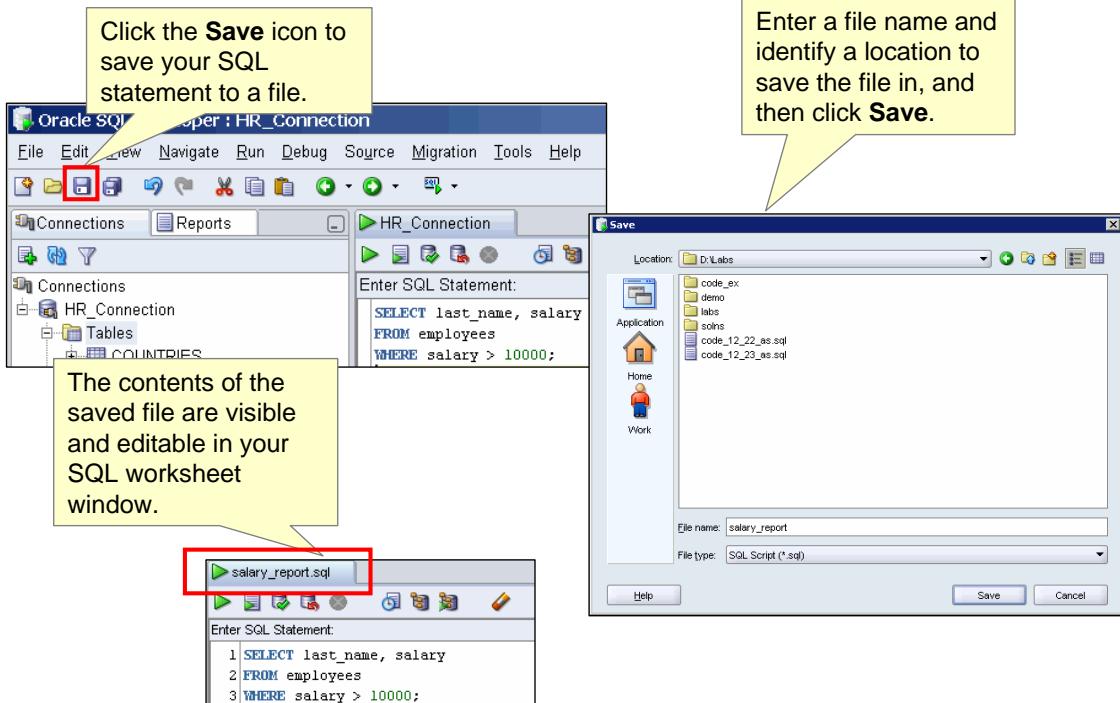
- Bookmarks
- Code Completion
- Code Folding
- Search and Replace

To edit the PL/SQL code, click the object name in the Connections Navigator, and then click the **Edit** icon. Optionally, double-click the object name to invoke the Object Definition page with its tabs and the Edit page. You can update only if you are on the Edit tabbed page.

The slide shows the Code Insight feature. For example, if you enter DBMS_OUTPUT, and then press Ctrl + Spacebar, you can select from a list of members of that package. Note that, by default, Code Insight is invoked automatically if you pause after entering a period (“.”) for more than one second.

When using the Code Editor to edit PL/SQL code, you can use Compile or Compile for Debug. Use the Compile for Debug option if you plan on using the SQL Developer Debugger. This option adds some debugging directives.

Saving SQL Scripts



Saving SQL Scripts

You can save your SQL statements from the SQL worksheet into a text file. To save the contents of the Enter SQL Statement box, perform the following steps:

1. Click the Save icon or use the **File > Save** menu option.
2. In the Windows Save dialog box, enter a file name and the location where you want to save the file.
3. Click Save.

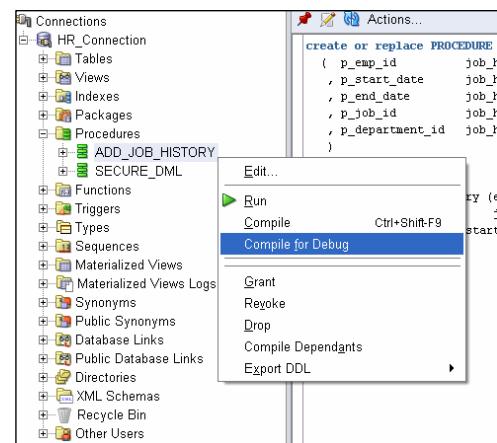
After you save the contents to a file, the Enter SQL Statement box displays a tabbed page of your file contents. You can have multiple files open simultaneously. Each file is displayed as a tabbed page.

Script Pathing

You can select a default path to look for scripts and to save scripts. Under **Tools > Preferences > Database > Worksheet Parameters**, enter a value in the **Select default path to look for scripts** field.

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.



ORACLE

1 - 21

Copyright © 2008, Oracle. All rights reserved.

Debugging Procedures and Functions

You can use the SQL Developer Debugger to debug PL/SQL procedures and functions. Using the Debug menu options, you can perform the following debugging tasks:

- **Find Execution Point** goes to the next execution point.
- **Resume** continues execution.
- **Step Over** bypasses the next method and goes to the next statement after the method.
- **Step Into** goes to the first statement in the next method.
- **Step Out** leaves the current method and goes to the next statement.
- **Step to End of Method** goes to the last statement of the current method.
- **Pause** halts execution but does not exit, thus allowing you to resume execution.
- **Terminate** halts and exits the execution. You cannot resume execution from this point; instead, to start running or debugging from the beginning of the function or procedure, click the Run or Debug icon on the Source tab toolbar.
- **Garbage Collection** removes invalid objects from the cache in favor of the more frequently accessed and more valid objects.

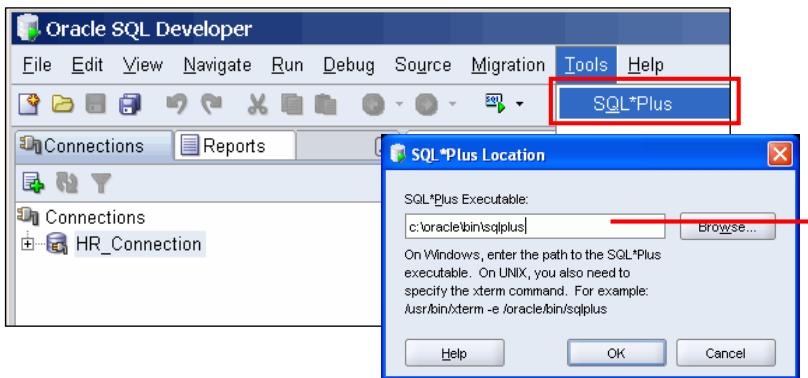
Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Using SQL Developer
- **Using SQL*Plus**
- Identifying the tables, data, and tools used in this course



Using SQL*Plus

- You can invoke the SQL*Plus command-line interface from SQL Developer.
- Close all SQL worksheets to enable the SQL*Plus menu option.



Provide the location of the `sqlplus.exe` file only for the first time you invoke SQL*Plus.

ORACLE

Using SQL*Plus

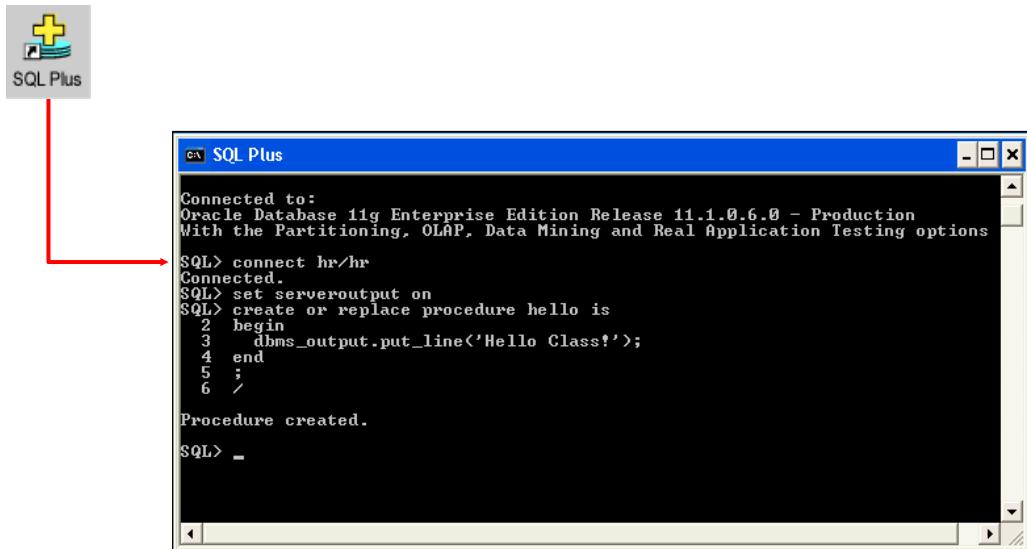
The SQL worksheet supports most of the SQL*Plus statements. SQL*Plus statements must be interpreted by the SQL worksheet before being passed to the database; any SQL*Plus statements that are not supported by the SQL worksheet are ignored and not passed to the database. To display the SQL*Plus command window, from the Tools menu, select **SQL*Plus**. To use the SQL*Plus command-line interface within SQL Developer, the system on which you are using SQL Developer must have an Oracle home directory or folder, with a SQL*Plus executable under that location. If the location of the SQL*Plus executable is not already stored in your SQL Developer preferences, you are asked to specify its location.

For example, some of the SQL*Plus statements that are not supported by SQL worksheet are:

- append
- archive
- attribute
- break

For a complete list of SQL*Plus statements that are either supported or not supported by the SQL worksheet, refer to the “SQL*Plus Statements Supported and Not Supported in SQL Worksheet” topic in the SQL Developer online Help.

Coding PL/SQL in SQL*Plus



ORACLE

1 - 24

Copyright © 2008, Oracle. All rights reserved.

Coding PL/SQL in SQL*Plus

You can also invoke Oracle SQL*Plus from the `sqlplus.exe` executable that is located in your Oracle home `/bin` directory. SQL*Plus is a command-line application that enables you to submit SQL statements and PL/SQL blocks for execution, and receive the results in an application or command window.

SQL*Plus is:

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

When coding PL/SQL subprograms by using SQL*Plus, remember the following:

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

Lesson Agenda

- Previewing the course agenda
- Describing the development environments
- Using SQL Developer
- Using SQL*Plus
- Identifying the tables, data, and tools used in this course



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 B contains more information about the sample schemas.

ORACLE

1 - 26

Copyright © 2008, Oracle. All rights reserved.

Tables Used in This Course

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

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

Each of these divisions is represented by a schema.

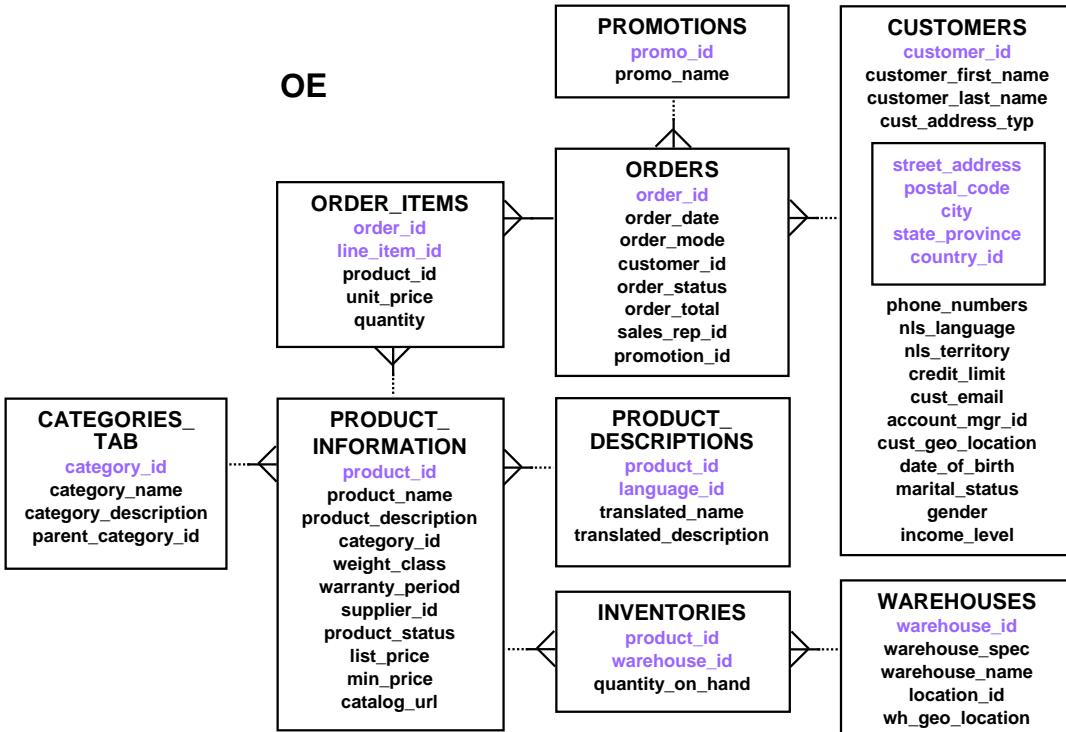
This course primarily uses the Order Entry (OE) sample schema.

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

All scripts necessary to create the OE schema reside in the \$ORACLE_HOME/demo/schema/order_entry folder.

All scripts necessary to create the HR schema reside in the \$ORACLE_HOME/demo/schema/human_resources folder.

The Order Entry Schema



ORACLE

The Order Entry (OE) Schema

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

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

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

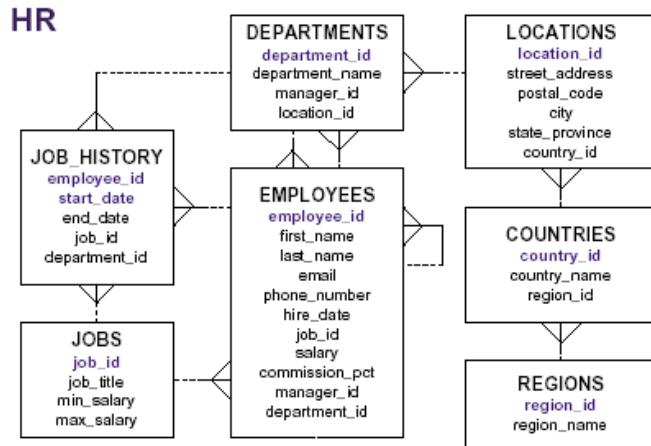
The Order Entry (OE) Schema (continued)

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

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

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

The Human Resources Schema



ORACLE

The Human Resources (HR) Schema

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

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

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

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

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

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



Summary

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

Practice 1 Overview: Getting Started

This practice covers the following topics:

- Reviewing the available SQL Developer resources
- Starting SQL Developer and creating new database connections and browsing the HR, OE, and SH tables
- Setting some SQL Developer preferences
- Executing SQL statements and an anonymous PL/SQL block by using SQL worksheet
- Accessing and bookmarking the Oracle Database 11g documentation and other useful Web sites



Practice 1: Overview

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

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

Practice 1

This is the first of many practices in this course. The solutions (if you require them) can be found in Appendix A. Practices are intended to cover most of the topics presented in the corresponding lesson.

In this practice, you review the available SQL Developer resources. You also learn about the user account that you use in this course. You start SQL Developer, create a new database connection, and browse your SH, HR, and OE tables. You also set some SQL Developer preferences, execute SQL statements, access and bookmark the Oracle Database 11g documentation and other useful Web sites that you can use in this course.

Identifying the Available SQL Developer Resources

1. Familiarize yourself with Oracle SQL Developer as needed by referring to Appendix C: Using SQL Developer.
2. Access the SQL Developer Home page that is available online at:
http://www.oracle.com/technology/products/database/sql_developer/index.html
3. Bookmark the page for easier future access.
4. Access the SQL Developer tutorial that is available online at:
<http://st-curriculum.oracle.com/tutorial/SQLDeveloper/index.htm>
5. Preview and experiment with the available links and demonstrations in the tutorial as needed, especially the Creating a Database Connection and Accessing Data links.

Creating and Using the New SQL Developer Database Connections

6. Start SQL Developer.
7. Create a database connection to SH using the following information:
 - a. Connection Name: sh_connection
 - b. Username: sh
 - c. Password: sh
 - d. Hostname: localhost
 - e. Port: 1521
 - f. SID: orcl
8. Test the new connection. If the Status is Success, connect to the database using this new connection.
 - a. Double-click the sh_connection icon on the Connections tabbed page.
 - b. Click the Test button in the New>Select Database Connection window. If the status is Success, click the Connect button.
9. Create a new database connection named hr_connection.
 - a. Right-click the sh_connection connection in the Object Navigation tree, and select the Properties menu option.
 - b. Enter hr_connection as the connection name and hr as the username and password, and click Save. This creates the new connection.
 - c. Repeat step 8 to test the new hr_connection connection.

Practice 1 (continued)

10. Repeat step 9 to create and test a new database connection named oe_connection. Enter oe as the database connection username and password.
11. Repeat step 9 to create and test a new database connection named sys_connection. Enter sys as the database connection username, oracle as the password, and SYSDBA as the role.

Browsing the HR, SH, and OE Schema Tables

12. Browse the structure of the EMPLOYEES table.
 - a. Expand the hr_connection connection by clicking the plus symbol next to it.
 - b. Expand the Tables icon by clicking the plus symbol next to it.
 - c. Display the structure of the EMPLOYEES table.
13. Browse the EMPLOYEES table and display its data.
14. Use the SQL worksheet to select the last names and salaries of all employees whose annual income is greater than \$10,000. Use both the Execute Statement (F9) and the Run Script (F5) icons to execute the SELECT statement. Review the results of both methods of executing the SELECT statements on the appropriate tabs.

Note: Take a few minutes to familiarize yourself with the data, or consult “Appendix B, Table Descriptions,” which provides the description and data for all tables in the HR, SH, and OE schemas that you will use in this course.

15. Create and execute a simple anonymous block that outputs “Hello World.”
 - a. Enable SET SERVEROUTPUT ON to display the output of the DBMS_OUTPUT package statements.
 - b. Use the SQL worksheet area to enter the code for your anonymous block.
 - c. Click the Run Script icon (F5) to run the anonymous block.
16. Browse the structure of the SALES table in the SH Schema connection and display its data.
 - a. Double-click the sh_connection connection.
 - b. Expand the Tables icon by clicking the plus symbol next to it.
 - c. Display the structure of the SALES table.
 - d. Browse the SALES table and display its data.
17. Browse the structure of the ORDERS table in the OE Schema and display its data.
 - a. Double-click the oe_connection connection.
 - b. Expand the Tables icon by clicking the plus symbol next to it.
 - c. Display the structure of the ORDERS table.
 - d. Browse the ORDERS table and display its data.

Accessing the Oracle Database 11g Release 1 Online Documentation Library

18. Access the Oracle Database 11g Release documentation Web page at:
<http://www.oracle.com/pls/db111/homepage>
19. Bookmark the page for easier future access.
20. Display the complete list of books available for Oracle Database 11g, Release 1.

Practice 1 (continued)

Accessing the Oracle Database 11g Release 1 Online Documentation Library (continued)

21. Make a note of the following documentation references that you will use in this course as needed:
- a. *Advanced Application Developer's Guide*
 - b. *New Features Guide*
 - c. *PL/SQL Language Reference*
 - d. *Oracle Database Reference*
 - e. *Oracle Database Concepts*
 - f. *SQL Developer User's Guide*
 - g. *SQL Language Reference Guide*
 - h. *SQL*Plus User's Guide and Reference*

PL/SQL Programming Concepts: Review

2

ORACLE

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

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



Objectives

PL/SQL supports various programming constructs. This lesson reviews the basic concept of PL/SQL programming. This lesson also reviews how to:

- Create subprograms
- Use cursors
- Handle exceptions
- Identify predefined Oracle server errors
- Manage dependencies

A quiz at the end of the lesson will assess your knowledge of PL/SQL.

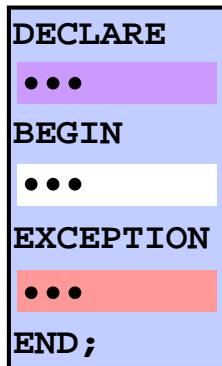
Note: The quiz is optional. Solutions to the quiz are provided in Appendix A.

Lesson Agenda

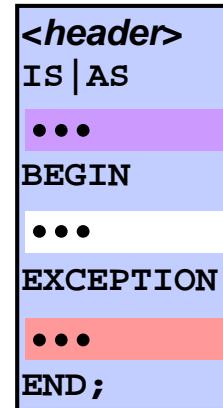
- Describing PL/SQL basics
- Listing restrictions on calling functions from SQL expressions
- Reviewing PL/SQL packages
- Identifying how explicit cursors are processed
- Handling exceptions
- Using the `raise_application_error` procedure
- Managing dependencies
- Using Oracle-supplied packages



PL/SQL Block Structure



**Anonymous
PL/SQL block**



**Stored
program unit**

ORACLE

PL/SQL Block Structure

An anonymous PL/SQL block structure consists of an optional `DECLARE` section, a mandatory `BEGIN-END` block, and an optional `EXCEPTION` section before the `END` statement of the main block.

A stored program unit has a mandatory header section. This section defines whether the program unit is a function, procedure, or a package, and contains the optional argument list and their modes. A stored program unit also has the other sections mentioned for the anonymous PL/SQL block. However, a stored program unit does not have an optional `DECLARE` section, but it does contain an `IS | AS` section that is mandatory and acts the same as the `DECLARE` section in an anonymous block.

Every PL/SQL construct is made from one or more blocks. These blocks can be entirely separate or nested within one another. Therefore, one block can represent a small part of another block, which in turn can be part of the whole unit of code.

Naming Conventions

Advantages of proper naming conventions:

- Easier to read
- Understandable
- Gives information about the functionality
- Easier to debug
- Ensures consistency
- Can improve performance



Naming Conventions

A proper naming convention makes the code easier to read and more understandable. It helps you understand the functionality of the identifier. If the code is written using proper naming conventions, you can easily find an error and rectify it. Most importantly, it ensures consistency among the code written by different developers.

The following table shows the naming conventions followed in this course:

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];
```



Procedures

A procedure is a named PL/SQL block that can accept parameters (sometimes referred to as *arguments*). Generally, you use a procedure to perform an action. A procedure is compiled and stored in the database as a schema object. Procedures promote reusability and maintainability.

Parameters are used to transfer data values to and from the calling environment and the procedure (or subprogram). Parameters are declared in the subprogram header, after the name and before the declaration section for local variables.

Parameters are subject to one of the three parameter-passing modes: IN, OUT, or IN OUT.

- An IN parameter passes a constant value from the calling environment into the procedure.
- An OUT parameter passes a value from the procedure to the calling environment.
- An IN OUT parameter passes a value from the calling environment to the procedure and a possibly different value from the procedure back to the calling environment using the same parameter.

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;
/
```



Procedure: Example

This reusable procedure has a parameter with a SELECT statement for getting average order totals for whatever customer value is passed in.

Note: If a developer drops a procedure, and then re-creates it, all applicable grants to execute the procedure are gone. Alternatively, the OR REPLACE command removes the old procedure and re-creates it but leaves all the grants against the said procedure in place. Thus, the OR REPLACE command is recommended wherever there is an existing procedure, function, or package; not merely for convenience, but also to protect granted privileges. If you grant object privileges, these privileges remain after you re-create the subprogram with the OR REPLACE option; otherwise, the privileges are not preserved.

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

A function is a named PL/SQL block that can accept parameters, be invoked, and return a value. In general, you use a function to compute a value. Functions and procedures are structured alike. A function must return a value to the calling environment, whereas a procedure returns zero or more values to its calling environment. Like a procedure, a function has a header, a declarative section, an executable section, and an optional exception-handling section. A function must have a RETURN clause in the header and at least one RETURN statement in the executable section, and must return a value in each exception handler to avoid the “ORA-06503: PL/SQL: Function returned without value” error.

Functions can be stored in the database as schema objects for repeated execution. A function that is stored in the database is referred to as a *stored function*. Functions can also be created on client-side applications.

Functions promote reusability and maintainability. When validated, they can be used in any number of applications. If the processing requirements change, only the function needs to be updated.

A function may also be called as part of a SQL expression or as part of a PL/SQL expression. In the context of a SQL expression, a function must obey specific rules to control side effects. In a PL/SQL expression, the function identifier acts like a variable whose value depends on the parameters passed to it.

Function: 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))
```

ORACLE

2 - 9

Copyright © 2008, Oracle. All rights reserved.

Function: Example

The `get_credit` function is created with a single input parameter and returns the credit limit as a number, as shown in the first code box in the slide. The `get_credit` function follows the common programming practice of assigning a returning value to a local variable and uses a single `RETURN` statement in the executable section of the code to return the value stored in the local variable. If your function has an exception section, it may also contain a `RETURN` statement.

Invoke a function as part of a PL/SQL expression, because the function returns a value to the calling environment. The second code box uses the SQL*Plus `EXECUTE` command to call the `DBMS_OUTPUT.PUT_LINE` procedure whose argument is the return value from the `get_credit` function. In this case, `DBMS_OUTPUT.PUT_LINE` is invoked first; it calls `get_credit` to calculate the credit limit of the customer with ID 101. The `credit_limit` value returned is supplied as the value of the `DBMS_OUTPUT.PUT_LINE` parameter, which then displays the result (if you have executed `SET SERVEROUTPUT ON`).

Note: The `%TYPE` attribute casts the data type to the type defined for the column in the table identified. You can use the `%TYPE` attribute as a data type specifier when declaring constants, variables, fields, and parameters.

A function must always return a value. The example does not return a value if a row is not found for a given ID. Ideally, create an exception handler to return a value as well.

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;
```

ORACLE

Ways to Execute Functions

If functions are designed thoughtfully, they can be powerful constructs. Functions can be invoked in the following ways:

- **As part of PL/SQL expressions:** You can use host or local variables to hold the returned value from a function. The first example in the slide uses a host variable and the second example uses a local variable in an anonymous block.
- **As a parameter to another subprogram:** The third example in the slide demonstrates this usage. The `get_credit` function, with all its arguments, is nested in the parameter required by the `DBMS_OUTPUT.PUT_LINE` procedure. This comes from the concept of nesting functions, as discussed in the *Oracle Database 10g: SQL Fundamentals I* course.
- **As an expression in a SQL statement:** The last example shows how a function can be used as a single-row function in a SQL statement.

Note: The restrictions and guidelines that apply to functions when used in a SQL statement are discussed in the next few pages.

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

ORACLE

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:
 - Parameters must be specified with positional notation
 - You must own the function or have the `EXECUTE` privilege

ORACLE

2 - 12

Copyright © 2008, Oracle. All rights reserved.

Restrictions on Calling Functions from SQL Expressions

The user-defined PL/SQL functions that are callable from SQL expressions must meet the following requirements:

- The function must be stored in the database.
- The function parameters must be input parameters and should be valid SQL data types.
- The functions must return data types that are valid SQL data types. They cannot be PL/SQL-specific data types such as `BOOLEAN`, `RECORD`, or `TABLE`. The same restriction applies to the parameters of the function.

The following restrictions apply when calling a function in a SQL statement:

- Parameters must use positional notation. Named notation is not supported.
- You must own or have the `EXECUTE` privilege on the function.

Other restrictions on a user-defined function include the following:

- It cannot be called from the `CHECK` constraint clause of a `CREATE TABLE` or `ALTER TABLE` statement.
- It cannot be used to specify a default value for a column.

Note: Only stored functions are callable from SQL statements. Stored procedures cannot be called unless invoked from a function that meets the preceding requirements.

Restrictions on Calling Functions from SQL Expressions

Functions called from:

- A SELECT statement cannot contain DML statements
- An UPDATE or DELETE statement on a table T cannot query or contain DML on the same table T
- SQL statements cannot 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.

ORACLE

2 - 13

Copyright © 2008, Oracle. All rights reserved.

Restrictions on Calling Functions from SQL Expressions (continued)

To execute a SQL statement that calls a stored function, the Oracle server must know whether the function is free of specific side effects. Side effects are unacceptable changes to database tables.

Additional restrictions also apply when a function is called in expressions of SQL statements. In particular, when a function is called from:

- A SELECT statement or a parallel UPDATE or DELETE statement, the function cannot modify a database table, unless the modification occurs in an autonomous transaction
- An INSERT... SELECT (but not an INSERT... VALUES), an UPDATE, or a DELETE statement, the function cannot query or modify a database table that was modified by that statement
- A SELECT, INSERT, UPDATE, or DELETE statement, the function cannot execute directly or indirectly through another subprogram or through a SQL transaction control statement such as:
 - A COMMIT or ROLLBACK statement
 - A session control statement (such as SET ROLE)
 - A system control statement (such as ALTER SYSTEM)
 - Any data definition language (DDL) statements (such as CREATE), because they are followed by an automatic commit

Note: The function *can* execute a transaction control statement if the transaction being controlled is autonomous.

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

ORACLE

2 - 15

Copyright © 2008, Oracle. All rights reserved.

PL/SQL Packages: Review

PL/SQL packages enable you to bundle related PL/SQL types, variables, data structures, exceptions, and subprograms into one container. For example, an Order Entry package can contain procedures for adding and deleting customers and orders, functions for calculating annual sales, and credit limit variables.

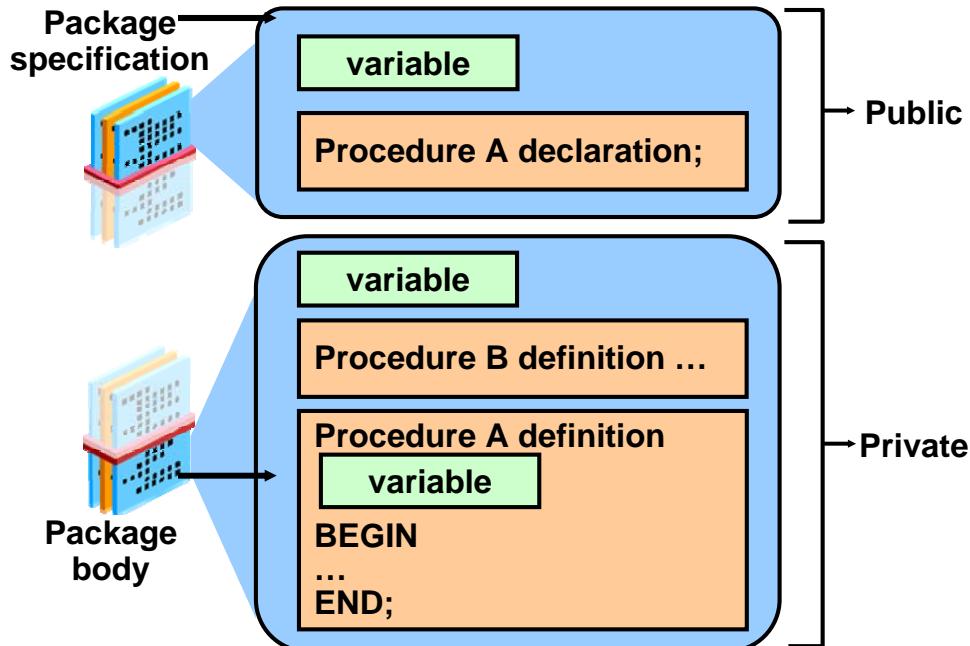
A package usually consists of two parts that are stored separately in the database:

- A specification
- A body (optional)

The package itself cannot be called, parameterized, or nested. After writing and compiling, the contents can be shared with many applications.

When a PL/SQL-packaged construct is referenced for the first time, the whole package is loaded into memory. However, subsequent access to constructs in the same package does not require disk I/O.

Components of a PL/SQL Package



Components of a PL/SQL Package

You create a package in two parts:

- The *package specification* is the interface to your applications. It declares the public types, variables, constants, exceptions, cursors, and subprograms that are available for use. The package specification may also include pragmas, which are directives to the compiler.
- The *package body* defines its own subprograms and must fully implement the subprograms that are declared in the specification part. The package body may also define PL/SQL constructs, such as object types, variables, constants, exceptions, and cursors.

Public components are declared in the package specification. The specification defines a public API for users of the package features and functionality. That is, public components can be referenced from any Oracle server environment that is external to the package.

Private components are placed in the package body but not referenced in the specification and can be referenced only by other constructs within the same package body. Alternatively, private components can reference the public components of the package.

Note: If a package specification does not contain subprogram declarations, there is no requirement for a package body.

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.

ORACLE

2 - 17

Copyright © 2008, Oracle. All rights reserved.

Creating the Package Specification

- To create packages, you declare all public constructs within the package specification.
 - Specify the OR REPLACE option if overwriting an existing package specification.
 - Initialize a variable with a constant value or formula within the declaration, if required; otherwise, the variable is initialized implicitly to NULL.
- The following are the definitions of items in the package syntax:
 - **package_name** specifies a name for the package that must be unique among objects within the owning schema. Including the package name after the END keyword is optional.
 - **public type and variable declarations** declares public variables, constants, cursors, exceptions, user-defined types, and subtypes.
 - **subprogram specifications** specifies the public procedure or function declarations.

Note: The package specification should contain procedure and function signatures terminated by a semicolon. The signature is every thing above IS | AS keywords. The implementation of a procedure or function that is declared in a package specification is done in the package body.

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.



Creating the Package Body

Create a package body to define and implement all public subprograms and the supporting private constructs. When creating a package body, perform the following:

- Specify the OR REPLACE option to overwrite a package body.
- Define the subprograms in an appropriate order. The basic principle is that you must declare a variable or subprogram before it can be referenced by other components in the same package body. It is common to see all private variables and subprograms defined first and the public subprograms defined last in the package body.
- The package body must complete the implementation for all procedures or functions declared in the package specification.

The following are the definitions of items in the package body syntax:

- ***package_name*** specifies a name for the package that must be the same as its package specification. Using the package name after the END keyword is optional.
- ***private type and variable declarations*** declares private variables, constants, cursors, exceptions, user-defined types, and subtypes.
- ***subprogram bodies*** specifies the full implementation of any private and/or public procedures or functions.
- **[*BEGIN initialization statements*]** is an optional block of initialization code that executes when the package is first referenced.

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

ORACLE

2 - 20

Copyright © 2008, Oracle. All rights reserved.

Cursor

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

Where Does Oracle Process SQL Statements?

The Oracle server allocates a private memory area, called the *context area*, to process SQL statements. The SQL statement is parsed and processed in this area. The information required for processing and the information retrieved after processing are stored in this area. Because this area is internally managed by the Oracle server, you have no control over this area. A cursor is a pointer to the context area. However, this cursor is an implicit cursor and is automatically managed by the Oracle server. When the executable block contains a SQL statement, an implicit cursor is created.

There are two types of cursors:

- **Implicit cursors:** Implicit cursors are created and managed by the Oracle server. You do not have access to them. The Oracle server creates such a cursor when it executes a SQL statement, such as `SELECT`, `INSERT`, `UPDATE`, or `DELETE`.

Cursor (continued)

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

Processing Explicit Cursors

The following three commands are used to process an explicit cursor:

- OPEN
- FETCH
- CLOSE

Alternatively, you can also use a cursor FOR loop.



Processing Explicit Cursors

You declare an explicit cursor when you need exact control over query processing. You use three commands to control a cursor:

- OPEN
- FETCH
- CLOSE

You initialize the cursor with the OPEN command, which recognizes the result set. Then, you execute the FETCH command repeatedly in a loop until all rows are retrieved. Alternatively, you can use a BULK COLLECT clause to fetch all rows at once. After the last row is processed, you release the cursor by using the CLOSE command.

Explicit Cursor Attributes

Every explicit cursor has the following attributes:

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



Cursor Attributes

When cursor attributes are appended to the cursors, they return useful information about the execution of the data manipulation language (DML) statement. The following are the four cursor attributes:

- ***cursor_name*%FOUND:** Returns TRUE if the last fetch returned a row; returns NULL before the first fetch from an OPEN cursor; returns FALSE if the last fetch failed to return a row
- ***cursor_name*%ISOPEN:** Returns TRUE if the cursor is open, otherwise returns FALSE
- ***cursor_name*%NOTFOUND:** Returns FALSE if the last fetch returned a row; returns NULL before the first fetch from an OPEN cursor; returns TRUE if the last fetch failed to return a row
- ***cursor_name*%ROWCOUNT:** Returns zero before the first fetch; after every fetch, returns the number of rows fetched so far

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.

ORACLE

2 - 24

Copyright © 2008, Oracle. All rights reserved.

Cursor FOR Loops

A cursor FOR loop processes rows in an explicit cursor. It is a shortcut, because the cursor is opened, a row is fetched once for each iteration in the loop, the loop exits when the last row is processed, and the cursor is closed automatically. The loop itself is terminated automatically at the end of the iteration where the last row is fetched.

In the syntax:

<i>record_name</i>	Is the name of the implicitly declared record
<i>cursor_name</i>	Is a PL/SQL identifier for the previously declared cursor

Guidelines

- Do not declare the record in the loop, because it is declared implicitly.
- Test the cursor attributes during the loop, if required.
- Supply the parameters for a cursor, if required, in parentheses following the cursor name in the FOR statement.

Cursor: Example

```
SET SERVEROUTPUT ON
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;
/
```



Cursor: Example

The example shows the use of a cursor FOR loop.

`cust_record` is the record that is implicitly declared. You can access the fetched data with this implicit record as shown in the slide.

Note: An `INTO` clause or a `FETCH` statement is not required because the `FETCH INTO` is implicit. The code does not have `OPEN` and `CLOSE` statements to open and close the cursor, respectively.

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

ORACLE

2 - 27

Copyright © 2008, Oracle. All rights reserved.

Handling Exceptions

An exception is an error in PL/SQL that is raised during the execution of a block. A block always terminates when PL/SQL raises an exception, but you can specify an exception handler to perform final actions before the block ends.

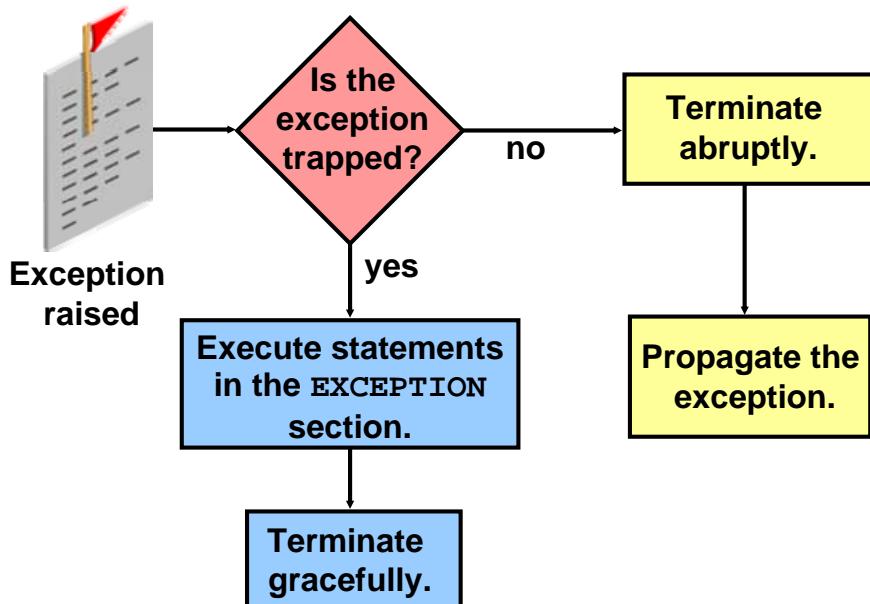
Methods for Raising an Exception

- An Oracle error occurs and the associated exception is raised automatically. For example, if the error ORA-01403 occurs when no rows are retrieved from the database in a SELECT statement, PL/SQL raises the NO_DATA_FOUND exception. These errors are converted into predefined exceptions.
- Depending on the business functionality that your program is implementing, you may have to explicitly raise an exception by issuing the RAISE statement within the block. The exception being raised may be either user-defined or predefined.
- There are some non-predefined Oracle errors. These errors are any standard Oracle errors that are not predefined. You can explicitly declare exceptions and associate them with the nonpredefined Oracle errors.

Methods for Handling an Exception

The third method in the slide for handling an exception involves trapping and propagating. It is often very important to be able to handle an exception after propagating it to the invoking environment, by issuing a simple RAISE statement.

Handling Exceptions



ORACLE

Handling Exceptions (continued)

Trapping an Exception

Include an EXCEPTION section in your PL/SQL program to trap exceptions. If the exception is raised in the executable section of the block, processing branches to the corresponding exception handler in the exception section of the block. If PL/SQL successfully handles the exception, the exception does not propagate to the enclosing block or to the calling environment. The PL/SQL block terminates successfully.

Propagating an Exception

If the exception is raised in the executable section of the block and there is no corresponding exception handler, the PL/SQL block terminates with failure and the exception is propagated to an enclosing block or to the calling environment. The calling environment can be any application, such as SQL*Plus, that invokes the PL/SQL program.

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;
/
```

ORACLE

2 - 29

Copyright © 2008, Oracle. All rights reserved.

Exceptions: Example

You have written PL/SQL blocks with a declarative section (beginning with the keyword DECLARE) and an executable section (beginning and ending with the keywords BEGIN and END, respectively). For exception handling, include another optional section called the EXCEPTION section. This section begins with the keyword EXCEPTION. If present, this is the last section in a PL/SQL block.

Examine the code in the slide to see the EXCEPTION section.

The output of this code is shown below:

Your select statement retrieved multiple rows. Consider using a cursor.

PL/SQL procedure successfully completed.

When the exception is raised, the control shifts to the EXCEPTION section and all statements in the specified EXCEPTION section are executed. The PL/SQL block terminates with normal, successful completion. Only one exception handler is executed.

Note the SELECT statement in the executable block. That statement requires that a query must return *only* one row. If multiple rows are returned, a “too many rows” exception is raised. If no rows are returned, a “no data found” exception is raised. The block of code in the slide tests for the “too many rows” exception.

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 Errors

You can reference predefined Oracle server errors by using its predefined name within the corresponding exception-handling routine.

For a complete list of predefined exceptions, see the *PL/SQL User's Guide and Reference*.

Note: PL/SQL declares predefined exceptions in the STANDARD package.

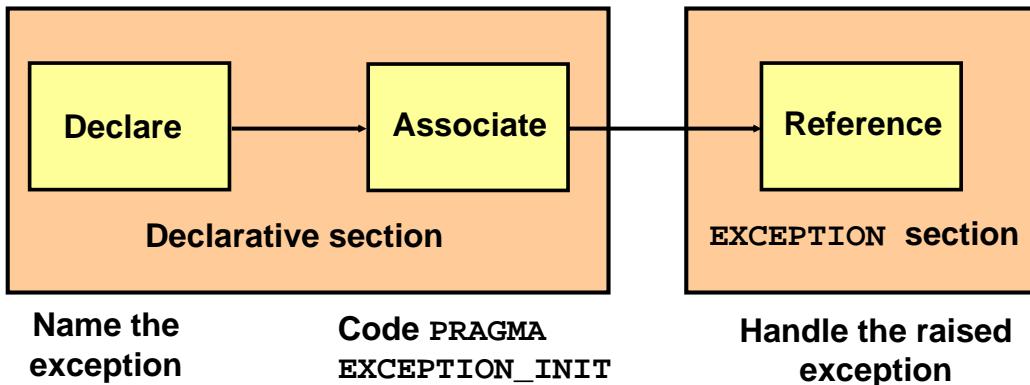
Predefined Oracle Server Errors (continued)

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	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	PL/SQL program issued a database call without being connected to the Oracle server.
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.

Predefined Oracle Server Errors (continued)

Exception Name	Oracle Server Error Number	Description
STORAGE_ERROR	ORA-06500	PL/SQL ran out of memory or memory is corrupted.
SUBSCRIPT_BEYOND_COUNT	ORA-06533	Referenced a nested table or varray element by using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	ORA-06532	Referenced a nested table or varray element by using an index number that is outside the legal range (for example -1).
SYS_INVALID_ROWID	ORA-01410	The conversion of a character string into a universal ROWID 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



ORACLE

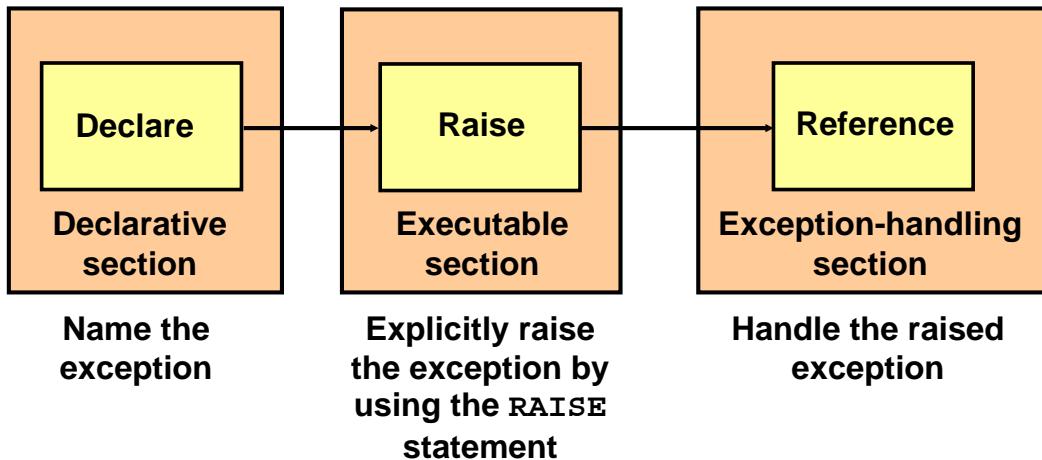
Trapping Non-Predefined Oracle Server Errors

Non-predefined exceptions are similar to predefined exceptions; however, they are not defined as PL/SQL exceptions in the Oracle server. They are standard Oracle errors. You can create exceptions with standard Oracle errors by using the PRAGMA EXCEPTION_INIT function. Such exceptions are called nonpredefined exceptions.

You can trap a nonpredefined Oracle server error by declaring it first. The declared exception is raised implicitly. In PL/SQL, PRAGMA EXCEPTION_INIT instructs the compiler to associate an exception name with an Oracle error number. This allows you to refer to any internal exception by name and to write a specific handler for it.

Note: PRAGMA (also called pseudoinstructions) is the keyword that signifies that the statement is a compiler directive, which is not processed when the PL/SQL block is executed. Rather, it directs the PL/SQL compiler to interpret all occurrences of the exception name within the block as the associated Oracle server error number.

Trapping User-Defined Exceptions



ORACLE

Trapping User-Defined Exceptions

With PL/SQL, you can define your own exceptions. You define exceptions depending on the requirements of your application. For example, you may prompt the user to enter a department number.

Define an exception to deal with error conditions in the input data. Check whether the department number exists. If it does not, you may have to raise the user-defined exception. PL/SQL exceptions must be:

- Declared in the declarative section of a PL/SQL block
- Raised explicitly with RAISE statements
- Handled in the EXCEPTION section

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



The RAISE_APPLICATION_ERROR Procedure

Syntax:

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

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

ORACLE

2 - 36

Copyright © 2008, Oracle. All rights reserved.

The RAISE_APPLICATION_ERROR Procedure

Use the `raise_application_error` procedure to communicate a predefined exception interactively by returning a nonstandard error code and error message. With `raise_application_error`, you can report errors to your application and avoid returning unhandled exceptions.

In the syntax:

`error_number` Is a user-specified number for the exception between –20,000 and –20,999 (this is not an Oracle-defined exception number).

`message` Is the user-specified message for the exception. It is a character string up to 2,048 bytes long.

`TRUE | FALSE` Is an optional Boolean parameter. (If TRUE, the error is placed on the stack of previous errors. If FALSE, the default, the error replaces all previous errors.)

The 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

ORACLE

2 - 37

Copyright © 2008, Oracle. All rights reserved.

The RAISE_APPLICATION_ERROR Procedure (continued)

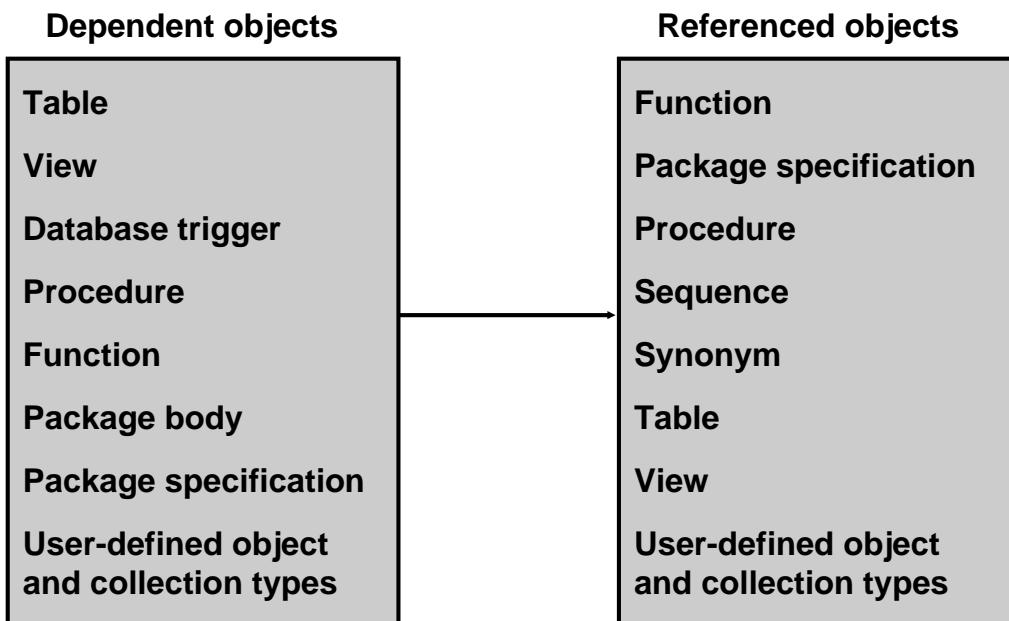
The `raise_application_error` procedure can be used in either the executable section or the exception section of a PL/SQL program, or both. The returned error is consistent with how the Oracle server processes a predefined, nonpredefined, or user-defined error. The error number and message are displayed to the user.

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



ORACLE

2 - 39

Copyright © 2008, Oracle. All rights reserved.

Dependencies

Some objects reference other objects as part of their definitions. For example, a stored procedure could contain a SELECT statement that selects columns from a table. For this reason, the stored procedure is called a *dependent object*, whereas the table is called a *referenced object*.

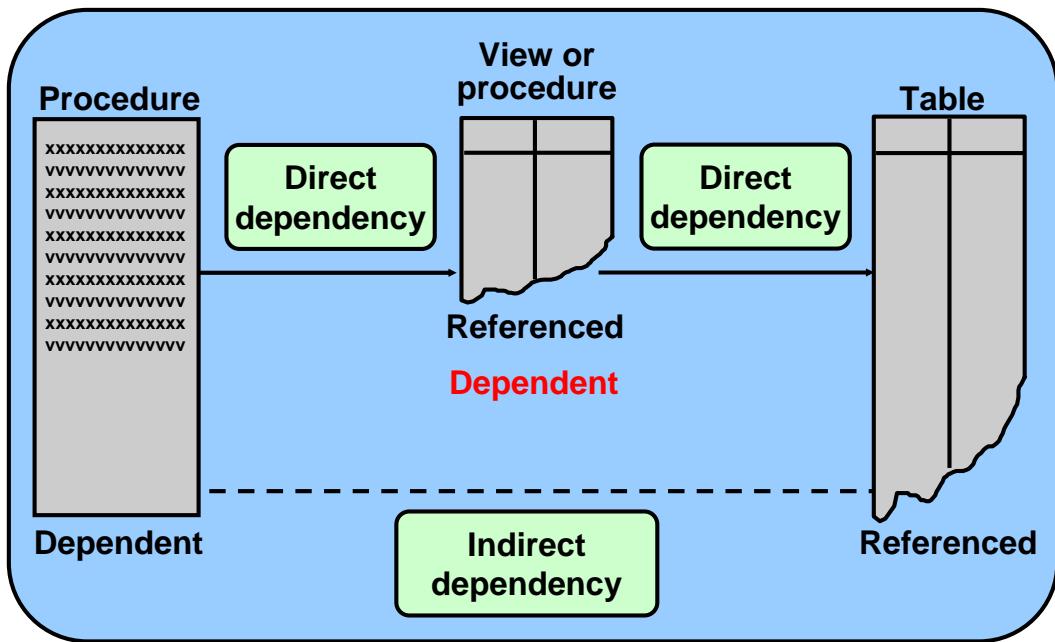
Dependency Issues

If you alter the definition of a referenced object, dependent objects may or may not continue to work properly. For example, if the table definition is changed, a procedure may or may not continue to work without an error.

The Oracle server automatically records dependencies among objects. To manage dependencies, all schema objects have a status (valid or invalid) that is recorded in the data dictionary, and you can view the status in the USER_OBJECTS data dictionary view.

Status	Significance
VALID	The schema object was compiled and can be immediately used when referenced.
INVALID	The schema object must be compiled before it can be used.

Dependencies



Dependencies (continued)

A procedure or function can directly or indirectly (through an intermediate view, procedure, function, or packaged procedure or function) reference the following objects:

- Tables
- Views
- Sequences
- Procedures
- Functions
- Packaged procedures or functions

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')
```



Displaying Direct and Indirect Dependencies

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

Example

1. Make sure that the `utldtree.sql` script was executed. This script is located in the `$ORACLE_HOME/rdbms/admin` folder.
2. Populate the `DEPTREE_TEMPTAB` table with information for a particular referenced object by invoking the `DEPTREE_FILL` procedure. There are three parameters for this procedure:

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

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.



Using Oracle-Supplied Packages

Packages are provided with the Oracle server to allow either of the following:

- PL/SQL access to certain SQL features
- The extension of the functionality of the database

You can use the functionality provided by these packages when creating your application, or you may simply want to use these packages as ideas when you create your own stored procedures.

Most of the standard packages are created by running catproc.sql.

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



Some of the Oracle-Supplied Packages

The list of PL/SQL packages provided with an Oracle database grows with the release of new versions. It would be impossible to cover the exhaustive set of packages and their functionality in this course. For more information, refer to the *PL/SQL Packages and Types Reference 10g* (previously known as the *PL/SQL Supplied Packages Reference*).

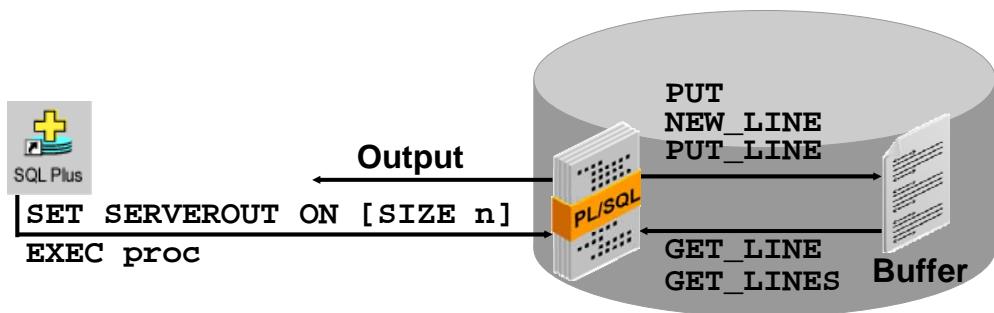
The following is a brief description of some listed packages:

- The DBMS_ALERT package supports asynchronous notification of database events. Messages or alerts are sent on a COMMIT command.
- The DBMS_LOCK package is used to request, convert, and release locks through Oracle Lock Management services.
- The DBMS_SESSION package enables programmatic use of the ALTER SESSION SQL statement and other session-level commands.
- The DBMS_OUTPUT package provides debugging and buffering of text data.
- The HTP package writes HTML-tagged data into database buffers.
- The UTL_FILE package enables reading and writing of operating system text files.
- The UTL_MAIL package enables composing and sending of email messages.
- The DBMS_SCHEDULER package enables scheduling and automated execution of PL/SQL blocks, stored procedures, and external procedures or executables.

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.)



ORACLE

2 - 45

Copyright © 2008, Oracle. All rights reserved.

DBMS_OUTPUT Package

The DBMS_OUTPUT package sends textual messages from any PL/SQL block into a buffer in the database. The procedures provided by the package include:

- PUT to append text from the procedure to the current line of the line output buffer
- NEW_LINE to place an end-of-line marker in the output buffer
- PUT_LINE to combine the action of PUT and NEW_LINE; to trim leading spaces
- GET_LINE to retrieve the current line from the buffer into a procedure variable
- GET_LINES to retrieve an array of lines into a procedure-array variable
- ENABLE/DISABLE to enable or disable calls to the DBMS_OUTPUT procedures

The buffer size can be set by using:

- The SIZE *n* option appended to the SET SERVEROUTPUT ON command, where *n* is between 2,000 (the default) and 1,000,000 (1 million characters)
- An integer parameter between 2,000 and 1,000,000 in the ENABLE procedure

Practical Uses

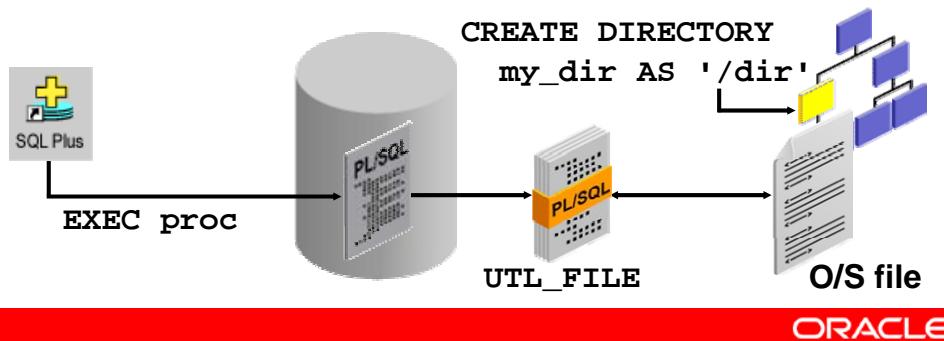
- You can output results to the window for debugging purposes.
- You can trace the code execution path for a function or procedure.
- You can send messages between subprograms and triggers.

Note: There is no mechanism to flush output during the execution of a procedure.

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.



2 - 46

Copyright © 2008, Oracle. All rights reserved.

UTL_FILE Package

The Oracle-supplied UTL_FILE package is used to access text files in the operating system of the database server. The database provides read and write access to specific operating system directories by using:

- A CREATE DIRECTORY statement that associates an alias with an operating system directory. The database directory alias can be granted the READ and WRITE privileges to control the type of access to files in the operating system. For example:

```
CREATE DIRECTORY my_dir AS '/temp/my_files';
GRANT READ, WRITE ON DIRECTORY my_dir TO public;
```

This approach of using the directory alias created by the CREATE DIRECTORY statement does not require the database to be restarted. The operating system directories specified should be accessible to and on the same machine as the database server processes. The path (directory) names may be case-sensitive for some operating systems.

Note: The DBMS_LOB package can be used to read binary files on the operating system.

Summary

In this lesson, you should have learned how to:

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



Summary

This lesson reviewed some basic PL/SQL concepts, such as:

- PL/SQL block structure
- Subprograms
- Cursors
- Exceptions
- Oracle-supplied packages

The quiz on the following pages is designed to test and review your PL/SQL knowledge. This knowledge is necessary as a baseline for the subsequent chapters to build upon.

Practice 2: Overview

This practice covers the review of the following topics:

- PL/SQL basics
- Cursor basics
- Exceptions
- Dependencies



Practice 2: Overview

In this practice, you test and review your PL/SQL knowledge. This knowledge is necessary as a base line for the subsequent chapters to build upon.

For answers to the questions in this practice, see Appendix A, “Practice Solutions.”

Practice 2: PL/SQL Knowledge Quiz

The questions are designed as a refresher. Use the space provided for your answers. If you do not know the answer, go on to the next question. For solutions to this quiz, see Appendix A.

PL/SQL Basics

1. Which are the four key areas of the basic PL/SQL block? What happens in each area?
2. What is a variable and where is it declared?
3. What is a constant and where is it declared?
4. What are the different modes for parameters and what does each mode do?
5. How does a function differ from a procedure?
6. Which are the two main components of a PL/SQL package?
 - a. In what order are they defined?
 - b. Are both required?
7. How does the syntax of a SELECT statement used within a PL/SQL block differ from a SELECT statement issued in SQL*Plus?
8. What is a record?
9. What is an index by table?
10. How are loops implemented in PL/SQL?
11. How is branching logic implemented in PL/SQL?

Practice 2: PL/SQL Knowledge Quiz (continued)

Cursor Basics

12. What is an explicit cursor?

13. Where do you define an explicit cursor?

14. Name the five steps for using an explicit cursor.

15. What is the syntax used to declare a cursor?

16. What does the FOR UPDATE clause do within a cursor definition?

17. Which command opens an explicit cursor?

18. Which command closes an explicit cursor?

19. Name five implicit actions that a cursor FOR loop provides.

20. Describe what the following cursor attributes do:

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

Practice 2: PL/SQL Knowledge Quiz (continued)

Exceptions

21. An exception occurs in your PL/SQL block, which is enclosed in another PL/SQL block. What happens to this exception?
22. An exception handler is mandatory within a PL/SQL subprogram. (True/False)
23. What syntax do you use in the exception handler area of a subprogram?
24. How do you code for a NO_DATA_FOUND error?
25. Name three types of exceptions.
26. To associate an exception identifier with an Oracle error code, what pragma would you use and where?
27. How do you explicitly raise an exception?
28. What types of exceptions are implicitly raised?
29. What does the `raise_application_error` procedure do?

Practice 2: PL/SQL Knowledge Quiz (continued)

Dependencies

30. Which objects can a procedure or function directly reference?

31. Which are the two statuses that a schema object can have and where are they recorded?

32. The Oracle server automatically recompiles invalid procedures when they are called from the same _____. To avoid compile problems with remote database calls, you can use the _____ model instead of the timestamp model.

33. Which data dictionary contains information on direct dependencies?

34. What script would you run to create the deptree and ideptree views?

35. What does the deptree_fill procedure do and what are the arguments that you need to provide?

Oracle-Supplied Packages

36. What does the dbms_output package do?

37. How do you write “This procedure works.” from within a PL/SQL program by using dbms_output?

38. What does dbms_sql do and how does this compare with Native Dynamic SQL?

Designing PL/SQL Code



ORACLE®

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

- Identify guidelines for cursor design
- Use cursor variables
- Create subtypes based on the existing types for an application



Objectives

This lesson discusses several concepts that apply to the designing of PL/SQL program units. This lesson explains how to:

- Design and use cursor variables
- Describe the predefined data types
- Create subtypes based on existing data types for an application

Lesson Agenda

- Identifying guidelines for cursor design
- Using cursor variables
- Creating subtypes based on existing types



Guidelines for Cursor Design

Fetch into a record when fetching from a cursor.

```
DECLARE
    CURSOR cur_cust IS
        SELECT customer_id, cust_last_name, cust_email
        FROM customers
        WHERE credit_limit = 1200;
        v_cust_record    cur_cust%ROWTYPE;

BEGIN
    OPEN cur_cust;
    LOOP
        FETCH cur_cust INTO v_cust_record;
    ...

```

ORACLE®

3 - 4

Copyright © 2008, Oracle. All rights reserved.

Guidelines for Cursor Design

When fetching from a cursor, fetch into a record. This way you do not need to declare individual variables, and you reference only the values that you want to use. Additionally, you can automatically use the structure of the SELECT column list.

Guidelines for Cursor Design

Create cursors with parameters.

```
CREATE OR REPLACE PROCEDURE cust_pack
  (p_crd_limit_in NUMBER, p_acct_mgr_in NUMBER)
IS
  v_credit_limit NUMBER := 1500;
  CURSOR cur_cust
    (p_crd_limit NUMBER, p_acct_mgr NUMBER)
  IS
    SELECT customer_id, cust_last_name, cust_email
    FROM customers
    WHERE credit_limit = p_crd_limit
      AND account_mgr_id = p_acct_mgr;
BEGIN
  OPEN cur_cust(p_crd_limit_in, p_acct_mgr_in);
  ...
  CLOSE cur_cust;
  ...
  OPEN cur_cust(v_credit_limit, 145);
  ...
END;
```

ORACLE®

3 - 5

Copyright © 2008, Oracle. All rights reserved.

Guidelines for Cursor Design (continued)

Whenever you need to use a cursor in multiple places with different values for the WHERE clause, create parameters for your cursor. Parameters increase the flexibility and reusability of cursors, because you can pass different values to the WHERE clause when you open a cursor, rather than hard-code a value for the WHERE clause.

Additionally, parameters help avoid scoping problems, because the result set for the cursor is not tied to a specific variable in a program. You can define a cursor at a higher level and use it in any subblock with variables defined in the local block.

Guidelines for Cursor Design

Reference implicit cursor attributes immediately after the SQL statement executes.

```
BEGIN
    UPDATE customers
        SET credit_limit = p_credit_limit
        WHERE customer_id = p_cust_id;
    → get_avg_order(p_cust_id); -- procedure call
    IF SQL%NOTFOUND THEN
        ...
    END IF;
```

ORACLE®

3 - 6

Copyright © 2008, Oracle. All rights reserved.

Guidelines for Cursor Design (continued)

If you are using an implicit cursor and reference a SQL cursor attribute, make sure you reference it immediately after a SQL statement is executed. This is because SQL cursor attributes are set on the result of the most recently executed SQL statement. The SQL statement can be executed in another program. Referencing a SQL cursor attribute immediately after a SQL statement executes ensures that you are dealing with the result of the correct SQL statement.

In the example in the slide, you cannot rely on the value of SQL%NOTFOUND for the UPDATE statement, because it is likely to be overwritten by the value of another SQL statement in the get_avg_order procedure. To ensure accuracy, the cursor attribute function SQL%NOTFOUND needs to be called immediately after the data manipulation language (DML) statement:

```
DECLARE
    v_flag BOOLEAN;
BEGIN
    UPDATE customers
        SET credit_limit = p_credit_limit
        WHERE customer_id = p_cust_id;
    v_flag := SQL%NOTFOUND
    get_avg_order(p_cust_id); -- procedure call
    IF v_flag THEN
        ...
    END IF;
```

Guidelines for Cursor Design

Simplify coding with cursor FOR loops.

```
CREATE OR REPLACE PROCEDURE cust_pack
  (p_crd_limit_in NUMBER, p_acct_mgr_in NUMBER)
IS
  v_credit_limit NUMBER := 1500;
  CURSOR cur_cust
    (p_crd_limit NUMBER, p_acct_mgr NUMBER)
  IS
    SELECT customer_id, cust_last_name, cust_email
    FROM customers
    WHERE credit_limit = p_crd_limit
      AND account_mgr_id = p_acct_mgr;
BEGIN
  FOR cur_rec IN cur_cust (p_crd_limit_in, p_acct_mgr_in)
  LOOP          -- implicit open and fetch
    ...
    END LOOP;    -- implicit close
    ...
END;
```

ORACLE®

3 - 7

Copyright © 2008, Oracle. All rights reserved.

Guidelines for Cursor Design (continued)

Whenever possible, use cursor FOR loops that simplify coding. Cursor FOR loops reduce the volume of code that you need to write to fetch data from a cursor and also reduce the chances of introducing loop errors in your code.

A cursor FOR loop automatically handles the open, fetch, and close operations, and defines a record type that matches the cursor definition. After it processes the last row, the cursor is closed automatically. If you do not use a cursor FOR loop, forgetting to close your cursor results in increased memory usage.

Guidelines for Cursor Design

- Close a cursor when it is no longer needed.
- Use column aliases in cursors for calculated columns fetched into records declared with %ROWTYPE.

```
CREATE OR REPLACE PROCEDURE cust_list
IS
  CURSOR cur_cust IS
    SELECT customer_id, cust_last_name, credit_limit*1.1
    FROM customers;
    cust_record cur_cust%ROWTYPE;
BEGIN
  OPEN cur_cust;
  LOOP
    FETCH cur_cust INTO cust_record;
    DBMS_OUTPUT.PUT_LINE('Customer ' ||
      cust_record.cust_last_name || ' wants credit ' ||
      cust_record.(credit_limit * 1.1)); ←
    EXIT WHEN cur_cust%NOTFOUND;
  END LOOP;
  ...

```

Use col. alias

ORACLE®

3 - 8

Copyright © 2008, Oracle. All rights reserved.

Guidelines for Cursor Design (continued)

- If you no longer need a cursor, close it explicitly. If your cursor is in a package, its scope is not limited to any particular PL/SQL block. The cursor remains open until you explicitly close it. An open cursor takes up memory space and continues to maintain row-level locks, if created with the FOR UPDATE clause, until a commit or rollback. Closing the cursor releases memory. Ending the transaction by committing or rolling back releases the locks. Along with a FOR UPDATE clause, you can also use a WHERE CURRENT OF clause with the DML statements inside the FOR loop. This automatically performs a DML transaction for the current row in the cursor's result set, thereby improving performance.

Note: It is a good programming practice to explicitly close your cursors. Leaving cursors open can generate an exception, because the number of cursors allowed to remain open within a session is limited.

- Make sure that you use column aliases in your cursor for calculated columns that you fetch into a record declared with a %ROWTYPE declaration. You would also need column aliases if you want to reference the calculated column in your program.

The code in the slide does not compile successfully, because it lacks a column alias for the calculation `credit_limit*1.1`. After you give it an alias, use the same alias later in the code to make a reference to the calculation.

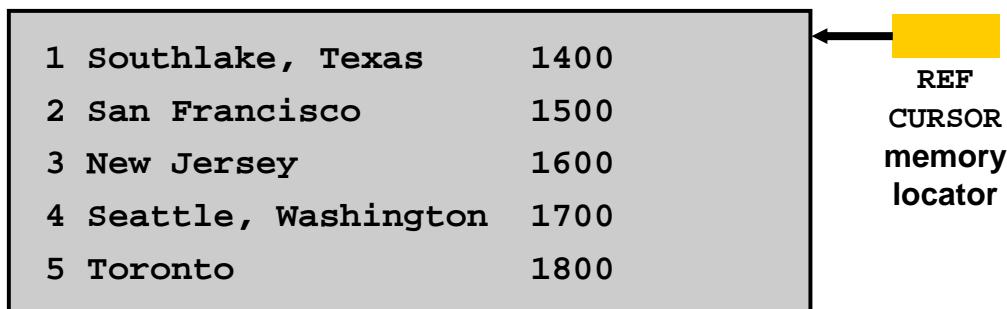
Lesson Agenda

- Identifying guidelines for cursor design
- **Using cursor variables**
- Creating subtypes based on existing types



Cursor Variables: Overview

Memory



ORACLE®

Cursor Variables: Overview

Like a cursor, a cursor variable points to the current row in the result set of a multiple-row query. Cursor variables, however, are like C pointers: they hold the memory location of an item instead of the item itself. Thus, cursor variables differ from cursors the way constants differ from variables. A cursor is static, a cursor variable is dynamic. In PL/SQL, a cursor variable has a **REF CURSOR** data type, where REF stands for reference, and CURSOR stands for the class of the object.

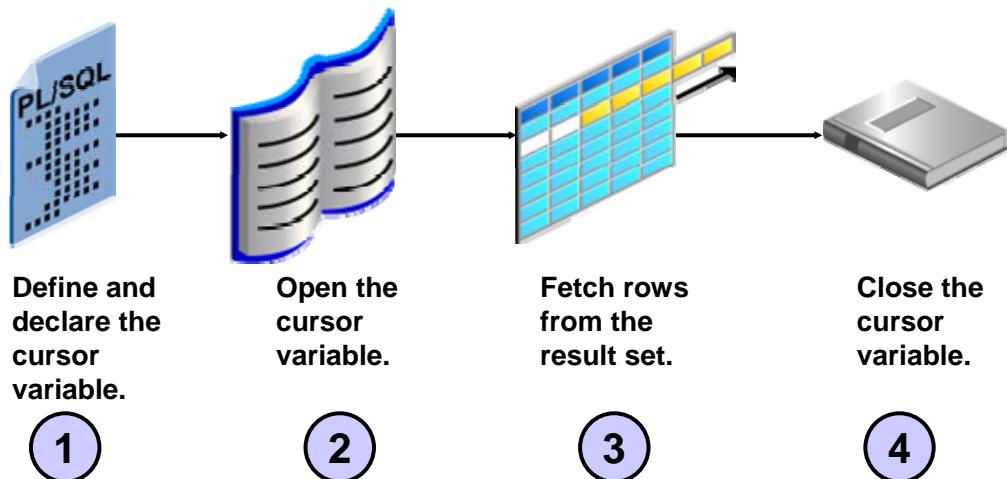
Using Cursor Variables

To execute a multiple-row query, the Oracle server opens a work area called a “cursor” to store the processing information. To access the information, you either explicitly name the work area, or you use a cursor variable that points to the work area. Whereas a cursor always refers to the same work area, a cursor variable can refer to different work areas. Therefore, cursors and cursor variables are not interoperable.

An explicit cursor is static and is associated with one SQL statement. A cursor variable can be associated with different statements at run time.

Primarily, you use a cursor variable to pass a pointer to query result sets between PL/SQL-stored subprograms and various clients, such as a Developer Forms application. None of them owns the result set. They simply share a pointer to the query work area that stores the result set.

Working with Cursor Variables



Working with Cursor Variables

There are four steps for handling a cursor variable. The next few sections contain detailed information about each step.

Strong Versus Weak REF CURSOR Variables

- Strong REF CURSOR:
 - Is restrictive
 - Specifies a RETURN type
 - Associates only with type-compatible queries
 - Is less error prone
- Weak REF CURSOR:
 - Is nonrestrictive
 - Associates with any query
 - Is very flexible

ORACLE®

3 - 12

Copyright © 2008, Oracle. All rights reserved.

Strong Versus Weak REF CURSOR Variables

REF CURSOR types can be strong (restrictive) or weak (nonrestrictive). A strong REF CURSOR type definition specifies a return type; a weak definition does not. PL/SQL enables you to associate a strong type only with type-compatible queries, whereas a weak type can be associated with any query. This makes strong REF CURSOR types less prone to error, but weak REF CURSOR types more flexible.

In the following example, the first definition is strong, whereas the second is weak:

```
DECLARE
    TYPE rt_cust IS REF CURSOR RETURN customers%ROWTYPE;
    TYPE rt_general_purpose IS REF CURSOR;
    ...

```

Step 1: Defining a REF CURSOR Type

Define a REF CURSOR type:

```
TYPE ref_type_name IS REF CURSOR  
[RETURN return_type];
```

- *ref_type_name* is a type specified in subsequent declarations.
- *return_type* represents a record type.
- *RETURN* keyword indicates a strong cursor.

```
DECLARE  
  TYPE rt_cust IS REF CURSOR  
    RETURN customers%ROWTYPE;  
  ...
```

ORACLE®

3 - 13

Copyright © 2008, Oracle. All rights reserved.

Step 1: Defining a Cursor Variable

To create a cursor variable, you first define a REF CURSOR type, and then declare a variable of that type.

Defining the REF CURSOR type:

```
TYPE ref_type_name IS REF CURSOR [RETURN return_type];
```

where: *ref_type_name* is a type specified in subsequent declarations.

return_type represents a row in a database table.

The REF keyword indicates that the new type is to be a pointer to the defined type. The *return_type* is a record type indicating the types of the select list that are eventually returned by the cursor variable. The return type must be a record type.

Example

```
DECLARE  
  TYPE rt_cust IS REF CURSOR RETURN customers%ROWTYPE;  
  ...
```

Step 1: Declaring a Cursor Variable

Declare a cursor variable of a cursor type:

```
cursor_variable_name ref_type_name;
```

- *cursor_variable_name* is the name of the cursor variable.
- *ref_type_name* is the name of a REF CURSOR type.

```
DECLARE
  TYPE rt_cust IS REF CURSOR
    RETURN customers%ROWTYPE;
  cv_cust rt_cust;
```

ORACLE®

3 - 14

Copyright © 2008, Oracle. All rights reserved.

Declaring a Cursor Variable

After the cursor type is defined, declare a cursor variable of that type.

```
cursor_variable_name ref_type_name;
```

where: *cursor_variable_name* is the name of the cursor variable.

ref_type_name is the name of the REF CURSOR type.

Cursor variables follow the same scoping and instantiation rules as all other PL/SQL variables.

In the following example, you declare the cursor variable *cv_cust*.

Step 1:

```
DECLARE
  TYPE ct_cust IS REF CURSOR RETURN customers%ROWTYPE;
  cv_cust rt_cust;
```

Step 1: Declaring a REF CURSOR Return Type

Options:

- Use %TYPE and %ROWTYPE.
- Specify a user-defined record in the RETURN clause.
- Declare the cursor variable as the formal parameter of a stored procedure or function.

ORACLE®

3 - 15

Copyright © 2008, Oracle. All rights reserved.

Step 1: Declaring a REF CURSOR Return Type

The following are other examples of cursor variable declarations:

- Use %TYPE and %ROWTYPE to provide the data type of a record variable:

```
DECLARE
    cust_rec customers%ROWTYPE; --a recd variable based on a row
    TYPE rt_cust IS REF CURSOR RETURN cust_rec%TYPE;
    cv_cust      rt_cust; --cursor variable
```

- Specify a user-defined record in the RETURN clause:

```
DECLARE
    TYPE cust_rec_typ IS RECORD
        (custno      NUMBER(4),
         custname    VARCHAR2(10),
         credit      NUMBER(7,2));
    TYPE rt_cust IS REF CURSOR RETURN cust_rec_typ;
    cv_cust      rt_cust;
```

- Declare a cursor variable as the formal parameter of a stored procedure or function:

```
DECLARE
    TYPE rt_cust IS REF CURSOR RETURN customers%ROWTYPE;
    PROCEDURE use_cust_cur_var(cv_cust IN OUT rt_cust)
    IS ...
```

Step 2: Opening a Cursor Variable

- Associate a cursor variable with a multiple-row SELECT statement.
- Execute the query.
- Identify the result set:

```
OPEN cursor_variable_name
      FOR select_statement;
```

- *cursor_variable_name* is the name of the cursor variable.
- *select_statement* is the SQL SELECT statement.

ORACLE®

3 - 16

Copyright © 2008, Oracle. All rights reserved.

Step 2: Opening a Cursor Variable

Other OPEN-FOR statements can open the same cursor variable for different queries. You do not need to close a cursor variable before reopening it. You must note that when you reopen a cursor variable for a different query, the previous query is lost.

In the following example, the packaged procedure declares a variable used to select one of several alternatives in an IF THEN ELSE statement. When called, the procedure opens the cursor variable for the chosen query.

```
CREATE OR REPLACE PACKAGE cust_data
IS
    TYPE rt_cust IS REF CURSOR RETURN customers%ROWTYPE;
    PROCEDURE open_cust_cur_var(cv_cust IN OUT rt_cust,
                                p_your_choice IN NUMBER);
END cust_data;
/
```

Step 2: Opening a Cursor Variable (continued)

```
CREATE OR REPLACE PACKAGE BODY cust_data
IS
    PROCEDURE open_cust_cur_var(cv_cust IN OUT rt_cust,
                                p_your_choice IN NUMBER)
    IS
    BEGIN
        IF p_your_choice = 1 THEN
            OPEN cv_cust FOR SELECT * FROM customers;
        ELSIF p_your_choice = 2 THEN
            OPEN cv_cust FOR SELECT * FROM customers
                WHERE credit_limit > 3000;
        ELSIF p_your_choice = 3 THEN
            ...
        END IF;
    END open_cust_cur_var;
END cust_data;
/
```

Step 3: Fetching from a Cursor Variable

- Retrieve rows from the result set one at a time.

```
FETCH cursor_variable_name  
  INTO variable_name1  
    [,variable_name2, . . .]  
  / record_name;
```

- The return type of the cursor variable must be compatible with the variables named in the `INTO` clause of the `FETCH` statement.

ORACLE®

3 - 18

Copyright © 2008, Oracle. All rights reserved.

Step 3: Fetching from a Cursor Variable

The `FETCH` statement retrieves rows from the result set one at a time. PL/SQL verifies that the return type of the cursor variable is compatible with the `INTO` clause of the `FETCH` statement. For each query column value returned, there must be a type-compatible variable in the `INTO` clause. Also, the number of query column values must equal the number of variables. In case of a mismatch in number or type, the error occurs at compile time for strongly typed cursor variables and at run time for weakly typed cursor variables.

Note: When you declare a cursor variable as the formal parameter of a subprogram that fetches from a cursor variable, you must specify the `IN` (or `IN OUT`) mode. If the subprogram also opens the cursor variable, you must specify the `IN OUT` mode.

Step 4: Closing a Cursor Variable

- Disable a cursor variable.
- The result set is undefined.

```
CLOSE cursor_variable_name;
```

- Accessing the cursor variable after it is closed raises the predefined exception `INVALID_CURSOR`.

ORACLE®

3 - 19

Copyright © 2008, Oracle. All rights reserved.

Step 4: Closing a Cursor Variable

The `CLOSE` statement disables a cursor variable, after which the result set is undefined. The syntax is:

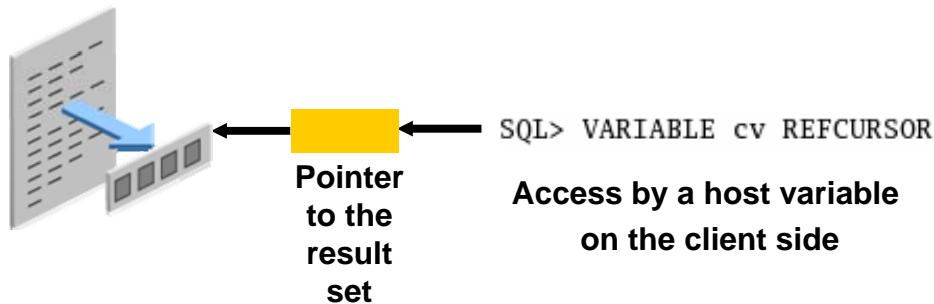
```
CLOSE cursor_variable_name;
```

In the following example, the cursor is closed when the last row is processed:

```
...
LOOP
  FETCH cv_cust INTO cust_rec;
  EXIT WHEN cv_cust%NOTFOUND;
  ...
END LOOP;
CLOSE cv_cust;
...
```

Passing Cursor Variables as Arguments

You can pass query result sets among PL/SQL-stored subprograms and various clients.



ORACLE®

3 - 20

Copyright © 2008, Oracle. All rights reserved.

Passing Cursor Variables as Arguments

Cursor variables are very useful for passing query result sets between PL/SQL-stored subprograms and various clients. Neither PL/SQL nor any of its clients owns a result set; they simply share a pointer to the query work area that identifies the result set. For example, an Oracle Call Interface (OCI) client, or an Oracle Forms application, or the Oracle server can all refer to the same work area. This might be useful in Oracle Forms, for instance, when you want to populate a multiple-block form.

Example

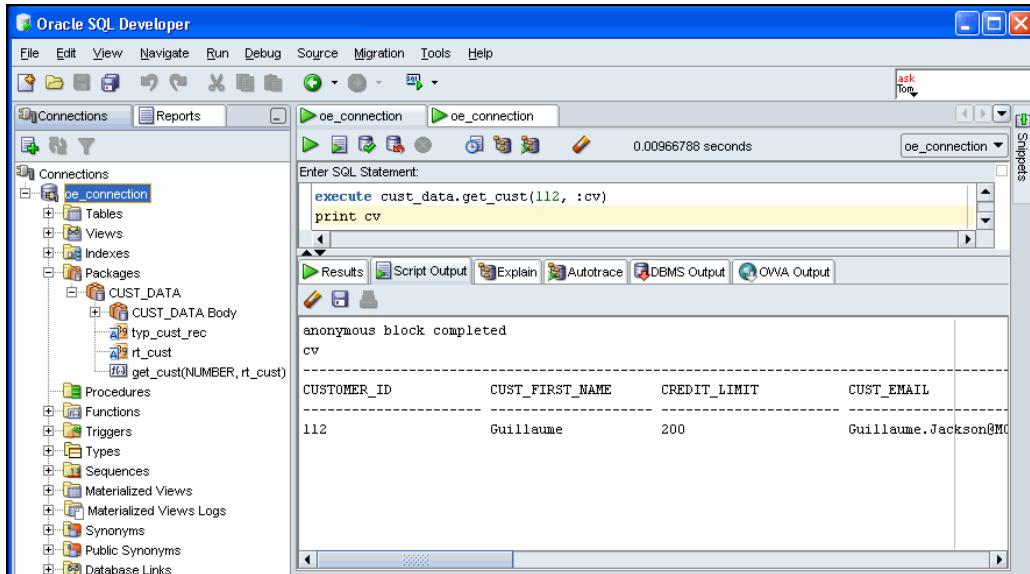
Using SQL*Plus, define a host variable with a data type of REFCURSOR to hold the query results generated from a REF CURSOR in a stored subprogram. Use the SQL*Plus PRINT command to view the host variable results. Optionally, you can set the SQL*Plus command SET AUTOPRINT ON to display the query results automatically.

```
SQL> VARIABLE cv REFCURSOR
```

Next, create a subprogram that uses a REF CURSOR to pass the cursor variable data back to the SQL*Plus environment.

Note: You can define a host variable in SQL*Plus or SQL Developer. This slide uses SQL*Plus. The next slide shows the use of SQL Developer.

Passing Cursor Variables as Arguments



Passing Cursor Variables as Arguments (continued)

```
CREATE OR REPLACE PACKAGE cust_data AS
  TYPE typ_cust_rec IS RECORD
    (cust_id NUMBER(6), custname VARCHAR2(20),
     credit   NUMBER(9,2), cust_email VARCHAR2(30));
  TYPE rt_cust IS REF CURSOR RETURN typ_cust_rec;
  PROCEDURE get_cust
    (p_custid IN NUMBER, p_cv_cust IN OUT rt_cust);
END ;
/
```

Passing Cursor Variables as Arguments (continued)

```
CREATE OR REPLACE PACKAGE BODY cust_data AS
  PROCEDURE get_cust
    (p_custid IN NUMBER, p_cv_cust IN OUT rt_cust)
  IS
  BEGIN
    OPEN p_cv_cust FOR
      SELECT customer_id, cust_first_name, credit_limit, cust_email
        FROM customers
       WHERE customer_id = p_custid;
    -- CLOSE p_cv_cust
  END;
END;
/
```

Note that the CLOSE p_cv_cust statement is commented. This is done because, if you close the REF cursor, it is not accessible from the host variable.

Using the Predefined Type SYS_REFCURSOR

```
CREATE OR REPLACE PROCEDURE REFCUR
(p_num IN NUMBER)
IS
refcur sys_refcursor; ←
empno      emp.empno%TYPE;
ename       emp.ename%TYPE;
BEGIN
IF p_num = 1 THEN
    OPEN refcur FOR SELECT empno, ename FROM emp;
    DBMS_OUTPUT.PUT_LINE('Employee#      Name');
    DBMS_OUTPUT.PUT_LINE('-----  -----');
    LOOP
        FETCH refcur INTO empno, ename;
        EXIT WHEN refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(empno || '      ' || ename);
    END LOOP;
ELSE
....
```

SYS_REFCURSOR is a built-in REF CURSOR type that allows any result set to be associated with it.

ORACLE

3 - 23

Copyright © 2008, Oracle. All rights reserved.

Using the Predefined Type SYS_REFCURSOR

You can define a cursor variable by using the built-in SYS_REFCURSOR data type as well as by creating a REF CURSOR type, and then declaring a variable of that type. SYS_REFCURSOR is a REF CURSOR type that allows any result set to be associated with it. As mentioned earlier, this is known as a *weak* (nonrestrictive) REF CURSOR.

SYS_REFCURSOR can be used to:

- Declare a cursor variable in an Oracle stored procedure or function
- Pass cursors from and to an Oracle stored procedure or function

Note: Strong (restrictive) REF CURSORS require the result set to conform to a declared number and order of fields with compatible data types, and can also, optionally, return a result set.

```
CREATE OR REPLACE PROCEDURE REFCUR
(p_num IN NUMBER)
IS
refcur sys_refcursor;
empno      emp.empno%TYPE;
ename       emp.ename%TYPE;
BEGIN
-- continued on the next page
```

Using the Predefined Type `SYS_REFCURSOR` (continued)

```
-- continued from the previous page
IF p_num = 1 THEN
    OPEN refcur FOR SELECT empno, ename FROM emp;
    DBMS_OUTPUT.PUT_LINE('Employee#      Name');
    DBMS_OUTPUT.PUT_LINE('-----      -----');
    LOOP
        FETCH refcur INTO empno, ename;
        EXIT WHEN refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(empno || '      ' || ename);
    END LOOP;
ELSE
    OPEN refcur FOR
        SELECT empno, ename
        FROM emp WHERE deptno = 30;
    DBMS_OUTPUT.PUT_LINE('Employee#      Name');
    DBMS_OUTPUT.PUT_LINE('-----      -----');
    LOOP
        FETCH refcur INTO empno, ename;
        EXIT WHEN refcur%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE(empno || '      ' || ename);
    END LOOP;
END IF;
CLOSE refcur;
END;
/
```

Rules for Cursor Variables

- You cannot use cursor variables with remote subprograms on another server.
- You cannot use comparison operators to test cursor variables.
- You cannot assign a null value to cursor variables.
- You cannot use REF CURSOR types in CREATE TABLE or VIEW statements.
- Cursors and cursor variables are not interoperable.

ORACLE®

3 - 25

Copyright © 2008, Oracle. All rights reserved.

Restrictions

- Remote subprograms on another server cannot accept the values of cursor variables. Therefore, you cannot use remote procedure calls (RPCs) to pass cursor variables from one server to another.
- If you pass a host cursor variable to PL/SQL, you cannot fetch from it on the server side unless you open it in the server on the same server call.
- You cannot use comparison operators to test cursor variables for equality, inequality, or nullity.
- You cannot assign NULLs to a cursor variable.
- You cannot use the REF CURSOR types to specify column types in a CREATE TABLE or CREATE VIEW statement. So, database columns cannot store the values of cursor variables.
- You cannot use a REF CURSOR type to specify the element type of a collection, which means that the elements in an index by table, nested table, or VARRAY cannot store the values of cursor variables.
- Cursors and cursor variables are not interoperable, that is, you cannot use one where the other is expected.

Comparing Cursor Variables with Static Cursors

Cursor variables have the following benefits:

- Are dynamic and ensure more flexibility
- Are not tied to a single SELECT statement
- Hold the value of a pointer
- Can reduce network traffic
- Give access to query work areas after a block completes



Comparing Cursor Variables with Static Cursors

Cursor variables are dynamic and provide wider flexibility. Unlike static cursors, cursor variables are not tied to a single SELECT statement. In applications where SELECT statements may differ depending on various situations, the cursor variables can be opened for each of the SELECT statements. Because cursor variables hold the value of a pointer, they can be easily passed between programs, no matter where the programs exist.

Cursor variables can reduce network traffic by grouping OPEN FOR statements and sending them across the network only once. For example, the following PL/SQL block opens two cursor variables in a single round trip:

```
/* anonymous PL/SQL block in host environment */
BEGIN
    OPEN :cv_cust FOR SELECT * FROM customers;
    OPEN :cv_orders FOR SELECT * FROM orders;
END;
```

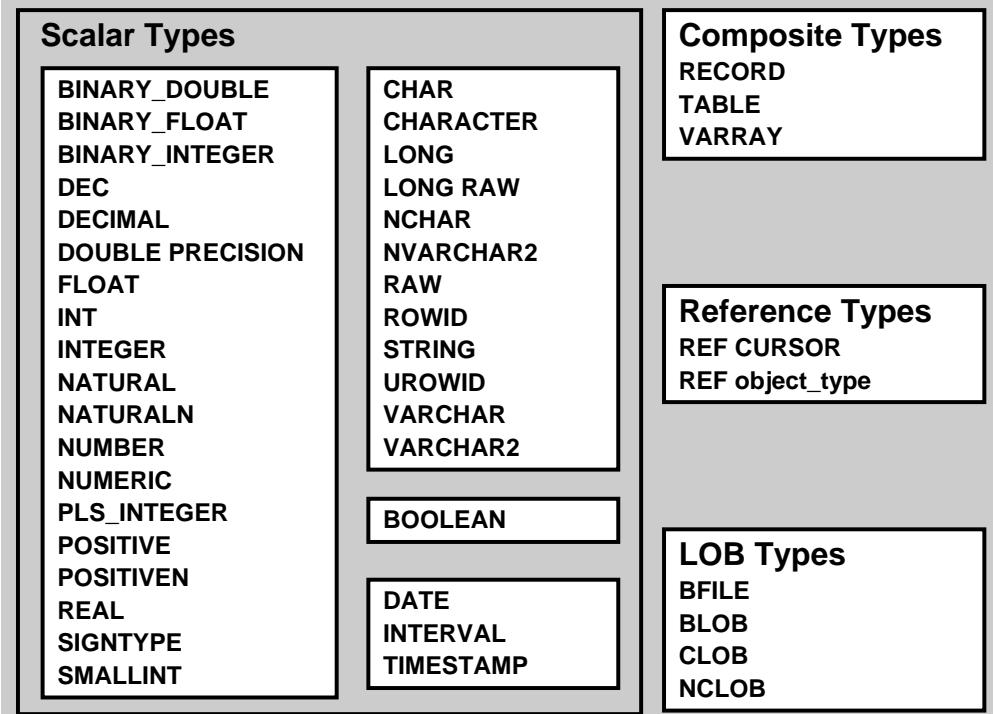
This may be useful in Oracle Forms, for instance, when you want to populate a multiple-block form. When you pass host cursor variables to a PL/SQL block for opening, the query work areas to which they point remain accessible after the block completes. This enables your OCI or Pro*C program to use these work areas for ordinary cursor operations.

Lesson Agenda

- Identifying guidelines for cursor design
- Using Cursor Variables
- Creating subtypes based on existing types



Predefined PL/SQL Data Types



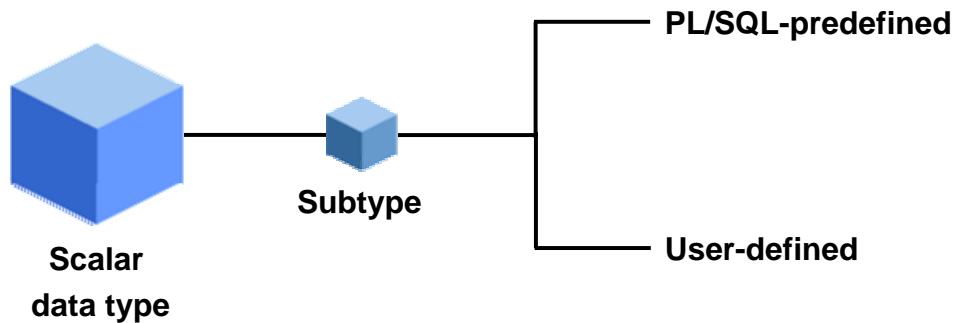
ORACLE®

Predefined PL/SQL Data Types

Every constant, variable, and parameter has a data type, which specifies a storage format, a valid range of values, and constraints. PL/SQL provides a variety of predefined data types. For instance, you can choose from integer, floating point, character, Boolean, date, collection, reference, and LOB types. In addition, PL/SQL enables you to define subtypes.

Subtypes: Overview

A subtype is a subset of an existing data type that may place a constraint on its base type.



ORACLE®

Subtypes: Overview

A subtype is a data type based on an existing data type. It does not define a new data type; instead, it places a constraint on an existing data type. There are several predefined subsets specified in the standard package. DECIMAL and INTEGER are subtypes of NUMBER. CHARACTER is a subtype of CHAR.

Standard Subtypes

BINARY_INTEGER	NUMBER	VARCHAR2
NATURAL	DEC	STRING
NATURALN	DECIMAL	VARCHAR
POSITIVE	DOUBLE PRECISION	
POSITIVEN	FLOAT	
SIGNTYPE	INTEGER	
	INT	
	NUMERIC	
	REAL	
	SMALLINT	

Subtypes: Overview (continued)

With NATURAL and POSITIVE subtypes, you can restrict an integer variable to nonnegative and positive values, respectively. NATURALN and POSITIVEN prevent the assigning of nulls to an integer variable. You can use SIGNTYPE to restrict an integer variable to the values -1, 0, and 1, which is useful in programming tri-state logic.

A constrained subtype is a subset of the values normally specified by the data type on which the subtype is based. POSITIVE is a constrained subtype of BINARY_INTEGER.

An unconstrained subtype is not a subset of another data type; it is an alias to another data type. FLOAT is an unconstrained subtype of NUMBER.

Use the subtypes DEC, DECIMAL, and NUMERIC to declare fixed-point numbers with a maximum precision of 38 decimal digits.

Use the subtypes DOUBLE PRECISION and FLOAT to declare floating-point numbers with a maximum precision of 126 binary digits, which is roughly equivalent to 38 decimal digits. Or, use the subtype REAL to declare floating-point numbers with a maximum precision of 63 binary digits, which is roughly equivalent to 18 decimal digits.

Use the subtypes INTEGER, INT, and SMALLINT to declare integers with a maximum precision of 38 decimal digits.

You can even create your own user-defined subtypes.

Note: You can use these subtypes for compatibility with ANSI/ISO and IBM types. Currently, VARCHAR is synonymous with VARCHAR2. However, in future releases of PL/SQL, to accommodate emerging SQL standards, VARCHAR may become a separate data type with different comparison semantics. It is a good idea to use VARCHAR2 rather than VARCHAR.

Benefits of Subtypes

Subtypes:

- Increase reliability
- Provide compatibility with ANSI/ISO and IBM types
- Promote reusability
- Improve readability
 - Clarity
 - Code self-documents

ORACLE®

3 - 31

Copyright © 2008, Oracle. All rights reserved.

Benefits of Subtypes

If your applications require a subset of an existing data type, you can create subtypes. By using subtypes, you can increase the reliability and improve the readability by indicating the intended use of constants and variables. Subtypes can increase reliability by detecting the out-of-range values.

With predefined subtypes, you have compatibility with other data types from other programming languages.

Declaring Subtypes

- Subtypes are defined in the declarative section of a PL/SQL block.

```
SUBTYPE subtype_name IS base_type [(constraint)]  
[NOT NULL];
```

- *subtype_name* is a type specifier used in subsequent declarations.
- *base_type* is any scalar or user-defined PL/SQL type.

ORACLE®

3 - 32

Copyright © 2008, Oracle. All rights reserved.

Declaring Subtypes

Subtypes are defined in the declarative section of a PL/SQL block, subprogram, or package.

Using the SUBTYPE keyword, you name the subtype and provide the name of the base type.

You can use the %TYPE attribute on the base type to pick up a data type from a database column or from an existing variable data type. You can also use the %ROWTYPE attribute.

Examples

```
CREATE OR REPLACE PACKAGE mytypes  
IS  
    SUBTYPE Counter IS INTEGER; -- based on INTEGER type  
    TYPE typ_TimeRec IS RECORD (minutes INTEGER, hours  
        INTEGER);  
    SUBTYPE Time IS typ_TimeRec; -- based on RECORD type  
    SUBTYPE ID_Num IS customers.customer_id%TYPE;  
    CURSOR cur_cust IS SELECT * FROM customers;  
    SUBTYPE CustFile IS cur_cust%ROWTYPE; -- based on cursor  
END mytypes;  
/
```

Using Subtypes

- Define a variable that uses the subtype in the declarative section.

```
identifier_name subtype_name;
```

- You can constrain a user-defined subtype when declaring variables of that type.

```
identifier_name subtype_name(size);
```

- You can constrain a user-defined subtype when declaring the subtype.

ORACLE®

3 - 33

Copyright © 2008, Oracle. All rights reserved.

Using Subtypes

After a subtype is declared, you can assign an identifier for that subtype. Subtypes can increase reliability by detecting out-of-range values.

```
DECLARE
    v_rows      mytypes.Counter; --use package subtype dfn
    v_customers mytypes.Counter;
    v_start_time mytypes.Time;
    SUBTYPE      Accumulator IS NUMBER;
    v_total      Accumulator(4,2);
    SUBTYPE      Scale IS NUMBER(1,0);   -- constrained subtype
    v_x_axis     Scale;   -- magnitude range is -9 .. 9
BEGIN
    v_rows := 1;
    v_start_time.minutes := 15;
    v_start_time.hours   := 03;
    dbms_output.put_line('Start time is: ' ||
    v_start_time.hours|| ':' || v_start_time.minutes);
END;
/
```

Subtype Compatibility

An unconstrained subtype is interchangeable with its base type.

```
DECLARE
    SUBTYPE Accumulator IS NUMBER (4,2);
    v_amount  accumulator;
    v_total   NUMBER;
BEGIN
    v_amount := 99.99;
    v_total  := 100.00;
    dbms_output.put_line('Amount is: ' || v_amount);
    dbms_output.put_line('Total is: ' || v_total);
    v_total := v_amount;
    dbms_output.put_line('This works too: ' ||
    v_total);
    -- v_amount := v_amount + 1; Will show value error
END;
/
```

ORACLE®

3 - 34

Copyright © 2008, Oracle. All rights reserved.

Subtype Compatibility

Some applications require constraining subtypes to a size specification for scientific purposes. The example in the slide shows that if you exceed the size of your subtype, you receive an error.

An unconstrained subtype is interchangeable with its base type. Different subtypes are interchangeable if they have the same base type. Different subtypes are also interchangeable if their base types are in the same data type family.

```
DECLARE
    v_rows      mytypes.Counter;  v_customers mytypes.Counter;
    SUBTYPE     Accumulator IS NUMBER (6,2);
    v_total     NUMBER;
BEGIN
    SELECT COUNT(*) INTO v_customers FROM customers;
    SELECT COUNT(*) INTO v_rows FROM orders;
    v_total := v_customers + v_rows;
    DBMS_OUTPUT.PUT_LINE('Total rows from 2 tables: ' ||
    v_total);
EXCEPTION
    WHEN value_error THEN
        DBMS_OUTPUT.PUT_LINE('Error in data type.');
END;
```

Summary

In this lesson, you should have learned how to:

- Use guidelines for cursor design
- Declare, define, and use cursor variables
- Use subtypes as data types



Summary

- Use the guidelines for designing the cursors.
- Take advantage of the features of cursor variables and pass pointers to result sets to different applications.
- You can use subtypes to organize and strongly type data types for an application.

Practice 3: Overview

This practice covers the following topics:

- Determining the output of a PL/SQL block
- Improving the performance of a PL/SQL block
- Implementing subtypes
- Using cursor variables



Practice 3: Overview

In this practice, you determine the output of a PL/SQL code snippet and modify the snippet to improve performance. Next, you implement subtypes and use cursor variables to pass values to and from a package.

Practice 3: Designing PL/SQL Code

Note: The files mentioned in the practice exercises are found in the `/labs` folder. Additionally, solution scripts are provided for each question and are located in the `/soln` folder. Your instructor will provide you with the exact location of these files. Connect as OE to perform the steps.

1. Determine the output of the following code snippet in the `lab_03_01.sql` file.

```
SET SERVEROUTPUT ON
BEGIN
    UPDATE orders SET order_status = order_status;
    FOR v_rec IN ( SELECT order_id FROM orders )
    LOOP
        IF SQL%ISOPEN THEN
            DBMS_OUTPUT.PUT_LINE('TRUE - ' || SQL%ROWCOUNT);
        ELSE
            DBMS_OUTPUT.PUT_LINE('FALSE - ' || SQL%ROWCOUNT);
        END IF;
    END LOOP;
END;
/
```

2. Modify the following code snippet in the `lab_03_02.sql` file to make better use of the `FOR UPDATE` clause and improve the performance of the program.

```
DECLARE
    CURSOR cur_update
        IS SELECT * FROM customers
        WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
    FOR v_rec IN cur_update
    LOOP
        IF v_rec IS NOT NULL
        THEN
            UPDATE customers
            SET credit_limit = credit_limit + 200
            WHERE customer_id = v_rec.customer_id;
        END IF;
    END LOOP;
END;
/
```

Practice 3 (continued)

3. Create a package specification that defines subtypes, which can be used for the `warranty_period` field of the `product_information` table. Name this package `MY_TYPES`. The type needs to hold the month and year for a warranty period.

4. Create a package named `SHOW_DETAILS` that contains two subroutines. The first subroutine should show order details for the given `order_id`. The second subroutine should show customer details for the given `customer_id`, including the customer ID, the first name, phone numbers, credit limit, and email address. Both the subroutines should use the cursor variable to return the necessary details.

Working with Collections

4

ORACLE

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

- Create collections
 - Nested table, varray
 - Associative arrays/PLSQL tables
 - Integer indexed
 - String indexed
- Use collections methods
- Manipulate collections
- Distinguish between the different types of collections and when to use them

ORACLE

4 - 2

Copyright © 2008, Oracle. All rights reserved.

Objectives

In this lesson, you are introduced to PL/SQL programming using collections.

A collection is an ordered group of elements, all of the same type (for example, phone numbers for each customer). Each element has a unique subscript that determines its position in the collection.

Collections work like the set, queue, stack, and hash table data structures found in most third-generation programming languages. Collections can store instances of an object type and can also be attributes of an object type. Collections can be passed as parameters. So, you can use them to move columns of data into and out of database tables, or between client-side applications and stored subprograms. You can define collection types in a PL/SQL package, and then use the same types across many applications.

Lesson Agenda

- Understanding collections
- Using associative arrays
- Using nested tables
- Using varrays
- Working with collections
- Programming for collection exceptions
- Summarizing collections

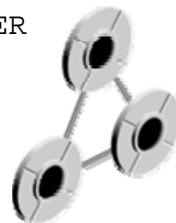
ORACLE

4 - 3

Copyright © 2008, Oracle. All rights reserved.

Understanding Collections

- A collection is a group of elements, all of the same type.
- Collections work like arrays.
- Collections can store instances of an object type and, conversely, can be attributes of an object type.
- Types of collections in PL/SQL:
 - Associative arrays
 - String-indexed collections
 - INDEX BY pls_integer or BINARY_INTEGER
 - Nested tables
 - Varrays



ORACLE

4 - 4

Copyright © 2008, Oracle. All rights reserved.

Understanding Collections

A collection is a group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection. Collections work like the arrays found in most third-generation programming languages. They can store instances of an object type and, conversely, can be attributes of an object type. Collections can also be passed as parameters. You can use them to move columns of data into and out of database tables, or between client-side applications and stored subprograms.

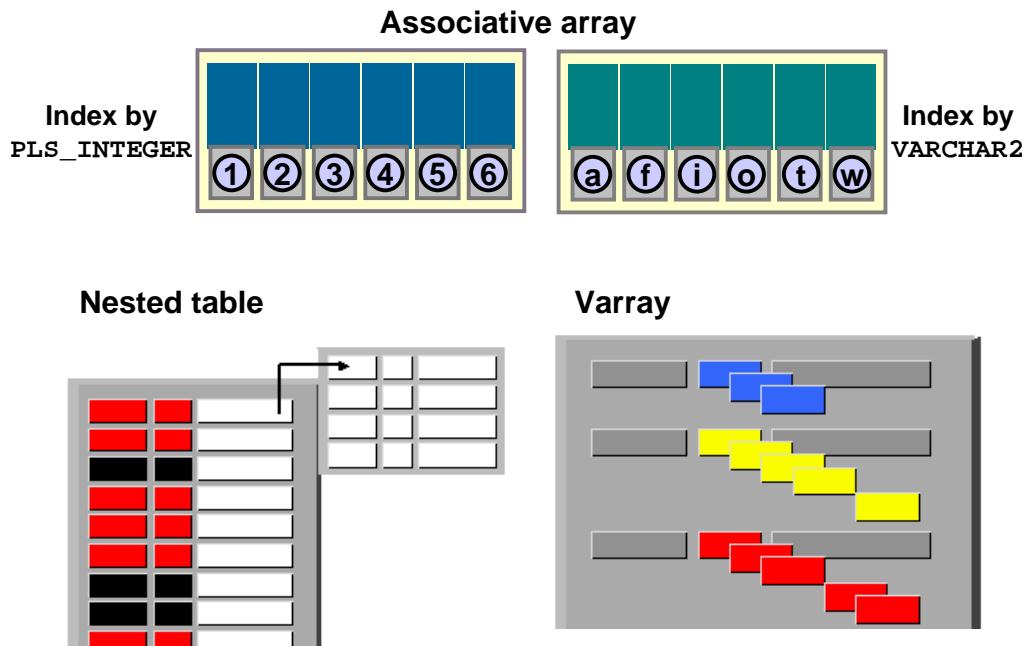
Object types are used not only to create object relational tables, but also to define collections.

You can use any of the three categories of collections:

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

Note: Associative arrays indexed by `pls_integer` are covered in the prerequisite courses—*Oracle Database 11g: Program with PL/SQL* and *Oracle Database 11g: Develop PL/SQL Program Units*—and are not emphasized in this course.

Collection Types



Collection Types

PL/SQL offers three collection types:

Associative Arrays

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

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

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

Collection Types (continued)

Nested Tables

A nested table holds a set of values. In other words, it is a table within a table. Nested tables are unbounded; that is, the size of the table can increase dynamically. Nested tables are available in both PL/SQL and the database. Within PL/SQL, nested tables are like one-dimensional arrays whose size can increase dynamically. Within the database, nested tables are column types that hold sets of values. The Oracle database stores the rows of a nested table in no particular order. When you retrieve a nested table from the database into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. This gives you an array-like access to individual rows. Nested tables are initially dense, but they can become sparse through deletions and, therefore, have nonconsecutive subscripts.

Varrays

Variable-size arrays, or varrays, are also collections of homogeneous elements that hold a fixed number of elements (although you can change the number of elements at run time). They use sequential numbers as subscripts. You can define equivalent SQL types, thereby allowing varrays to be stored in database tables. They can be stored and retrieved through SQL, but with less flexibility than nested tables. You can reference the individual elements for array operations or manipulate the collection as a whole.

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

Choosing a PL/SQL Collection Type

If you already have code or business logic that uses another language, you can usually translate that language's array and set the types directly to the PL/SQL collection types.

- Arrays in other languages become varrays in PL/SQL.
- Sets and bags in other languages become nested tables in PL/SQL.
- Hash tables and other kinds of unordered lookup tables in other languages become associative arrays in PL/SQL.

If you are writing original code or designing the business logic from the start, consider the strengths of each collection type and decide which is appropriate.

Why Use Collections?

Collections offer object-oriented features such as variable-length arrays and nested tables that provide higher-level ways to organize and access data in the database. Below the object layer, data is still stored in columns and tables, but you are able to work with the data in terms of the real-world entities, such as customers and purchase orders, that make the data meaningful.

Lesson Agenda

- Understanding collections
- Using associative arrays
- Using nested tables
- Using varrays
- Working with collections
- Programming for collection exceptions
- Summarizing collections

ORACLE

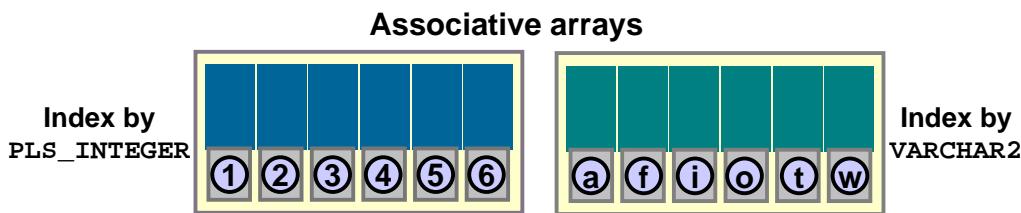
4 - 7

Copyright © 2008, Oracle. All rights reserved.

Using Associative Arrays

Associative arrays:

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



Using Associative Arrays

Associative arrays (known as “index by tables” in previous Oracle releases) are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be an integer or a string.

When to Use String-Indexed Arrays

You can use INDEX BY VARCHAR2 tables (also known as string-indexed arrays). These tables are optimized for efficiency by implicitly using the B*-tree organization of the values.

The INDEX BY VARCHAR2 table is optimized for efficiency of lookup on a nonnumeric key, where the notion of sparseness is not applicable. In contrast, the INDEX BY PLS_INTEGER tables are optimized for compactness of storage on the assumption that the data is dense.

Note: Associative arrays indexed by PLS_INTEGER are covered in the prerequisite courses—*Oracle Database 11g: Program with PL/SQL* and *Oracle Database 11g: Develop PL/SQL Program Units*—and are not emphasized in this course.

Creating the Array

Associative array in PL/SQL (string-indexed):

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

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(30);

  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;
  ...
  
```

ORACLE

4 - 9

Copyright © 2008, Oracle. All rights reserved.

Using String-Indexed Arrays

If you need to do heavy processing of customer information in your program that requires going back and forth over the set of selected customers, you can use string-indexed arrays to store, process, and retrieve the required information.

This can also be done in SQL but probably in a less efficient implementation. If you need to do multiple passes over a significant set of static data, you can instead move it from the database to a set of collections. Accessing collection-based data is much faster than going through the SQL engine.

After transferring the data from the database to the collections, you can use string- and integer-based indexing on those collections to, in essence, mimic the primary key and unique indexes on the table.

In the REPORT_CREDIT procedure shown in the slide, you may need to determine whether a customer has adequate credit. The string-indexed collection is loaded with the customer information in the LOAD_ARRAYS procedure. In the main body of the program, the collection is traversed to find the credit information. The email name is reported in case more than one customer has the same last name.

Populating the 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)

For credit amount of: 1200
Customer Walken: Emmet.Walken@LIMPKIN.COM has credit limit of: 3600
Customer Walken: Prem.Walken@BRANT.COM has credit limit of: 3700
```

ORACLE

4 - 10

Copyright © 2008, Oracle. All rights reserved.

Using String-Indexed Arrays (continued)

In this example, the string-indexed collection is traversed using the NEXT method.

A more efficient use of the string-indexed collection is to index the collection with the customer email. Then you can immediately access the information based on the customer email key. You would need to pass the email name instead of the customer last name.

Using String-Indexed Arrays (continued)

Here is the modified code:

```
CREATE OR REPLACE PROCEDURE report_credit
    (p_email      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(30);

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

    BEGIN
        load_arrays;
        dbms_output.put_line
            ('For credit amount of: ' || p_credit_limit);
        IF v_by_cust_email(p_email).credit_limit > p_credit_limit
            THEN dbms_output.put_line ( 'Customer ' ||
                v_by_cust_email(p_email).cust_last_name ||
                ': ' || v_by_cust_email(p_email).cust_email ||
                ' has credit limit of: ' ||
                v_by_cust_email(p_email).credit_limit);
        END IF;
    END report_credit;
/

EXECUTE report_credit('Prem.Walken@BRANT.COM', 100)

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

PL/SQL procedure successfully completed.
```

Lesson Agenda

- Understanding collections
- Using associative arrays
- **Using nested tables**
- Using varrays
- Working with collections
- Programming for collection exceptions
- Summarizing collections

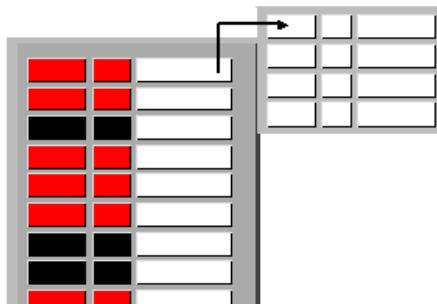
ORACLE

Using Nested Tables

Nested table characteristics:

- A table within a table
- Unbounded
- Available in both SQL and PL/SQL as well as the database
- Array-like access to individual rows

Nested table:



ORACLE

4 - 13

Copyright © 2008, Oracle. All rights reserved.

Nested Tables

A nested table holds a set of values. In other words, it is a table within a table. Nested tables are unbounded, meaning that the size of the table can increase dynamically. Nested tables are available in both PL/SQL as well as the database. Within PL/SQL, nested tables are like one-dimensional arrays whose size can increase dynamically. Within the database, nested tables are column types that hold sets of values. The Oracle database stores the rows of a nested table in no particular order. When you retrieve a nested table from the database into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. This gives you an array-like access to individual rows.

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

Nested Table Storage

Nested tables are stored out-of-line in storage tables.

pOrder nested table:

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

Storage table:

NESTED_TABLE_ID	PRODID	PRICE
	55	555
	56	566
	57	577
NESTED_TABLE_ID	PRODID	PRICE
	88	888

ORACLE

Nested Table Storage

The rows for all nested tables of a particular column are stored within the same segment. This segment is called the *storage table*.

A storage table is a system-generated segment in the database that holds instances of nested tables within a column. You specify a name for the storage table by using the NESTED TABLE STORE AS clause in the CREATE TABLE statement. The storage table inherits storage options from the outermost table.

To distinguish between nested table rows belonging to different parent table rows, a system-generated nested table identifier that is unique for each outer row enclosing a nested table is created.

Operations on storage tables are performed implicitly by the system. You should not access or manipulate the storage table, except implicitly through its containing objects.

The column privileges of the parent table are transferred to the nested table.

Creating Nested Tables

To create a nested table in the database:

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

To create a nested table in PL/SQL:

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



Creating Collection Types

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

Creating Collections in the Database

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

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

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

Creating Collections in PL/SQL

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

Note: Collections can be nested. Collections of collections are also possible.

Declaring Collections: Nested Table

- First, define an object type:

```
CREATE [TYPE] typ_item AS [OBJECT] --create object 1
  (prodid  NUMBER(5),
   price    NUMBER(7,2) )
/
CREATE [TYPE] typ_item_nst -- define nested table type 2
  AS [TABLE OF typ_item]
/
```

- Second, declare a column of that collection type:

```
CREATE TABLE pOrder ( -- create database table 3
  ordid    NUMBER(5),
  supplier NUMBER(5),
  requester      NUMBER(4),
  ordered     DATE,
  items      typ_item_nst)
  NESTED TABLE items STORE AS item_stor_tab
/
```

ORACLE

4 - 16

Copyright © 2008, Oracle. All rights reserved.

Declaring Collections: Nested Table

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

1. Create the `typ_item` type, which holds the information for a single line item.
2. Create the `typ_item_nst` type, which is created as a table of the `typ_item` type.
Note: You must create the `typ_item_nst` nested table type based on the previously declared type, because it is illegal to declare multiple data types in this nested table declaration.
3. Create the `pOrder` table and use the nested table type in a column declaration, which includes an arbitrary number of items based on the `typ_item_nst` type. Thus, each row of `pOrder` may contain a table of items.

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

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

Using Nested Tables

- Add data to the nested table:

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

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

pOrder nested table

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

PRODID	PRICE
55	555
56	566
57	577

PRODID	PRICE
88	888

ORACLE

4 - 17

Copyright © 2008, Oracle. All rights reserved.

Using Nested Tables

To insert data into the nested table, you use the `INSERT` statement. A constructor is a system-defined function that is used to identify where the data should be placed, essentially “constructing” the collection from the elements passed to it.

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

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

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

Using Nested Tables

- Querying the results:

```
SELECT * FROM porder;

      ORDID    SUPPLIER REQUESTER ORDERED
----- -----
ITEMS(PRODID, PRICE)
-----
      500        50      5000 31-OCT-07
TYP_ITEM_NST(TYP_ITEM(55, 555), TYP_ITEM(56, 566), TYP_ITEM(57, 577))
      800        80      8000 31-OCT-07
TYP_ITEM_NST(TYP_ITEM(88, 888))
```

- Querying the results with the TABLE function:

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

      ORDID    PRODID      PRICE
----- -----
      800        88      888
      500        57      577
      500        55      555
      500        56      566
```

ORACLE

4 - 18

Copyright © 2008, Oracle. All rights reserved.

Querying Nested Tables

You can use two general methods to query a table that contains a column or attribute of a collection type. One method returns the collections nested in the result rows that contain them. By including the collection column in the SELECT list, the output shows as a row associated with the other row output in the SELECT list.

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

Querying Collections with the TABLE Expression

To view collections in a conventional format, you must unnest, or flatten, the collection attribute of a row into one or more relational rows. You can do this by using a TABLE expression with the collection. A TABLE expression enables you to query a collection in the FROM clause like a table. In effect, you join the nested table with the row that contains the nested table without writing a JOIN statement.

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

Referencing Collection Elements

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

- Syntax:

```
collection_name(subscript)
```

- Example:

```
v_with_discount(i)
```

- To reference a field in a collection:

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

ORACLE

4 - 19

Copyright © 2008, Oracle. All rights reserved.

Referencing Collection Elements

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

```
collection_name(subscript)
```

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

Using Nested Tables in PL/SQL

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

ORACLE

4 - 20

Copyright © 2008, Oracle. All rights reserved.

Using Nested Tables in PL/SQL

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

In the example shown in the slide:

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

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;
```

v_form_items variable

PRODID	PRICE
1804	65
3172	42
3337	800
2144	14

Resulting data in the pOrder nested table

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

PRODID	PRICE
1804	65
3172	42
3337	800
2144	14

ORACLE

Using Nested Tables in PL/SQL (continued)

In the example code shown in the slide:

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

Lesson Agenda

- Understanding collections
- Using associative arrays
- Using nested tables
- **Using varrays**
- Working with collections
- Programming for collection exceptions
- Summarizing collections

ORACLE

Understanding Varrays

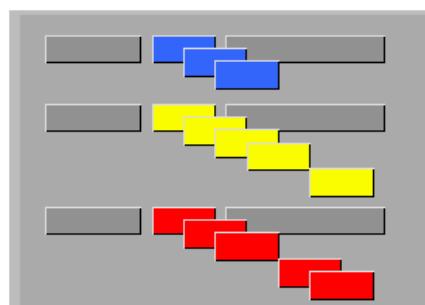
- To create a varray in the database:

```
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];
```

Varray:



ORACLE

Understanding Varrays

Varrays are also collections of homogeneous elements that hold a fixed number of elements (although you can change the number of elements at run time). They use sequential numbers as subscripts.

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

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

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

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

Declaring Collections: Varray

- First, define a collection type:

```
CREATE [TYPE] typ_Project AS OBJECT( --create object 1
    project_no NUMBER(4),
    title      VARCHAR2(35),
    cost       NUMBER(12,2))
/
CREATE [TYPE] typ_ProjectList AS VARRAY(50) OF typ_Project 2
/
-- define VARRAY type
```

- Second, declare a collection of that type:

```
CREATE TABLE department ( -- create database table 3
    dept_id  NUMBER(2),
    name     VARCHAR2(25),
    budget   NUMBER(12,2),
    projects typ_ProjectList) -- declare varray as column
/
-- declare varray as column
```

ORACLE

4 - 24

Copyright © 2008, Oracle. All rights reserved.

Example

The example above shows how to create a table based on a varray.

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

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

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

Using Varrays

Add data to the table containing a varray column:

```
INSERT INTO department
    VALUES (10, 'Executive Administration', 30000000, 1
           typ_ProjectList(
               typ_Project(1001, 'Travel Monitor', 400000),
               typ_Project(1002, 'Open World', 10000000)));
2
INSERT INTO department
    VALUES (20, 'Information Technology', 5000000,
           typ_ProjectList(
               typ_Project(2001, 'DB11gR2', 900000)));
```

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

ORACLE

Example (continued)

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

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

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

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

Using Varrays

- Querying the results:

```
SELECT * FROM department;

DEPT_ID NAME          BUDGET
----- -----
PROJECTS(PROJECT_NO, TITLE, COST)
-----
10 Executive Administration    30000000
TYP_PROJECTLIST(TYP_PROJECT(1001, 'Travel Monitor', 400000),
TYP_PROJECT(1002, 'Open World', 10000000))
20 Information Technology     5000000
TYP_PROJECTLIST(TYP_PROJECT(2001, 'DB11gR2', 900000))
```

- Querying the results with the TABLE function:

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

DEPT_ID NAME          PROJECT_NO TITLE      COST
----- -----
10 Executive Administration 1001 Travel Monitor 400000
10 Executive Administration 1002 Open World   10000000
20 Information Technology  2001 DB11gR2    900000
```

ORACLE

Querying Varray Columns

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

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

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

Lesson Agenda

- Understanding collections
- Using associative arrays
- Using nested tables
- Using varrays
- **Working with collections**
- Programming for collection exceptions
- Summarizing collections

ORACLE

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.
- Collections follow the usual scoping and instantiation rules.

```
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;
```

ORACLE

4 - 28

Copyright © 2008, Oracle. All rights reserved.

Working with Collections in PL/SQL

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

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

Working with Collections: Example

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

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

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

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

Working with Collections: Example (continued)

-- continued from previous page

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

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

END manage_dept_proj;
```

Initializing Collections

Three ways to initialize:

- Use a constructor.
- Fetch from the database.
- Assign another collection variable directly.

```
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;
```

ORACLE

4 - 31

Copyright © 2008, Oracle. All rights reserved.

Initializing Collections

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

- Use a constructor, which is a system-defined function with the same name as the collection type. A constructor allows the creation of an object from an object type.
Invoking a constructor is a way to instantiate (create) an object. This function “constructs” collections from the elements passed to it. In the example shown in the slide, you pass three elements to the typ_ProjectList() constructor, which returns a varray containing those elements.
- Read an entire collection from the database using a fetch.
- Assign another collection variable directly. You can copy the entire contents of one collection to another as long as both are built from the same data type.

Initializing Collections

```
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

ORACLE

4 - 32

Copyright © 2008, Oracle. All rights reserved.

Initializing Collections (continued)

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

In the second example in the slide, the entire content of one collection variable is assigned to another collection variable.

Referencing Collection Elements

```
-- 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;
```

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

ORACLE

4 - 33

Copyright © 2008, Oracle. All rights reserved.

Referencing Collection Elements

In the example in the slide, the code calls the MANIPULATE_PROJECT function in the MANAGE_DEPT_PROJ package. Department 10 is passed in as the parameter. The output shows the varray element values for the PROJECTS column in the DEPARTMENT table for department 10.

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

Using Collection Methods

- EXISTS
- COUNT
- LIMIT
- FIRST and LAST
- PRIOR and NEXT
- EXTEND
- TRIM
- DELETE

```
collection_name.method_name [(parameters)]
```

ORACLE

4 - 34

Copyright © 2008, Oracle. All rights reserved.

Using Collection Methods

You can use collection methods from procedural statements but not from SQL statements.

Here is a list of some of the collection methods that you can use. You have already seen a few in the preceding examples.

Using Collection Methods (continued)

Function or Procedure	Description
EXISTS	Returns TRUE if the nth element in a collection exists; otherwise, EXIST(N) returns FALSE.
COUNT	Returns the number of elements that a collection contains.
LIMIT	For nested tables that have no maximum size, LIMIT returns NULL; for varrays, LIMIT returns the maximum number of elements that a varray can contain.
FIRST and LAST	Returns the first and last (smallest and largest) index numbers in a collection, respectively.
PRIOR and NEXT	PRIOR(n) returns the index number that precedes index n in a collection; NEXT(n) returns the index number that follows index n.
EXTEND	Appends one null element. EXTEND(n) appends n elements; EXTEND(n, i) appends n copies of the ith element.
TRIM	Removes one element from the end; TRIM(n) removes n elements from the end of a collection
DELETE	Removes all elements from a nested or associative array table. DELETE(n) removes the nth element ; DELETE(m, n) removes a range. Note: Does not work on varrays.

Using Collection Methods

Traverse collections with the following methods:

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

ORACLE

4 - 36

Copyright © 2008, Oracle. All rights reserved.

Traversing Collections

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

You can use the PRIOR and NEXT methods to traverse collections indexed by any series of subscripts. In the example shown, the NEXT method is used to traverse a varray.

PRIOR(n) returns the index number that precedes index n in a collection. NEXT(n) returns the index number that succeeds index n. If n has no predecessor, PRIOR(n) returns NULL. Likewise, if n has no successor, NEXT(n) returns NULL. PRIOR is the inverse of NEXT.

PRIOR and NEXT do not wrap from one end of a collection to the other.

When traversing elements, PRIOR and NEXT ignore deleted elements.

Using Collection Methods

```
-- sample caller program to check_costs
set serverout 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('Project rejected: overbudget');
    ELSE
        dbms_output.put_line('Project accepted, fill out forms.');
    END IF;
END;
```

```
Audit Accounts Payable exceeded allowable budget.
Project rejected: overbudget
```

V_PROJECT_LIST variable:

PROJECT_NO	TITLE	COSTS
1	Dsgn New Expense Rpt	3250
2	Outsource Payroll	120000
3	Audit Accounts Payable	14250000

ORACLE

Traversing Collections (continued)

The code shown in the slide calls the CHECK_COSTS function (shown on the previous page). The CHECK_COSTS function accepts a varray parameter and returns a Boolean value. If it returns true, the costs for a project element are too high. The maximum budget allowed for a project element is defined by the C_MAX_ALLOWED constant in the function.

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

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

Manipulating Individual Elements

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

ORACLE

4 - 38

Copyright © 2008, Oracle. All rights reserved.

Manipulating Individual Elements

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

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

Manipulating Individual Elements

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

DEPT_ID	NAME	PROJECT_NO	TITLE	COST
10	Executive Administration	1001	Travel Monitor	400000
10	Executive Administration	1002	Open World	10000000
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;
```

```
-- 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
10	Executive Administration	1001	Travel Monitor	400000
10	Executive Administration	1002	Open World	10000000
20	Information Technology	2001	DB11gR2	900000
20	Information Technology	2002	AQM	80000

ORACLE

4 - 39

Copyright © 2008, Oracle. All rights reserved.

Manipulating Individual Elements (continued)

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

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

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

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

What happens if you request a project element to be inserted at position 5?

Lesson Agenda

- Understanding collections
- Using associative arrays
- Using nested tables
- Using varrays
- Working with collections
- Programming for collection exceptions
- Summarizing collections

ORACLE

Avoiding Collection Exceptions

Common exceptions with collections:

- COLLECTION_IS_NULL
- NO_DATA_FOUND
- SUBSCRIPT_BEYOND_COUNT
- SUBSCRIPT_OUTSIDE_LIMIT
- VALUE_ERROR

ORACLE

4 - 41

Copyright © 2008, Oracle. All rights reserved.

Avoiding Collection Exceptions

In most cases, if you reference a nonexistent collection element, PL/SQL raises a predefined exception.

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

Avoiding Collection Exceptions: Example

Common exceptions with collections:

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

```

ORACLE

4 - 42

Copyright © 2008, Oracle. All rights reserved.

Avoiding Collection Exceptions: Example

In the first case, the nested table is atomically null. In the second case, the subscript is null. In the third case, the subscript is outside the legal range. In the fourth case, the subscript exceeds the number of elements in the table. In the fifth case, the subscript designates an element that was deleted.

Lesson Agenda

- Understanding collections
- Using associative arrays
- Using nested tables
- Using varrays
- Working with collections
- Programming for collection exceptions
- Summarizing collections

ORACLE

Listing Characteristics for Collections

	PL/SQL Nested Tables	DB Nested Tables	PL/SQL Varrays	DB Varrays	PL/SQL Associative Arrays
Maximum size	No	No	Yes	Yes	Dynamic
Sparsity	Can be	No	Dense	Dense	Yes
Storage	N/A	Stored out-of-line	N/A	Stored inline (if < 4,000 bytes)	N/A
Ordering	Does not retain ordering and subscripts	Does not retain ordering and subscripts	Retains ordering and subscripts	Retains ordering and subscripts	Retains ordering and subscripts

ORACLE

Choosing Between Nested Tables and Associative Arrays

- Use associative arrays when:
 - You need to collect information of unknown volume.
 - You need flexible subscripts (negative, nonsequential, or string-based).
 - You need to pass the collection to and from the database server (use associative arrays with the bulk constructs).
- Use nested tables when:
 - You need persistence.
 - You need to pass the collection as a parameter.

Choosing Between Nested Tables and Varrays

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

Guidelines for Using Collections Effectively

- Varrays involve fewer disk accesses and are more efficient.
- Use nested tables for storing large amounts of data.
- Use varrays to preserve the order of elements in the collection column.
- If you do not have a requirement to delete elements in the middle of a collection, favor varrays.
- Varrays do not allow piecewise updates.
- After deleting the elements, release the unused memory with `DBMS_SESSION.FREE_UNUSED_USER_MEMORY`

ORACLE

4 - 45

Copyright © 2008, Oracle. All rights reserved.

Guidelines for Using Collections Effectively

- Because varray data is stored inline (in the same tablespace), retrieving and storing varrays involves fewer disk accesses. Varrays are thus more efficient than nested tables.
- To store large amounts of persistent data in a column collection, use nested tables. Thus, the Oracle server can use a separate table to hold the collection data, which can grow over time. For example, when a collection for a particular row could contain 1 to 1,000,000 elements, a nested table is simpler to use.
- If your data set is not very large and it is important to preserve the order of elements in a collection column, use varrays. For example, if you know that the collection will not contain more than 10 elements in each row, you can use a varray with a limit of 10.
- If you do not want to deal with deletions in the middle of the data set, use varrays.
- If you expect to retrieve the entire collection simultaneously, use varrays.
- Varrays do not allow piecewise updates.
- After deleting the elements, you can release the unused memory with the `DBMS_SESSION.FREE_UNUSED_USER_MEMORY` procedure.

Note: If your application requires negative subscripts, you can use only associative arrays.

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
- Define nested tables, varrays, and associative arrays in PL/SQL
 - Access collection elements
 - Use collection methods in PL/SQL
 - Identify raised exceptions with collections
 - Decide which collection type is appropriate for each scenario



Summary

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

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

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

Practice 4: Overview

This practice covers the following topics:

- Analyzing collections
- Using collections



Practice 4: Overview

In this practice, you analyze collections for common errors, create a collection, and then write a PL/SQL package to manipulate the collection.

Use the OE schema for this practice.

For detailed instructions on performing this practice, see Appendix A, “Practice Solutions.”

Practice 4

In this practice, you create a nested table collection and use PL/SQL code to manipulate the collection.

Analyzing Collections

1. Examine the following definitions. Run the `lab_04_01.sql` script to create these objects.

```
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
/
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
/
```

2. The following code generates an error. Run the `lab_04_02.sql` script to generate and view the error.

```
BEGIN
    -- Insert an order
    INSERT INTO pOrder
        (ordid, supplier, requester, ordered, items)
        VALUES (1000, 12345, 9876, SYSDATE, NULL);
    -- insert the items for the order created
    INSERT INTO TABLE (SELECT items
                      FROM   pOrder
                      WHERE  ordid = 1000)
        VALUES(typ_item(99, 129.00));
END;
/
```

a. Why does the error occur?

b. How can you fix the error?

Practice 4 (continued)

Collection Analysis (continued)

3. Examine the following code, which produces an error. Which line causes the error, and how do you fix it?

(Note: You can run the `lab_04_03.sql` script to view the error output).

```
DECLARE
    TYPE credit_card_typ
    IS VARRAY(100) OF VARCHAR2(30);

    v_mc    credit_card_typ := credit_card_typ();
    v_visa  credit_card_typ := credit_card_typ();
    v_am    credit_card_typ;
    v_disc  credit_card_typ := credit_card_typ();
    v_dc    credit_card_typ := credit_card_typ();

BEGIN
    v_mc.EXTEND;
    v_visa.EXTEND;
    v_am.EXTEND;
    v_disc.EXTEND;
    v_dc.EXTEND;
END;
/
```

Practice 4 (continued)

Using Collections

In the following practice exercises, you implement a nested table column in the CUSTOMERS table and write PL/SQL code to manipulate the nested table.

4. Create a nested table to hold credit card information.

- a. Create an object type called typ_cr_card. It should have the following specification:

```
card_type  VARCHAR2(25)
card_num   NUMBER
```

- b. Create a nested table type called typ_cr_card_nst that is a table of typ_cr_card.
 - c. Add a column to the CUSTOMERS table called credit_cards. Make this column a nested table of type typ_cr_card_nst. You can use the following syntax:

```
ALTER TABLE customers ADD
(credit_cards typ_cr_card_nst)
    NESTED TABLE credit_cards STORE AD c_c_store_tab;
```

5. Create a PL/SQL package that manipulates the credit_cards column in the CUSTOMERS table.

- a. Open the lab_04_05.sql file. It contains the package specification and part of the package body.
 - b. Complete the code so that the package:
 - Inserts credit card information (the credit card name and number for a specific customer)
 - Displays credit card information in an unnested format

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
         VARCHAR2);
    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/
```

Practice 4 (continued)

Using Collections (continued)

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN
        -- cards exist, add more

        -- fill in code here

        ELSE -- no cards for this customer, construct one

        -- fill in code here

        END IF;
    END update_card_info;

    PROCEDURE display_card_info
        (p_cust_id NUMBER)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;

        -- fill in code here to display the nested table
        -- contents

        END display_card_info;
    END credit_card_pkg;  -- package body
/
```

Practice 4 (continued)

Using Collections (continued)

6. Test your package with the following statements and compare the output:

```
EXECUTE credit_card_pkg.display_card_info(120)
Customer has no credit cards.
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
(120, 'Visa', 11111111)
PL/SQL procedure successfully completed.

SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;

CREDIT_CARDS(CARD_TYPE, CARD_NUM)
-----
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111))

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
(120, 'MC', 2323232323)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
(120, 'DC', 4444444)
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 4444444
PL/SQL procedure successfully completed.
```

Practice 4 (continued)

Using Collections (continued)

7. Write a SELECT statement against the credit_cards column to unnest the data. Use the TABLE expression. Use SQL*Plus.

For example, if the SELECT statement returns:

```
SELECT credit_cards
  FROM   customers
 WHERE  customer_id = 120;

CREDIT_CARDS(CARD_TYPE, CARD_NUM)
-----
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111),
                 TYP_CR_CARD('MC', 2323232323), TYP_CR_CARD('DC',
44444444))
```

rewrite it using the TABLE expression so that the results look like this:

```
-- Use the table expression so that the result is:
CUSTOMER_ID CUST_LAST_NAME CARD_TYPE          CARD_NUM
-----
120        Higgins       Visa           11111111
120        Higgins       MC            2323232323
120        Higgins       DC            44444444
```


Using Advanced Interface Methods

5

ORACLE®

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

- Execute external C programs from PL/SQL
- Execute Java programs from PL/SQL

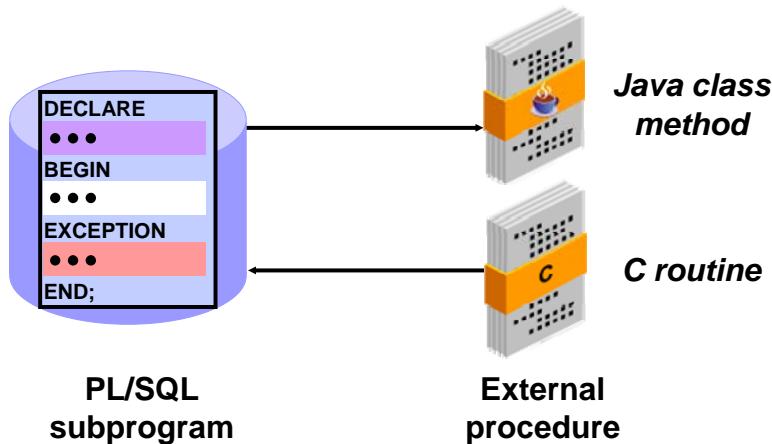


Objectives

In this lesson, you learn how to implement an external C routine from PL/SQL code and how to incorporate Java code into your PL/SQL programs.

Calling External Procedures from PL/SQL

With external procedures, you can make “callouts” and, optionally, “callbacks” through PL/SQL.



External Procedures: Overview

An *external procedure* (also called an *external routine*) is a routine stored in a dynamic link library (DLL), shared object (.so file in UNIX), or libunit in the case of a Java class method that can perform special purpose processing. You publish the routine with the base language, and then call it to perform special-purpose processing. You call the external routine from within PL/SQL or SQL. With C, you publish the routine through a library schema object, which is called from PL/SQL, that contains the compiled library file name that is stored on the operating system. With Java, publishing the routine is accomplished through creating a class libunit.

A *callout* is a call to the external procedure from your PL/SQL code.

A *callback* occurs when the external procedure calls back to the database to perform SQL operations. If the external procedure is to execute SQL or PL/SQL, it must “call back” to the database server process to get this work done.

An external procedure enables you to:

- Move computation-bound programs from the client to the server where they execute faster (because they avoid the round trips entailed in across-network communication)
- Interface the database server with external systems and data sources
- Extend the functionality of the database itself

Benefits of External Procedures

- External procedures integrate the strength and capability of different languages to give transparent access to these routines within the database.
- Extensibility: External procedures provide functionality in the database that is specific to a particular application, company, or technological area.
- Reusability: External procedures can be shared by all users on a database, and they can be moved to other databases or computers, thereby providing standard functionality with limited cost in development, maintenance, and deployment.

ORACLE®

5 - 4

Copyright © 2008, Oracle. All rights reserved.

Benefits of External Procedures

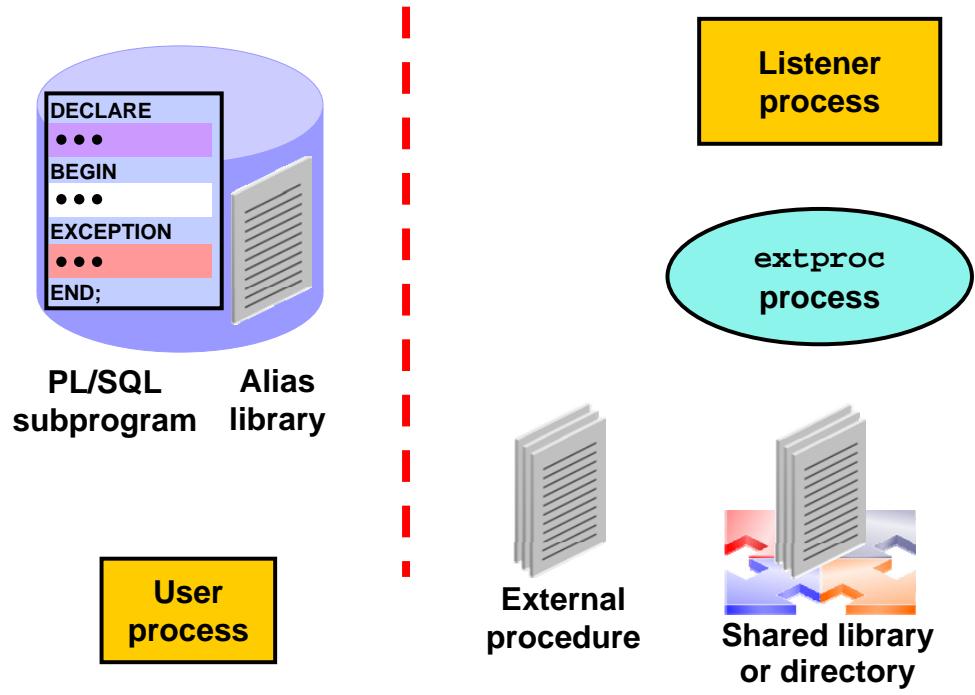
If you use the external procedure call, you can invoke an external routine by using a PL/SQL program unit. Additionally, you can integrate the powerful programming features of 3GLs with the ease of data access of SQL and PL/SQL commands.

You can extend the database and provide backward compatibility. For example, you can invoke different index or sorting mechanisms as an external procedure to implement data cartridges.

Example

A company has very complicated statistics programs written in C. The customer wants to access the data stored in an Oracle database and pass the data into the C programs. After execution of the C programs, depending on the result of the evaluations, data is inserted into the appropriate Oracle database tables.

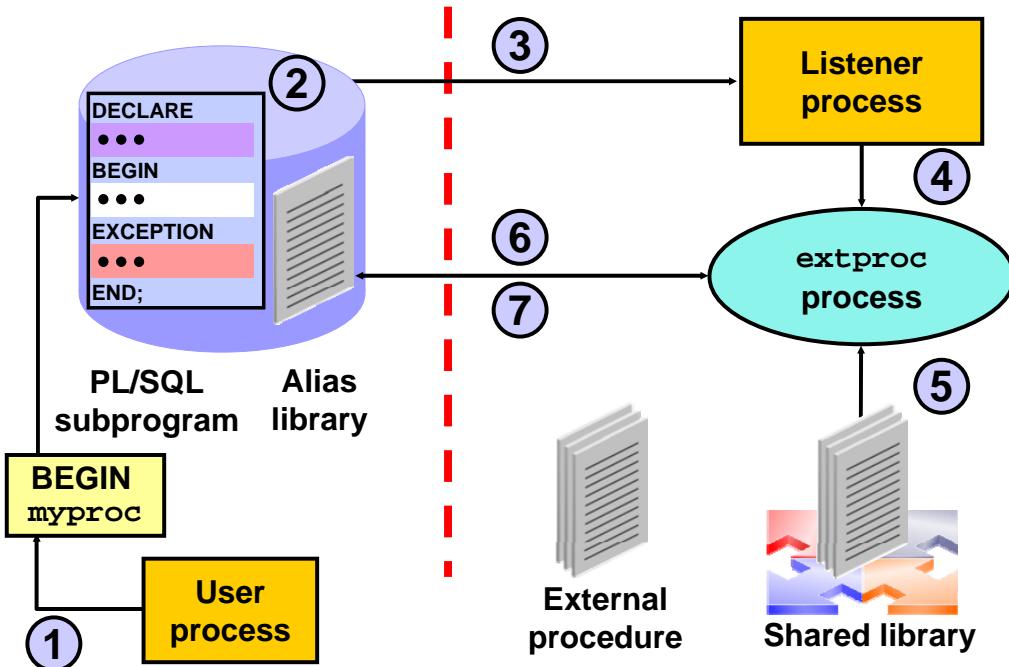
External C Procedure Components



External C Procedure Components

- **External procedure:** A unit of code written in C
- **Shared library:** An operating system file that stores the external procedure
- **Alias library:** A schema object that represents the operating system shared library
- **PL/SQL subprograms:** Packages, procedures, or functions that define the program unit specification and mapping to the PL/SQL library
- **extproc process:** A session-specific process that executes external procedures
- **Listener process:** A process that starts the extproc process and assigns it to the process executing the PL/SQL subprogram

How PL/SQL Calls a C External Procedure



How PL/SQL Calls a C External Procedure

1. The user process invokes a PL/SQL program.
2. The server process executes a PL/SQL subprogram, which looks up the alias library.
3. The PL/SQL subprogram passes the request to the listener.
4. The listener process spawns the extproc process. The extproc process remains active throughout your Oracle session until you log off.
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.

The extproc Process

- The extproc process services the execution of external procedures for the duration of the session until the user logs off.
- Each session uses a different extproc process to execute external procedures.
- The listener must be configured to allow the server to be associated with the extproc process.
- The listener must be on the same machine as the server.

ORACLE®

5 - 7

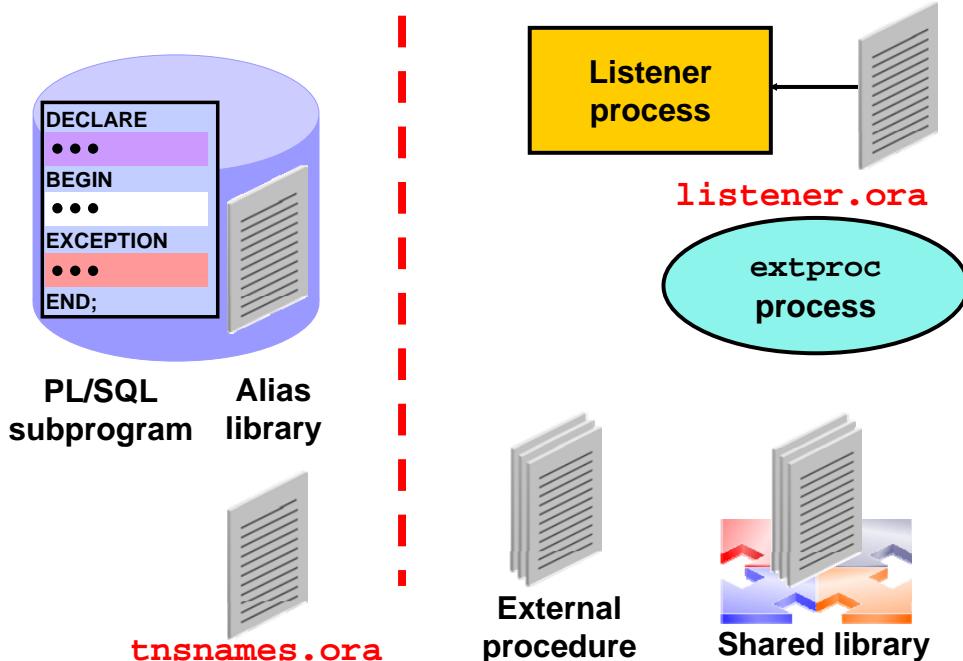
Copyright © 2008, Oracle. All rights reserved.

The extproc Process

The extproc process performs the following actions:

- **Converts PL/SQL calls to C calls:**
 - Loads the dynamic library
- **Executes the external procedures:**
 - Raises exceptions if necessary
 - Converts C back to PL/SQL
 - Sends arguments or exceptions back to the server process

The Listener Process



5 - 8

Copyright © 2008, Oracle. All rights reserved.

ORACLE

The Listener Process

When the Oracle server executes the external procedure, the request is passed to the listener process, which spawns an extproc process that executes the call to the external procedure.

This listener returns the information to the server process. A single extproc process is created for each session. The listener process starts the extproc process. The external procedure resides in a dynamic library. The Oracle Database Server runs the extproc process to load the dynamic library and to execute the external procedure.

3GL Call Dependencies: Example

Libraries are objects with the following dependencies:

Given library L1 and procedure P1, which depends on L1, when the procedure P1 is executed, library L1 is loaded, and the corresponding external library is dynamically loaded. P1 can now use the external library handle and call the appropriate external functions.

If L1 is dropped, P1 is invalidated and needs to be recompiled.

Development Steps for External C Procedures

- 1.** Create and compile the external procedure in 3GL.
- 2.** Link the external procedure with the shared library at the operating system level.
- 3.** Create an alias library schema object to map to the operating system's shared library.
- 4.** Grant execute privileges on the library.
- 5.** Publish the external C procedure by creating the PL/SQL subprogram unit specification, which references the alias library.
- 6.** Execute the PL/SQL subprogram that invokes the external procedure.

ORACLE®

5 - 9

Copyright © 2008, Oracle. All rights reserved.

Development Steps for External C Procedures

Steps 1 and 2 vary according to the operating system. Consult your operating system or the compiler documentation. After these steps are completed, you create an alias library schema object that identifies the operating system's shared library within the server. Any user who needs to execute the C procedure requires execute privileges on the library. Within your PL/SQL code, you map the C arguments to the PL/SQL parameters, and execute the PL/SQL subprogram that invokes the external routine.

Development Steps for External C Procedures

1. 2. *Varies for each operating system; consult documentation.*
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;
```

ORACLE®

5 - 10

Copyright © 2008, Oracle. All rights reserved.

Creating the Alias Library

An alias library is a database object that is used to map to an external shared library. An external procedure that you want to use needs to be stored in a DLL or a shared object library (SO) operating system file. The DBA controls access to the DLL or SO files by using the CREATE LIBRARY statement to create a schema object called an alias library that represents the external file. The DBA must give you EXECUTE privileges on the library object so that you can publish the external procedure, and then call it from a PL/SQL program.

Steps

- 1, 2. Steps 1 and 2 vary for each operating system. Consult your operating system or the compiler documentation.
3. Create an alias library object by using the CREATE LIBRARY command:

```
CONNECT /as sysdba
```

```
CREATE OR REPLACE LIBRARY c_utility  
AS 'd:\labs\labs\calc_tax.dll';
```

The example shows the creation of a database object called `c_utility`, which references the location of the file and the name of the operating system file, `calc_tax.dll`.

Creating the Alias Library (continued)

4. Grant EXECUTE privilege on the library object:

```
GRANT EXECUTE ON c_utility TO OE;
```

5. Publish the external C routine.

6. Call the external C routine from PL/SQL.

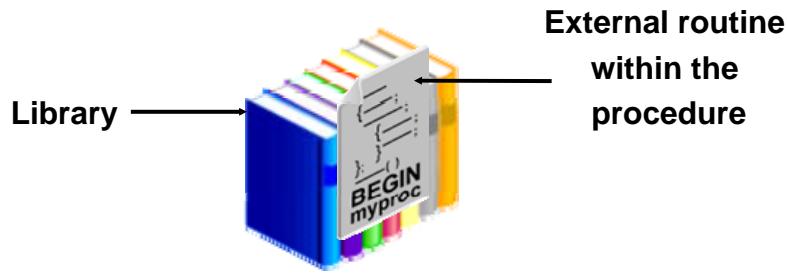
Dictionary Information

The alias library definitions are stored in the USER_LIBRARIES and ALL_LIBRARIES data dictionary views.

Development Steps for External C Procedures

Publish the external procedure in PL/SQL through call specifications:

- The body of the subprogram contains the external routine registration.
- The external procedure runs on the same machine.
- Access is controlled through the alias library.



ORACLE®

5 - 12

Copyright © 2008, Oracle. All rights reserved.

Method to Access a Shared Library Through PL/SQL

You can access a shared library by specifying the alias library in a PL/SQL subprogram. The PL/SQL subprogram then calls the alias library.

- The body of the subprogram contains the external procedure registration.
- The external procedure runs on the same machine.
- Access is controlled through the alias library.

You can publish the external procedure in PL/SQL by:

- Identifying the characteristics of the C procedure to the PL/SQL program
- Accessing the library through PL/SQL

The package specification does not require changes. You do not need definitions for the external procedure.

The Call Specification

Call specifications enable:

- Dispatching the appropriate C or Java target procedure
- Data type conversions
- Parameter mode mappings
- Automatic memory allocation and cleanup
- Purity constraints to be specified, where necessary, for packaged functions that are called from SQL
- Calling Java methods or C procedures from database triggers
- Location flexibility

ORACLE®

5 - 13

Copyright © 2008, Oracle. All rights reserved.

The Call Specification

The current way to publish external procedures is through call specifications. Call specifications enable you to call external routines from other languages. Although the specification is designed for intercommunication between SQL, PL/SQL, C, and Java, it is accessible from any base language that can call these languages.

To use an existing program as an external procedure, load, publish, and then call it.

Call specifications can be specified in any of the following locations:

- Stand-alone PL/SQL procedures and functions
- PL/SQL package specifications
- PL/SQL package bodies
- Object type specifications
- Object type bodies

Note: For functions that have the RESTRICT_REFERENCES pragma, use the TRUST option.

The SQL engine cannot analyze those functions to determine whether they are free from side effects. The TRUST option makes it easier to call the Java and C procedures.

The Call Specification

- 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]);
```

ORACLE®

5 - 14

Copyright © 2008, Oracle. All rights reserved.

Publishing an External C Routine

You create the PL/SQL procedure or function and use the IS | AS LANGUAGE C to publish the external C procedure. The external body contains the external routine information.

Syntax Definitions

where:	LANGUAGE	Is the language in which the external routine was written (defaults to C)
	LIBRARY libname	Is the name of the library database object
	NAME "C_function_name"	Represents the name of the C function; if omitted, the external procedure name must match the name of the PL/SQL subprogram
	CALLING STANDARD	Specifies the Windows NT calling standard (C or Pascal) under which the external routine was compiled (defaults to C)
	WITH CONTEXT	Specifies that a context pointer is passed to the external routine for callbacks
	parameters	Identifies arguments passed to the external routine

The 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]
```



The PARAMETER Clause

The foreign parameter list can be used to specify the position and the types of arguments, as well as to indicate whether they should be passed by value or by reference.

Syntax Definitions

where:	formal_parameter_name [INDICATOR]	Is the name of the PL/SQL parameter that is being passed to the external routine; the INDICATOR keyword is used to map a C parameter whose value indicates whether the PL/SQL parameter is null
	RETURN INDICATOR	Corresponds to the C parameter that returns a null indicator for the function
	CONTEXT	Specifies that a context pointer will be passed to the external routine
	BY REFERENCE	In C, you can pass IN scalar parameters by value (the value is passed) or by reference (a pointer to the value is passed). Use BY REFERENCE to pass the parameter by reference.
	External_datatype	Is the external data type that maps to a C data type

Note: The PARAMETER clause is optional if the mapping of the parameters is done on a positional basis, and indicators, reference, and context are not needed.

Publishing an External C Routine

Example

- Publish a C function called `calc_tax` from a PL/SQL function.

```
CREATE 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);
```

ORACLE®

5 - 16

Copyright © 2008, Oracle. All rights reserved.

Example

You have an external C function called `calc_tax` that takes in one argument, the total sales amount. The function returns the tax amount calculated at 8%. The prototype for your `calc_tax` function is as follows:

```
int calc_tax (n);
```

To publish the `calc_tax` function in a stored PL/SQL function, use the `AS LANGUAGE C` clause within the function definition. The `NAME` identifies the name of the C function. Double quotation marks are used to preserve the case of the function defined in the C program. The `LIBRARY` identifies the library object that locates the C file. The `PARAMETERS` clause is not needed in this example, because the mapping of the parameters is done on a positional basis.

Executing the External Procedure

1. Create and compile the external procedure in 3GL.
2. Link the external procedure with the shared library at the operating system level.
3. Create an alias library schema object to map to the operating system's shared library.
4. Grant execute privileges on the library.
5. Publish the external C procedure by creating the PL/SQL subprogram unit specification, which references the alias library.
6. Execute the PL/SQL subprogram that invokes the external procedure.

ORACLE®

5 - 17

Copyright © 2008, Oracle. All rights reserved.

Executing the External Procedure: Example

Here is a simple example of invoking the external routine:

```
BEGIN  
    DBMS_OUTPUT.PUT_LINE(tax_amt(100));  
END;
```

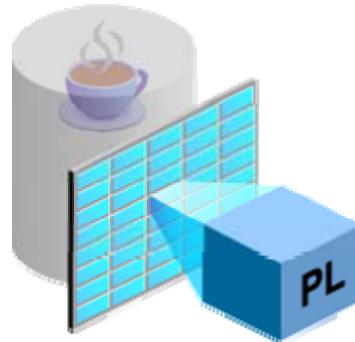
You can call the function in a cursor FOR loop or in any location where a PL/SQL function call is allowed:

```
DECLARE  
    CURSOR cur_orders IS  
        SELECT order_id, order_total  
        FROM   orders;  
        v_tax  NUMBER(8,2);  
BEGIN  
    FOR order_record IN cur_orders  
    LOOP  
        v_tax := tax_amt(order_record.order_total);  
        DBMS_OUTPUT.PUT_LINE('Total tax: ' || v_tax);  
    END LOOP;  
END;
```

Java: Overview

The Oracle database can store Java classes and Java source, which:

- Are stored in the database as procedures, functions, or triggers
- Run inside the database
- Manipulate data



ORACLE®

5 - 18

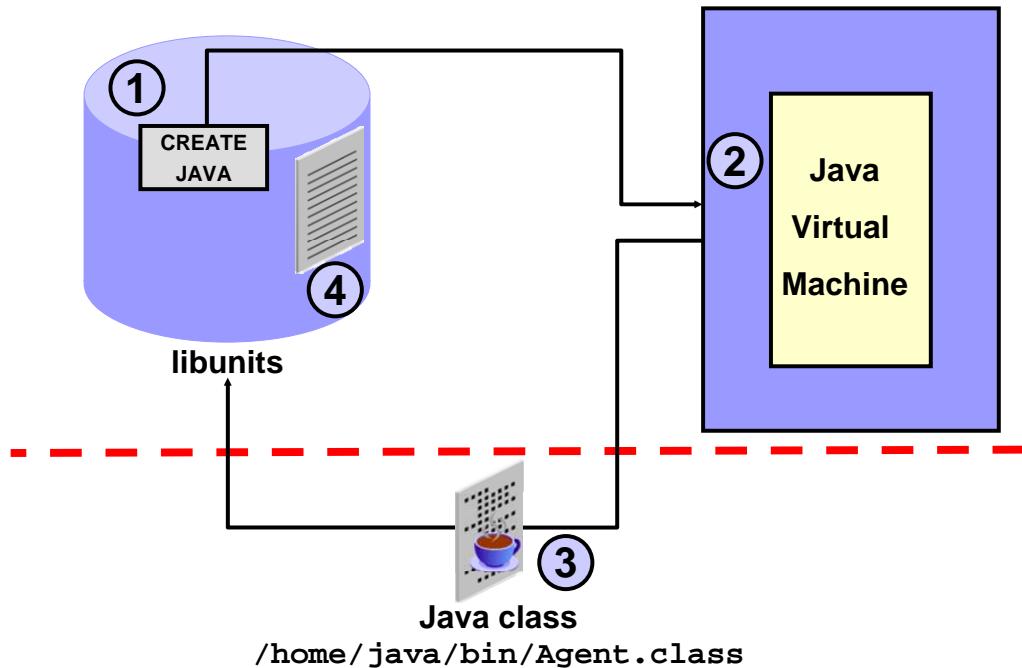
Copyright © 2008, Oracle. All rights reserved.

Java: Overview

The Oracle database can store Java classes (.class files) and Java source code (.java files), which are stored in the database as procedures, functions, or triggers. These classes can manipulate data but cannot display graphical user interface (GUI) elements such as Abstract Window Toolkit (AWT) or Swing components. Running Java inside the database helps these Java classes to be called many times and manipulate large amounts of data without the processing and network overhead that comes with running on the client machine.

You must write these named blocks, and then define them by using the `loadjava` command or the SQL `CREATE FUNCTION`, `CREATE PROCEDURE`, `CREATE TRIGGER`, or `CREATE PACKAGE` statements.

Calling a Java Class Method by Using PL/SQL



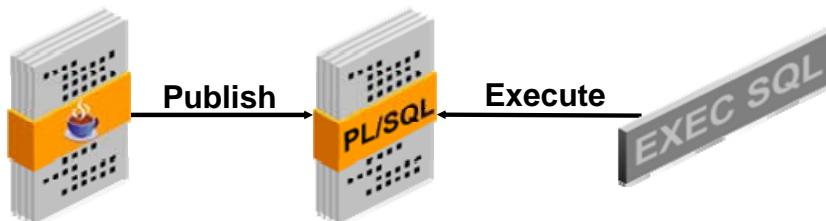
Calling a Java Class Method by Using PL/SQL

The `loadjava` command-line utility uploads the Java binaries and resources into a system-generated database table. It then uses the `CREATE JAVA` statement to load the Java files into the RDBMS libunits. You can upload the Java files from file systems, Java IDEs, intranets, or the Internet.

When the `CREATE JAVA` statement is invoked, the Java Virtual Machine library manager on the server loads the Java binaries and resources from the local `BFILE`s or `LOB` columns into the RDBMS libunits. Libunits can be considered analogous to the DLLs written in C, although they map one-to-one with Java classes, whereas DLLs can contain multiple routines.

Development Steps for Java Class Methods

1. Upload the Java file.
2. Publish the Java class method by creating the PL/SQL subprogram unit specification that references the Java class methods.
3. Execute the PL/SQL subprogram that invokes the Java class method.



ORACLE®

Steps for Using Java Class Methods

Similar to using external C routines, the following steps are required to complete the setup before executing the Java class method from PL/SQL:

1. Upload the Java file. This takes an external Java binary file and stores the Java code in the database.
2. Publish the Java class method by creating the PL/SQL subprogram unit specification that references the Java class methods.
3. Execute the PL/SQL subprogram that invokes the Java class method.

Loading Java Class Methods

1. Upload the Java file.

- At the operating system, 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 Factorial.java
```

- To load the Java class file, use:

```
>loadjava -user oe/oe Factorial.class
```

- If you load the Java source file, you do not need to load the Java class file.

ORACLE®

5 - 21

Copyright © 2008, Oracle. All rights reserved.

Loading Java Class Methods

Java classes and their methods are stored in RDBMS libunits where the Java sources, binaries, and resources can be loaded.

Use the `loadjava` command-line utility to load and resolve the Java classes. Using the `loadjava` utility, you can upload the Java source, class, or resource files into an Oracle database, where they are stored as Java schema objects. You can run `loadjava` from the command line or from an application.

After the file is loaded, it is visible in the data dictionary views.

```
SELECT object_name, object_type FROM    user_objects
WHERE  object_type like 'J%';
OBJECT_NAME          OBJECT_TYPE
-----
Factorial           JAVA CLASS
Factorial           JAVA SOURCE

SELECT text FROM    user_source WHERE name = 'Factorial';
TEXT
-----
public class Factorial {
    public static int calcFactorial (int n) {
        if (n == 1) return 1;
        else return n * calcFactorial (n - 1) ;    }}
```

Publishing a Java Class Method

2. Publish the Java class method by creating the PL/SQL subprogram unit specification that references the Java class methods.
 - 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.

```
CREATE OR REPLACE
{ PROCEDURE procedure_name [(parameter_list)]
| FUNCTION function_name [(parameter_list]...)]
  RETURN datatype}
  regularbody | externalbody
END;
↓
{IS | AS} LANGUAGE JAVA
  NAME 'method_fullname (java_type_fullname
    [, java_type_fullname]...)
    [return java_type_fullname]';
```

ORACLE®

Publishing a Java Class Method

The publishing of Java class methods is specified in the AS LANGUAGE clause. This call specification identifies the appropriate Java target routine, data type conversions, parameter mode mappings, and purity constraints. You can publish value-returning Java methods as functions and void Java methods as procedures.

Publishing a Java Class Method

- Example:

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

- Java method definition:

```
public class Factorial {
    public static int calcFactorial (int n) {
        if (n == 1) return 1;
        else return n * calcFactorial (n - 1) ;
    }
}
```

ORACLE®

5 - 23

Copyright © 2008, Oracle. All rights reserved.

Example

You want to publish a Java method named calcFactorial that returns the factorial of its argument, as shown above:

- The PL/SQL function plstojavafac_fun is created to identify the parameters and the Java characteristics.
- The NAME clause string uniquely identifies the Java method
- The parameter named N corresponds to the int argument

Executing the Java Routine

1. Upload the Java file.
2. Publish the Java class method by creating the PL/SQL subprogram unit specification that references the Java class methods.
3. Execute the PL/SQL subprogram that invokes the Java class method.

ORACLE®

5 - 24

Copyright © 2008, Oracle. All rights reserved.

Example (continued)

You can call the calcFactorial class method by using the following command:

```
EXECUTE DBMS_OUTPUT.PUT_LINE(plstojavafac_fun (5));
```

```
Anonymous block completed
120
```

Alternatively, to execute a SELECT statement from the DUAL table:

```
SELECT plstojavafac_fun (5)
FROM dual;
```

```
PLSTOJAVAFAc_FUN(5)
-----
120
```

Creating Packages for Java Class Methods

```
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)';
```

ORACLE®

5 - 25

Copyright © 2008, Oracle. All rights reserved.

Creating Packages for Java Class Methods

The examples in the slide create a package specification and body named Demo_pack.

The package is a container structure. It defines the specification of the PL/SQL procedure named plsToJ_InSpec_proc.

Note that you cannot tell whether this procedure is implemented by PL/SQL or by way of an external procedure. The details of the implementation appear only in the package body in the declaration of the procedure body.

Summary

In this lesson, you should have learned how to:

- Use external C routines and call them from your PL/SQL programs
- Use Java methods and call them from your PL/SQL programs



Summary

You can embed calls to external C programs from your PL/SQL programs by publishing the external routines in a PL/SQL block. You can take external Java programs and store them in the database to be called from PL/SQL functions, procedures, and triggers.

Practice 5: Overview

This practice covers the following topics:

- Writing programs to interact with C routines
- Writing programs to interact with Java code



Practice 5: Overview

In this practice, you write two PL/SQL programs: One program calls an external C routine and the second program calls a Java routine.

Use the OE schema for this practice.

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”

Practice 5

Using External C Routines

An external C routine definition is created for you. The .c file is stored in the D:\labs\labs directory. This function returns the tax amount based on the total sales figure that is passed to the function as a parameter. The .c file is named calc_tax.c. The function is defined as:

```
__declspec(dllexport)
int calc_tax(n)
int n;
{
    int tax;
    tax = (n*8)/100;
    return (tax);
}
```

1. A DLL file called calc_tax.dll was created for you. Copy the file from the D:\labs\labs directory into your D:\app\Administrator\product\11.1.0\db_1\BIN directory.
2. As the SYS user, create the alias library object. Name the library object c_code and define its path as:
connect / as sysdba

CREATE OR REPLACE LIBRARY c_code
AS 'd:\app\Administrator\product\11.1.0\db_1\bin\calc_tax.dll';
/
3. Grant execute privilege on the library to the OE user by executing the following command:
GRANT EXECUTE ON c_code TO OE;
4. Publish the external C routine.
As the OE user, create a function named call_c. This function has one numeric parameter and it returns a binary integer. Identify the AS LANGUAGE, LIBRARY, and NAME clauses of the function.
5. Create a procedure to call the call_c function that was created in the previous step.
Name this procedure C_OUTPUT. It has one numeric parameter. Include a DBMS_OUTPUT.PUT_LINE statement so that you can view the results returned from your C function.
6. Set SERVEROUTPUT ON and execute the C_OUTPUT procedure.

Practice 5 (continued)

Calling Java from PL/SQL

A Java method definition is created for you. The method accepts a 16-digit credit card number as the argument and returns the formatted credit card number (4 digits followed by a space). The name of the .class file is FormatCreditCardNo.class. The method is defined as:

```
public class FormatCreditCardNo
{
    public static final void formatCard(String[] cardno)
    {
        int count=0, space=0;
        String oldcc=cardno[0];
        String[] newcc= {" "};
        while (count<16)
        {
            newcc[0]+= oldcc.charAt(count);
            space++;
            if (space ==4)
            {   newcc[0]+=" "; space=0;   }
            count++;
        }
        cardno[0]=newcc [0];
    }
}
```

7. Load the .java source file.
8. Publish the Java class method by defining a PL/SQL procedure named CCFORMAT. This procedure accepts one IN OUT parameter.

Use the following definition for the NAME parameter:

```
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
```

9. Execute the Java class method. Define one SQL*Plus or SQL Developer variable, initialize it, and use the EXECUTE command to execute the CCFORMAT procedure. Your output should match the PRINT output as shown below.

```
EXECUTE ccformat(:x);
```

```
PRINT x
X
-----
1234 5678 1234 5678
```




Implementing Fine-Grained Access Control for VPD

ORACLE®

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

- Describe the process of fine-grained access control
- Implement and test fine-grained access control



Objectives

In this lesson, you learn about the security features in the Oracle Database from an application developer's standpoint.

For more information about these features, refer to *Oracle Supplied PL/SQL Packages and Types Reference*, *Oracle Label Security Administrator's Guide*, *Oracle Single Sign-On Application Developer's Guide*, and *Oracle Security Overview*.

Lesson Agenda

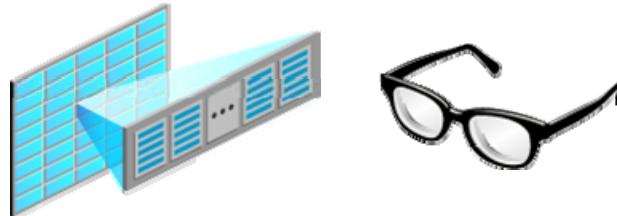
- Describing the process of fine-grained access control
- Implementing and testing fine-grained access control



Fine-Grained Access Control: Overview

Fine-grained access control:

- Enables you to enforce security through a low level of granularity
- Restricts users to viewing only “their” information
- Is implemented through a security policy attached to tables
- Is implemented by highly privileged system DBAs, perhaps in coordination with developers
- Dynamically modifies user statements to fit the policy



ORACLE®

6 - 4

Copyright © 2008, Oracle. All rights reserved.

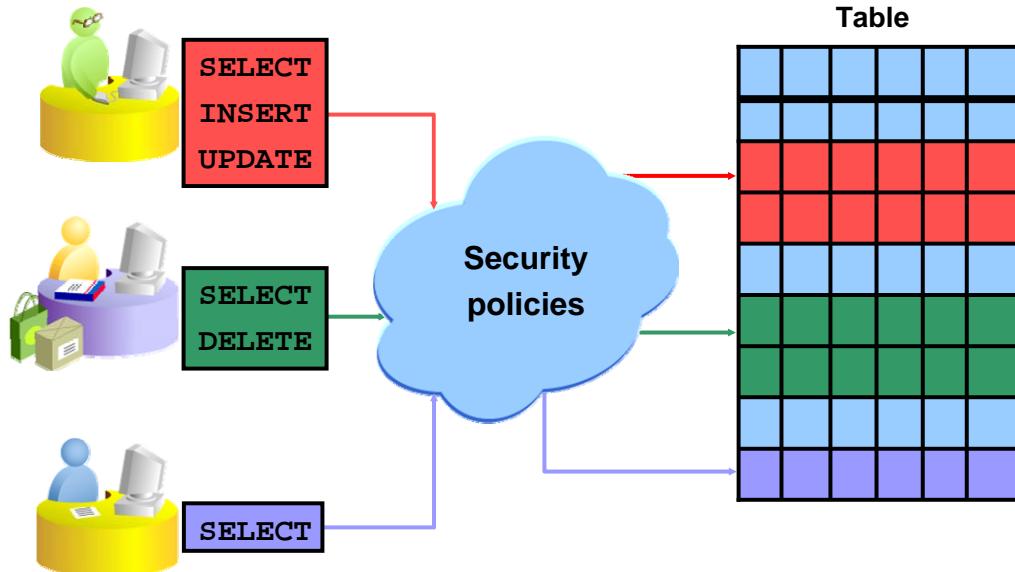
Fine-Grained Access Control: Overview

Fine-grained access control enables you to build applications that enforce security rules (or policies) at a low level of granularity. For example, you can use it to restrict customers who access the Oracle server to see only their own account, physicians to see only the records of their own patients, or managers to see only the records of employees who work for them.

When you use fine-grained access control, you create security policy functions attached to the table or view on which you based your application. When a user enters a data manipulation language (DML) statement on that object, the Oracle server dynamically modifies the user’s statement—transparently to the user—so that the statement implements the correct access control.

Fine-grained access is also known as a virtual private database (VPD), because it implements row-level security, essentially giving users access to their own private database. Fine-grained means at the individual row level.

Identifying Fine-Grained Access Features



6 - 5

Copyright © 2008, Oracle. All rights reserved.

ORACLE®

Features

You can use fine-grained access control to implement security rules called policies with functions, and then associate those security policies with tables or views. The database server automatically enforces those security policies, no matter how the data is accessed.

A security policy is a collection of rules needed to enforce the appropriate privacy and security rules in the database itself, making it transparent to users of the data structure.

Attaching security policies to tables or views, rather than to applications, provides greater security, simplicity, and flexibility.

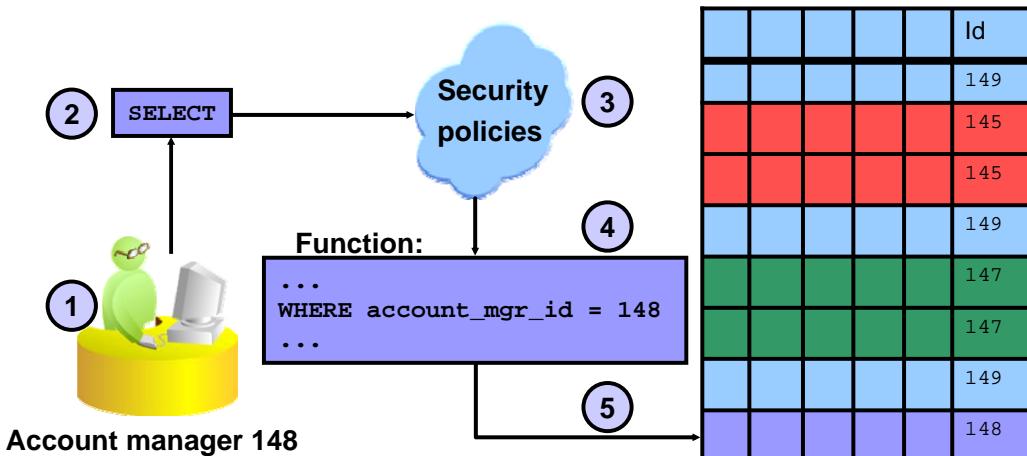
You can:

- Use different policies for SELECT, INSERT, UPDATE, and DELETE statements
- Use security policies only where you need them
- Use multiple policies for each table, including building on top of base policies in packaged applications
- Distinguish policies between different applications by using policy groups

How Fine-Grained Access Works

Implement the policy on the CUSTOMERS table:

“Account managers can see only their own customers.”



How Fine-Grained Access Works

To implement a virtual private database so that account managers can see only their own customers, you must do the following:

1. Create a function to add a `WHERE` clause identifying a selection criterion to a user's SQL statement.
2. Have the user (the account manager) enter a SQL statement.
3. Implement the security policy through the function that you created. The Oracle server calls the function automatically.
4. Dynamically modify the user's statement through the function.
5. Execute the dynamically modified statement.

How Fine-Grained Access Works

- You write a function to return the account manager ID:

```
account_mgr_id := (SELECT account_mgr_id  
                      FROM   customers  
                     WHERE account_mgr_id =  
                           SYS_CONTEXT ('userenv','session_user'));
```

- The account manager user enters a query:

```
SELECT customer_id, cust_last_name, cust_email  
      FROM   customers;
```

- The query is modified with the function results:

```
SELECT customer_id, cust_last_name, cust_email  
      FROM   orders  
     WHERE account_mgr_id = (SELECT account_mgr_id  
                               FROM   customers  
                              WHERE account_mgr_id =  
                                    SYS_CONTEXT ('userenv','session_user'));
```

ORACLE®

6 - 7

Copyright © 2008, Oracle. All rights reserved.

How Fine-Grained Access Works (continued)

Fine-grained access control is based on a dynamically modified statement. In the example in the slide, the user enters a broad query against the CUSTOMERS table that retrieves customer names and email names for a specific account manager. The Oracle server calls the function to implement the security policy. This modification is transparent to the user. It results in successfully restricting access to other customers' information, displaying only the information relevant to the account manager.

Note: The SYS_CONTEXT function returns a value for an attribute, in this case, connection attributes. This is explained in detail in the following pages.

Why Use Fine-Grained Access?

To implement the business rule “Account managers can see only their own customers,” you have three options:

Option	Comment
Modify all existing application code to include a predicate (a WHERE clause) for all SQL statements.	Does not ensure privacy enforcement outside the application. Also, all application code may need to be modified in the future as business rules change.
Create views with the necessary Predicates, and then create synonyms with the same name as the table names for these views.	This can be difficult to administer, especially if there are a large number of views to track and manage.
Create a VPD for each of the account managers by creating policy functions to generate dynamic predicates. These predicates can then be applied across all objects.	This option offers the best security without major administrative overheads and it also ensures complete privacy of information. 

ORACLE®

6 - 8

Copyright © 2008, Oracle. All rights reserved.

Why Use Fine-Grained Access?

There are other methods by which you can implement the business rule “Account managers can see only their own customers.” The options are listed above. However, by using fine-grained access, you implement security without major overheads.

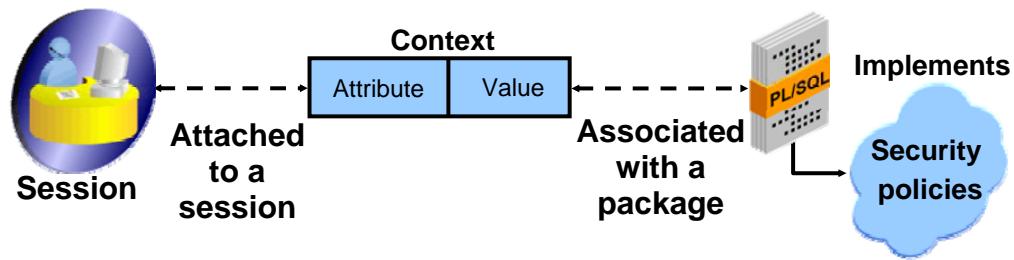
Lesson Agenda

- Describing the process of fine-grained access control
- Implementing and testing fine-grained access control



Using an Application Context

- An application context is used to facilitate the implementation of fine-grained access control.
- It is a named set of attribute/value pairs associated with a PL/SQL package.
- Applications can have their own application-specific contexts.
- Users cannot change their application's context.



Using an Application Context

An application context:

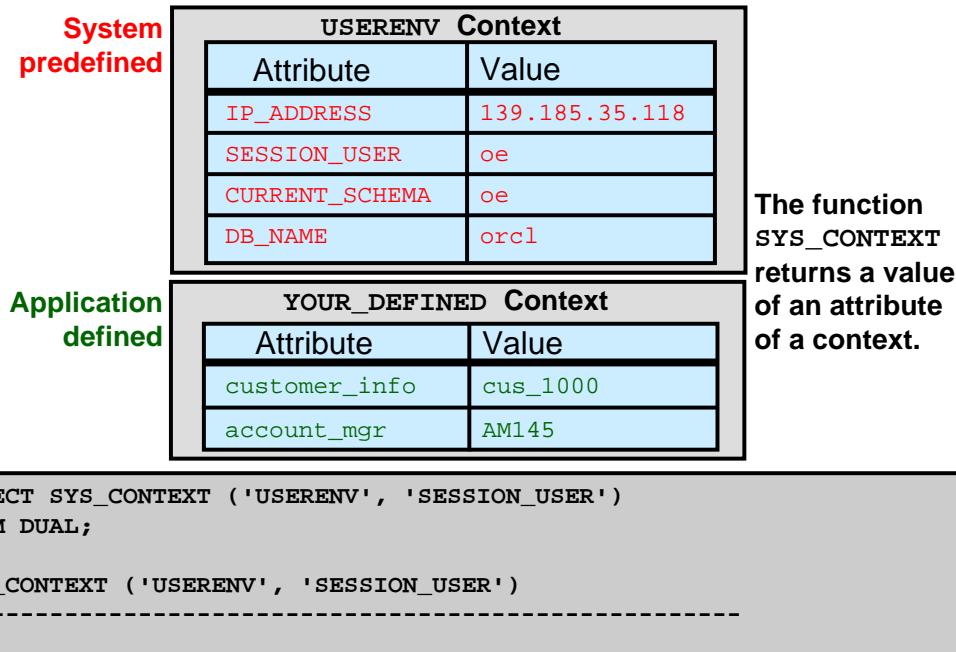
- Is a named set of attribute/value pairs associated with a PL/SQL package
- Is attached to a session
- Enables you to implement security policies with functions, and then associate them with applications

A context is a named set of attribute/value pairs that are global to your session. You can define an application context, name it, and associate a value with that context with a PL/SQL package. An application context enables you to write applications that draw upon certain aspects of a user's session information. It provides a way to define, set, and access attributes that an application can use to enforce access control—specifically, fine-grained access control.

Most applications contain information about the basis on which access is to be limited. In an order entry application, for example, you limit the customers' access to their own orders (ORDER_ID) and customer number (CUSTOMER_ID). Or, you may limit account managers (ACCOUNT_MGR_ID) to view only their own customers. These values can be used as security attributes. Your application can use a context to set values that are accessed within your code and used to generate WHERE clause predicates for fine-grained access control.

An application context is owned by SYS.

Using an Application Context



6 - 11

Copyright © 2008, Oracle. All rights reserved.

ORACLE®

Using an Application Context (continued)

A predefined application context named `USERENV` has a predefined list of attributes. Predefined attributes can be very useful for access control. You find the values of the attributes in a context by using the `SYS_CONTEXT` function. Although the predefined attributes in the `USERENV` application context are accessed with the `SYS_CONTEXT` function, you cannot change them.

With the `SYS_CONTEXT` function, you pass the context name and the attribute name. The attribute value is returned.

The following statement returns the name of the database that is being accessed:

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

```
SYS_CONTEXT('USERENV', 'DB_NAME')
-----
ORCL
```

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 CONTEXT order_ctx USING oe.orders_app_pkg;
```

```
Context created.
```

ORACLE®

Creating an Application Context

For fine-grained access where you want account manager to view only their customers, customers to view only their information, and sales representatives to view only their orders, you can create a context called ORDER_CTX and define for it the ACCOUNT_MGR, CUST_ID and SALE REP attributes.

Because a context is associated with a PL/SQL package, you need to name the package that you are associating with the context. This package does not need to exist at the time of context creation.

Setting a Context

- Use the supplied package procedure DBMS_SESSION.SET_CONTEXT to set a value for an attribute within a context.

```
DBMS_SESSION.SET_CONTEXT('context_name',
                          'attribute_name',
                          'attribute_value')
```

- Set the attribute value in the package that is associated with the context.

```
CREATE OR REPLACE PACKAGE orders_app_pkg
...
BEGIN
    DBMS_SESSION.SET_CONTEXT('ORDER_CTX',
                            'ACCOUNT_MGR',
                            v_user)
...

```

ORACLE®

6 - 13

Copyright © 2008, Oracle. All rights reserved.

Setting a Context

When a context is defined, you can use the DBMS_SESSION.SET_CONTEXT procedure to set a value for an attribute within a context. The attribute is set in the package that is associated with the context.

```
CREATE OR REPLACE PACKAGE orders_app_pkg
IS
    PROCEDURE set_app_context;
END;
/
CREATE OR REPLACE PACKAGE BODY orders_app_pkg
IS
    c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';
    PROCEDURE set_app_context
    IS
        v_user VARCHAR2(30);
    BEGIN
        SELECT user INTO v_user FROM dual;
        DBMS_SESSION.SET_CONTEXT
            (c_context, 'ACCOUNT_MGR', v_user);
    END;
END;
/
```

Setting a Context (continued)

In the example on the previous page, the ORDER_CTX context has the ACCOUNT_MGR attribute set to the current user logged (determined by the USER function).

For this example, assume that users AM145, AM147, AM148, and AM149 exist. As each user logs on and the DBMS_SESSION.SET_CONTEXT is invoked, the attribute value for that ACCOUNT_MGR is set to the user ID.

```
GRANT EXECUTE ON oe.orders_app_pkg
    TO AM145, AM147, AM148, AM149;

CONNECT AM145/oracle
Connected.

EXECUTE oe.orders_app_pkg.set_app_context

SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;

SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')
-----
AM145
```

If you switch the user ID, the attribute value is also changed to reflect the current user.

```
CONNECT AM147/oracle
Connected.

EXECUTE oe.orders_app_pkg.set_app_context

SELECT SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR') FROM dual;

SYS_CONTEXT('ORDER_CTX', 'ACCOUNT_MGR')
-----
AM147
```

Implementing a Policy

Follow these steps:

1. Set up a driving context.

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

2. Create the package associated with the context that you defined in step 1. In the package:
 - a. Set the context.
 - b. Define the predicate.
3. Define the policy.
4. Set up a logon trigger to call the package at logon time and set the context.
5. Test the policy.

ORACLE®

6 - 15

Copyright © 2008, Oracle. All rights reserved.

Implementing a Policy

In this example, assume that the users AM145, AM147, AM148, and AM149 exist. Next, create a context and a package associated with the context. The package will be owned by OE.

Step 1: Set Up a Driving Context

Use the CREATE CONTEXT syntax to create a context.

```
CONNECT /AS sysdba
```

```
CREATE CONTEXT order_ctx USING oe.orders_app_pkg;
```

Step 2: Creating the Package

```
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
/
```



Implementing a Policy (continued)

Step 2: Create a Package

In the OE schema, the ORDERS_APP_PKG is created. This package contains three routines:

- **show_app_context:** For learning and testing purposes, this procedure displays a context attribute and value.
- **set_app_context:** This procedure sets a context attribute to a specific value.
- **the_predicate:** This function builds the predicate (the WHERE clause) that controls the rows visible in the CUSTOMERS table to a user. (Note that this function requires two input parameters. An error occurs when the policy is implemented if you exclude these two parameters.)

Implementing a Policy (continued)

Step 2: Create a Package (continued)

```
CREATE OR REPLACE PACKAGE BODY orders_app_pkg
IS
    c_context CONSTANT VARCHAR2(30) := 'ORDER_CTX';
    c_attrib  CONSTANT VARCHAR2(30) := 'ACCOUNT_MGR';

    PROCEDURE show_app_context
    IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE('Type: ' || c_attrib ||
            ' - ' || SYS_CONTEXT(c_context, c_attrib));
    END show_app_context;

    PROCEDURE set_app_context
    IS
        v_user VARCHAR2(30);
    BEGIN
        SELECT user INTO v_user FROM dual;
        DBMS_SESSION.SET_CONTEXT
            (c_context, c_attrib, v_user);
    END set_app_context;

    FUNCTION the_predicate
    (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2
    IS
        v_context_value VARCHAR2(100) :=
            SYS_CONTEXT(c_context, c_attrib);
        v_restriction VARCHAR2(2000);
    BEGIN
        IF v_context_value LIKE 'AM%' THEN
            v_restriction :=
                'ACCOUNT_MGR_ID =
                    SUBSTR(''' || v_context_value || ''', 3, 3)';
        ELSE
            v_restriction := null;
        END IF;
        RETURN v_restriction;
    END the_predicate;

    END orders_app_pkg; -- package body
    /
```

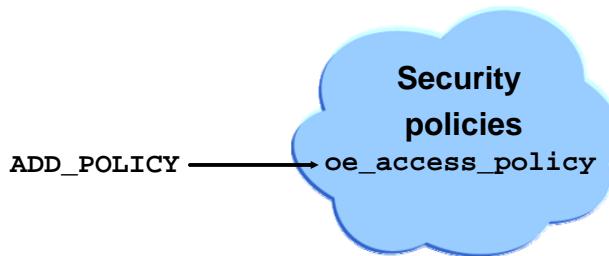
Note that the THE_PREDICATE function builds the WHERE clause and stores it in the V_RESTRICTION variable. If the SYS_CONTEXT function returns an attribute value that starts with AM, the WHERE clause is built with ACCOUNT_MGR_ID = *the last three characters of the attribute value*. If the user is AM145, the WHERE clause will be:

```
WHERE account_mgr_id = 145
```

Step 3: Defining the Policy

Use the DBMS_RLS package:

- It contains the fine-grained access administrative interface.
- It adds a fine-grained access control policy to a table or view.
- You use the ADD_POLICY procedure to add a fine-grained access control policy to a table or view.



ORACLE®

6 - 18

Copyright © 2008, Oracle. All rights reserved.

Implementing a Policy (continued)

The DBMS_RLS package contains the fine-grained access control administrative interface. The package holds several procedures. But the package by itself does nothing until you add a policy. To add a policy, you use the ADD_POLICY procedure within the DBMS_RLS package.

Note: DBMS_RLS is available only with the Enterprise Edition.

Step 3: Define the Policy

The DBMS_RLS.ADD_POLICY procedure adds a fine-grained access control policy to a table or view. The procedure causes the current transaction, if any, to commit before the operation is carried out. However, this does not cause a commit first if it is inside a DDL event trigger. These are the parameters for the ADD_POLICY procedure:

```
DBMS_RLS.ADD_POLICY (
    object_schema    IN VARCHAR2 := NULL,
    object_name      IN VARCHAR2,
    policy_name      IN VARCHAR2,
    function_schema  IN VARCHAR2 := NULL,
    policy_function  IN VARCHAR2,
    statement_types  IN VARCHAR2 := NULL,
    update_check     IN BOOLEAN := FALSE,
    enable           IN BOOLEAN := TRUE);
```

Implementing a Policy (continued)

Step 3: Define the Policy (continued)

Parameter	Description
OBJECT_SCHEMA	Schema containing the table or view (logon user, if NULL).
OBJECT_NAME	Name of the table or view to which the policy is added.
POLICY_NAME	Name of the policy to be added. For any table or view, each POLICY_NAME must be unique.
FUNCTION_SCHEMA	Schema of the policy function (logon user, if NULL).
POLICY_FUNCTION	Name of the function that generates a predicate for the policy. If the function is defined within a package, the name of the package must be present.
STATEMENT_TYPES	Statement types that the policy will apply. It can be any combination of SELECT, INSERT, UPDATE, and DELETE. The default is to apply all these statement types to the policy.
UPDATE_CHECK	Optional argument for the INSERT or UPDATE statement types. The default is FALSE. Setting update_check to TRUE causes the server to also check the policy against the value after INSERT or UPDATE.
ENABLE	Indicates whether the policy is enabled when it is added. The default is TRUE.

The following is a list of the procedures contained in the DBMS_RLS package. For detailed information, refer to the *PL/SQL Packages and Types Reference 11g Release 1 (11.1)*.

Procedure	Description
ADD_POLICY	Adds a fine-grained access control policy to a table or view
DROP_POLICY	Drops a fine-grained access control policy from a table or view
REFRESH_POLICY	Causes all the cached statements associated with the policy to be reparsed
ENABLE_POLICY	Enables or disables a fine-grained access control policy
CREATE_POLICY_GROUP	Creates a policy group
ADD_GROUPED_POLICY	Adds a policy associated with a policy group
ADD_POLICY_CONTEXT	Adds the context for the active application
DELETE_POLICY_GROUP	Deletes a policy group
DROP_GROUPED_POLICY	Drops a policy associated with a policy group
DROP_POLICY_CONTEXT	Drops a driving context from the object so that it has one less driving context
ENABLE_GROUPED_POLICY	Enables or disables a row-level group security policy
REFRESH_GROUPED_POLICY	Reparses the SQL statements associated with a refreshed policy

Step 3: Defining the Policy

```
CONNECT /as sysdba
```

```
DECLARE
BEGIN
  DBMS_RLS.ADD_POLICY (
    'OE',
    'CUSTOMERS',
    'OE_ACCESS_POLICY',
    'OE',
    '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

ORACLE®

6 - 20

Copyright © 2008, Oracle. All rights reserved.

Implementing a Policy (continued)

Step 3: Define the Policy (continued)

The security policy OE_ACCESS_POLICY is created and added with the DBMS_RLS.ADD_POLICY procedure. The predicate function that defines how the policy is to be implemented is associated with the policy being added.

This example specifies that whenever a SELECT, UPDATE, or DELETE statement on the OE.CUSTOMERS table is executed, the predicate function return result is appended to the WHERE clause.

Step 4: Setting Up a Logon Trigger

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

```
CONNECT /as sysdba

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



Implementing a Policy (continued)

Step 4: Set Up a Logon Trigger

After the context is created, the security package is defined, the predicate is defined, and the policy is defined, you create a logon trigger to implement fine-grained access control. This trigger causes the context to be set as each user is logged on.

Example Results

Data in the CUSTOMERS table:

```
CONNECT as oe
SELECT COUNT(*), account_mgr_id
FROM customers
GROUP BY account_mgr_id;

COUNT(*) ACCOUNT_MGR_ID
-----
111 145
76 147
58 148
74 149
1
```

```
CONNECT AM148/oracle
SELECT customer_id, customer_last_name
FROM oe.customers;

CUSTOMER_ID CUSTOMER_LAST_NAME
-----
...
58 rows selected.
```

ORACLE®

6 - 22

Copyright © 2008, Oracle. All rights reserved.

Example Results

The AM148 user who logs on sees only those rows in the CUSTOMERS table that are defined by the predicate function. The user can issue SELECT, UPDATE, and DELETE statements against the CUSTOMERS table, but only the rows defined by the predicate function can be manipulated.

```
UPDATE oe.customers
SET credit_limit = credit_limit + 5000
WHERE customer_id = 101;

0 rows updated.
```

The AM148 user does not have access to customer ID 101. Customer ID 101 has the account manager of 145. Any updates, deletes, or selects attempted by user AM148 on customers that do not have him or her as an account manager are not performed. It is as though these customers do not exist.

Data Dictionary Views

- USER_POLICIES
- ALL_POLICIES
- DBA_POLICIES
- ALL_CONTEXT
- DBA_CONTEXT



ORACLE®

6 - 23

Copyright © 2008, Oracle. All rights reserved.

Data Dictionary Views

You can query the data dictionary views to find information about the policies available in your schema.

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;

NAMESPACE          SCHEMA          PACKAGE
-----            -----
ORDER_CTX          OE              ORDERS_APP_PKG
```

ORACLE®

6 - 24

Copyright © 2008, Oracle. All rights reserved.

Using the ALL_CONTEXT Dictionary View

You can use the ALL_CONTEXT dictionary view to view information about the contexts to which you have access. In the slide, the NAMESPACE column is equivalent to the context name.

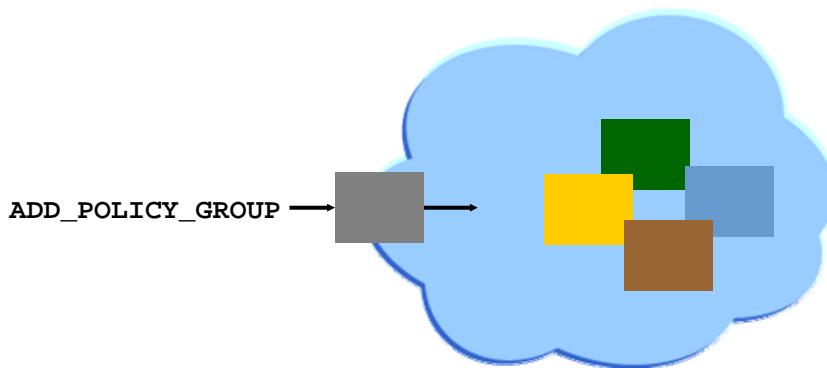
You can use the ALL_POLICIES dictionary view to view information about the policies to which you have access. In the following example, information is shown about the OE_ACCESS_POLICY policy.

```
SELECT object_name, policy_name, pf_owner, package,
       function, sel, ins, upd, del
  FROM all_policies;
```

OBJECT_NAME	POLICY_NAME
PF_OWNER	PACKAGE
FUNCTION	SEL INS UPD DEL
CUSTOMERS	OE_ACCESS_POLICY
OE	ORDERS_APP_PKG
THE_PREDICATE	YES NO YES YES

Policy Groups

- Indicate a set of policies that belong to an application
- Are set up by a DBA through an application context called a driving context
- Use the DBMS_RLS package to administer the security policies



ORACLE®

Policy Groups

Policy groups were introduced in Oracle9*i*, release 1 (9.0.1). The DBA designates an application context, called a driving context, to indicate the policy group in effect. When tables or views are accessed, the fine-grained access control engine looks up the driving context to determine the policy group in effect and enforces all associated policies that belong to that policy group.

The PL/SQL DBMS_RLS package enables you to administer your security policies and groups. Using this package, you can add, drop, enable, disable, and refresh the policy groups that you create.

More About Policies

- **SYS_DEFAULT** is the default policy group:
 - The **SYS_DEFAULT** group may or may not contain policies.
 - All policies belong to **SYS_DEFAULT** by default.
 - You cannot drop the **SYS_DEFAULT** policy group.
- Use `DBMS_RLS.CREATE_POLICY_GROUP` to create a new group.
- Use `DBMS_RLS.ADD_GROUPED_POLICY` to add a policy associated with a policy group.
- You can apply multiple driving contexts to the same table or view.

ORACLE®

6 - 26

Copyright © 2008, Oracle. All rights reserved.

More About Policies

A policy group is a set of security policies that belong to an application. You can designate an application context (known as a driving context) to indicate the policy group in effect. When the tables or views are accessed, the server looks up the driving context to determine the policy group in effect. It enforces all associated policies that belong to that policy group.

By default, all policies belong to the **SYS_DEFAULT** policy group. The policies defined in this group for a particular table or view are always executed along with the policy group specified by the driving context. The **SYS_DEFAULT** policy group may or may not contain policies. If you attempt to drop the **SYS_DEFAULT** policy group, an error is raised. If you add policies associated with two or more objects to the **SYS_DEFAULT** policy group, each such object has a separate **SYS_DEFAULT** policy group associated with it.

For example, the CUSTOMERS table in the OE schema has one **SYS_DEFAULT** policy group, and the ORDERS table in the OE schema has a different **SYS_DEFAULT** policy group associated with it.

```
SYS_DEFAULT
  - policy1 (OE/CUSTOMERS)
  - policy3 (OE/CUSTOMERS)
SYS_DEFAULT
  - policy2 (OE/ORDERS)
```

More About Policies (continued)

When adding a policy to a table or view, you can use the DBMS_RLS.ADD_GROUPED_POLICY interface to specify the group to which the policy belongs. To specify which policies are effective, you can add a driving context using the DBMS_RLS.ADD_POLICY_CONTEXT interface. If the driving context returns an unknown policy group, an error is returned.

If the driving context is not defined, all policies are executed. Likewise, if the driving context is NULL, the policies from all policy groups are enforced. Thus, an application that accesses the data cannot bypass the security setup module (that sets up the application context) to avoid applicable policies.

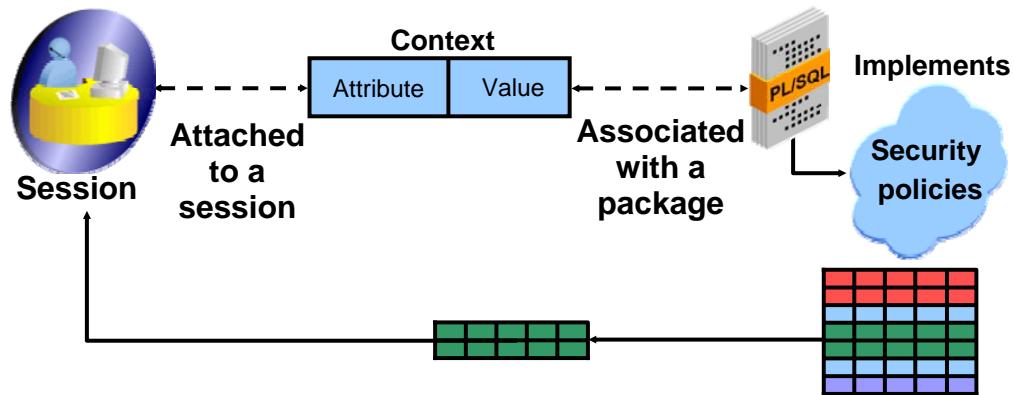
You can apply multiple driving contexts to the same table or view, and each of them are processed individually. Thus, you can configure multiple active sets of policies to be enforced.

You can create a new policy by using the DBMS_RLS package either from the command line or programmatically, or you can access the Oracle Policy Manager graphical user interface in Oracle Enterprise Manager.

Summary

In this lesson, you should have learned how to:

- Describe the process of fine-grained access control
- Implement and test fine-grained access control



Summary

In this lesson, you should have learned about fine-grained access control and the steps required to implement a virtual private database.

Practice 6: 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



Practice 6: Overview

In this practice, you implement and test fine-grained access control.

Practice 6: Implementing Fine-Grained Access Control for VPD

In this practice, you define an application context and security policy to implement the policy: “Sales Representatives can see only their own order information in the ORDERS table.” You create sales representative IDs to test the success of your implementation.

Examine the definition of the ORDERS table and the ORDER count for each sales representative:

```
DESCRIBE orders
Name          Null?    Type
-----        -----
ORDER_ID      NOT NULL NUMBER(12)
ORDER_DATE    NOT NULL TIMESTAMP(6) WITH LOCAL TIME ZONE
ORDER_MODE    VARCHAR2(8)
CUSTOMER_ID   NOT NULL NUMBER(6)
ORDER_STATUS  NUMBER(2)
ORDER_TOTAL   NUMBER(8,2)
SALES REP ID NUMBER(6)
PROMOTION_ID NUMBER(6)

SELECT sales_rep_id, count(*)
FROM   orders
GROUP BY sales_rep_id;

SALES REP ID COUNT(*)
-----
153          5
154          10
155          5
156          5
158          7
159          7
160          6
161          13
163          12
35

10 rows selected.
```

1. Use your OE connection. Examine and then run the `lab_06_01.sql` script.

This script creates the sales representative ID accounts with appropriate privileges to access the database.

2. Set up an application context:

- a. Connect to the database as SYSDBA before creating this context.
- b. Create an application context named `sales_orders_ctx`.
- c. Associate this context to the `oe.sales_orders_pkg`.

Practice 6 (continued)

3. Connect as OE.

a. Examine this package specification:

```
CREATE OR REPLACE PACKAGE sales_orders_pkg
IS
  PROCEDURE set_app_context;
  FUNCTION the_predicate
    (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2;
END sales_orders_pkg;      -- package spec
/
```

b. Create this package specification and the package body in the OE schema.

c. When you create the package body, set up two constants as follows:

```
c_context CONSTANT VARCHAR2(30) := 'SALES_ORDERS_CTX';
c_attrib  CONSTANT VARCHAR2(30) := 'SALES REP';
```

d. Use these constants in the SET_APP_CONTEXT procedure to set the application context to the current user.

4. Connect as SYSDBA and define the policy.

a. Use DBMS_RLS.ADD_POLICY to define the policy.

b. Use these specifications for the parameter values:

```
object_schema    OE
object_name      ORDERS
policy_name      OE_ORDERS_ACCESS_POLICY
function_schema  OE
policy_function  SALES_ORDERS_PKG.THE_PREDICATE
statement_types   SELECT, INSERT, UPDATE, DELETE
update_check     FALSE,
enable           TRUE);
```

5. Connect as SYSDBA and create a logon trigger to implement fine-grained access control.

You can call the trigger SET_ID_ON_LOGON. This trigger causes the context to be set as each user is logged on.

Practice 6 (continued)

6. Test the fine-grained access implementation. Connect as your SR user and query the ORDERS table. For example, your results should match:

```
CONNECT sr153/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES REP_ID      COUNT( * )
-----
153           5

CONNECT sr154/oracle

SELECT sales_rep_id, COUNT(*)
FROM   orders
GROUP BY sales_rep_id;

SALES REP_ID      COUNT( * )
-----
154           10
```

Note

During debugging, you may need to disable or remove some of the objects created for this lesson.

- If you need to disable the logon trigger, issue this command:
`ALTER TRIGGER set_id_on_logon DISABLE;`
- If you need to remove the policy that you created, issue this command:
`EXECUTE DBMS_RLS.DROP_POLICY('OE', 'ORDERS', -
'OE_ORDERS_ACCESS_POLICY')`



Manipulating Large Objects

ORACLE®

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

- Compare and contrast LONG and large object (LOB) data types
- Create and maintain LOB data types
- Differentiate between internal and external LOBs
- Use the DBMS_LOB PL/SQL package
- Describe the use of temporary LOBs

ORACLE®

7 - 2

Copyright © 2008, Oracle. All rights reserved.

Objectives

Databases have long been used to store large objects. However, the mechanisms built into databases have never been as useful as the large object (LOB) data types that have been provided since Oracle8. This lesson describes the characteristics of the new data types, comparing and contrasting them with the earlier data types. Examples, syntax, and issues regarding the LOB types are also presented.

Note: A LOB is a data type and should not be confused with an object type.

Lesson Agenda

- Introduction to LOBS
- Managing BFILEs by using the DBMS_LOB package
- Migrating LONG data types to LOBS
- Manipulating LOB data
- Using temporary LOBS

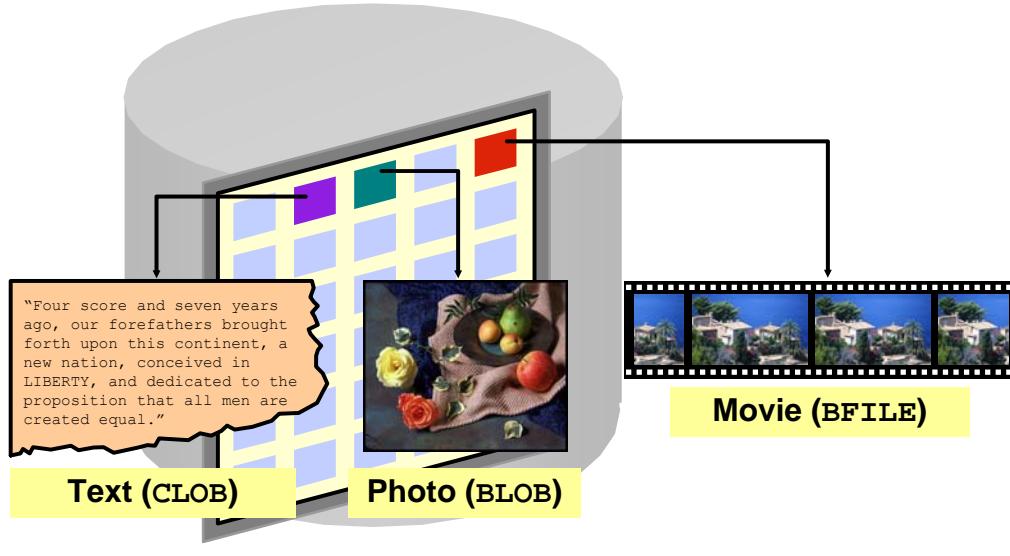
ORACLE®

7 - 3

Copyright © 2008, Oracle. All rights reserved.

What Is a LOB?

LOBs are used to store large, unstructured data such as text, graphic images, films, and sound waveforms.



LOB: Overview

A LOB is a data type that is used to store large, unstructured data such as text, graphic images, video clippings, and so on. Structured data, such as a customer record, may be a few hundred bytes large, but even small amounts of multimedia data can be thousands of times larger. Also, multimedia data may reside in operating system (OS) files, which may need to be accessed from a database.

There are four large object data types:

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

LOBs are characterized in two ways, according to their interpretations by the Oracle server (binary or character) and their storage aspects. LOBs can be stored internally (inside the database) or in host files. There are two categories of LOBs:

- **Internal LOBs (CLOB, NCLOB, BLOB):** Stored in the database
- **External files (BFILE):** Stored outside the database

LOB: Overview (continued)

Oracle Database 10g performs implicit conversion between CLOB and VARCHAR2 data types. The other implicit conversions between LOBs are not possible. For example, if the user creates a table T with a CLOB column and a table S with a BLOB column, the data is not directly transferable between these two columns.

BFILEs can be accessed only in read-only mode from an Oracle server.

Contrasting LONG and LOB Data Types

LONG and LONG RAW	LOB
Single LONG column per table	Multiple LOB columns per table
Up to 2 GB	Up to 4 GB
SELECT returns data	SELECT returns locator
Data stored inline	Data stored inline or out-of-line
Sequential access to data	Random access to data

ORACLE®

7 - 6

Copyright © 2008, Oracle. All rights reserved.

LONG and LOB Data Types

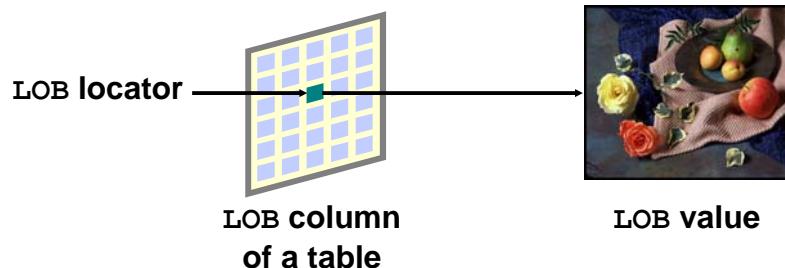
The LONG and LONG RAW data types were previously used for unstructured data, such as binary images, documents, or geographical information. These data types are superseded by the LOB data types. Oracle Database 10g provides a LONG-to-LOB API to migrate from LONG columns to LOB columns. The following bulleted list compares the LOB functionality with the older types, where LONGs refer to LONG and LONG RAW, and LOBs refer to all LOB data types:

- A table can have multiple LOB columns and object type attributes. A table can have only one LONG column.
- The maximum size of LONGs is 2 GB; LOBs can be up to 4 GB.
- LOBs return the locator; LONGs return the data.
- LOBs store a locator in the table and the data in a different segment, unless the data is less than 4,000 bytes; LONGs store all data in the same data block. In addition, LOBs allow data to be stored in a separate segment and tablespace, or in a host file.
- LOBs can be object type attributes; LONGs cannot be object type attributes.
- LOBs support random piecewise access to the data through a file-like interface; LONGs are restricted to sequential piecewise access.

The TO_LOB function can be used to convert LONG and LONG RAW values in a column to LOB values. You use this in the SELECT list of a subquery in an INSERT statement.

Components of a LOB

The LOB column stores a locator to the LOB's value.



ORACLE®

7 - 7

Copyright © 2008, Oracle. All rights reserved.

Components of a LOB

There are two parts to a LOB:

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

Regardless of where the LOB value is stored, a locator is stored in the row. You can think of a LOB locator as a pointer to the actual location of the LOB value.

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

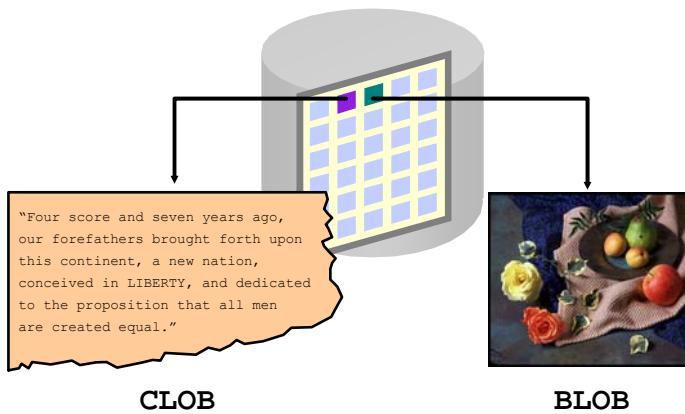
When a user creates an internal LOB, the value is stored in the LOB segment and a locator to the out-of-line LOB value is placed in the LOB column of the corresponding row in the table.

External LOBs store the data outside the database, so only a locator to the LOB value is stored in the table.

To access and manipulate LOBs without SQL data manipulation language (DML), you must create a LOB locator. The programmatic interfaces operate on the LOB values by using these locators in a manner similar to OS file handles.

Internal LOBs

The LOB value is stored in the database.



ORACLE®

7 - 8

Copyright © 2008, Oracle. All rights reserved.

Internal LOBs

An internal LOB is stored in the Oracle server. A BLOB, NCLOB, or CLOB can be one of the following:

- An attribute of a user-defined type
- A column in a table
- A bind or host variable
- A PL/SQL variable, parameter, or result

Internal LOBs can take advantage of Oracle features, such as:

- Concurrency mechanisms
- Redo logging and recovery mechanisms
- Transactions with COMMIT or ROLLBACK

The BLOB data type is interpreted by the Oracle server as a bitstream, similar to the LONG RAW data type.

The CLOB data type is interpreted as a single-byte character stream.

The NCLOB data type is interpreted as a multiple-byte character stream, based on the byte length of the database national character set.

Managing Internal LOBs

- To interact fully with LOB, file-like interfaces are provided in:
 - PL/SQL package DBMS_LOB
 - Oracle Call Interface (OCI)
 - Oracle Objects for object linking and embedding (OLE)
 - Pro*C/C++ and Pro*COBOL precompilers
 - Java Database Connectivity (JDBC)
- The Oracle server provides some support for LOB management through SQL.

ORACLE®

7 - 9

Copyright © 2008, Oracle. All rights reserved.

Managing Internal LOBs

To manage an internal LOB, perform the following steps:

1. Create and populate the table containing the LOB data type.
2. Declare and initialize the LOB locator in the program.
3. Use SELECT FOR UPDATE to lock the row containing the LOB into the LOB locator.
4. Manipulate the LOB with DBMS_LOB package procedures, OCI calls, Oracle Objects for OLE, Oracle precompilers, or JDBC by using the LOB locator as a reference to the LOB value. You can also manage LOBs through SQL.
5. Use the COMMIT command to make any changes permanent.

Lesson Agenda

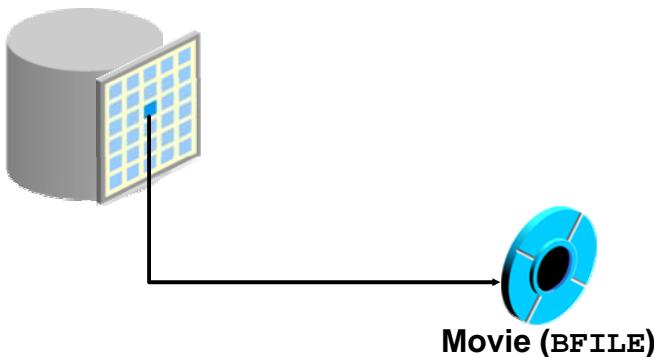
- Introduction to LOBS
- Managing BFILEs by using the DBMS_LOB package
- Migrating LONG data types to LOBS
- Manipulating LOB data
- Using temporary LOBS

ORACLE®

What Are BFILEs?

The BFILE data type supports an external or file-based large object as:

- Attributes in an object type
- Column values in a table



ORACLE®

7 - 11

Copyright © 2008, Oracle. All rights reserved.

What Are BFILEs?

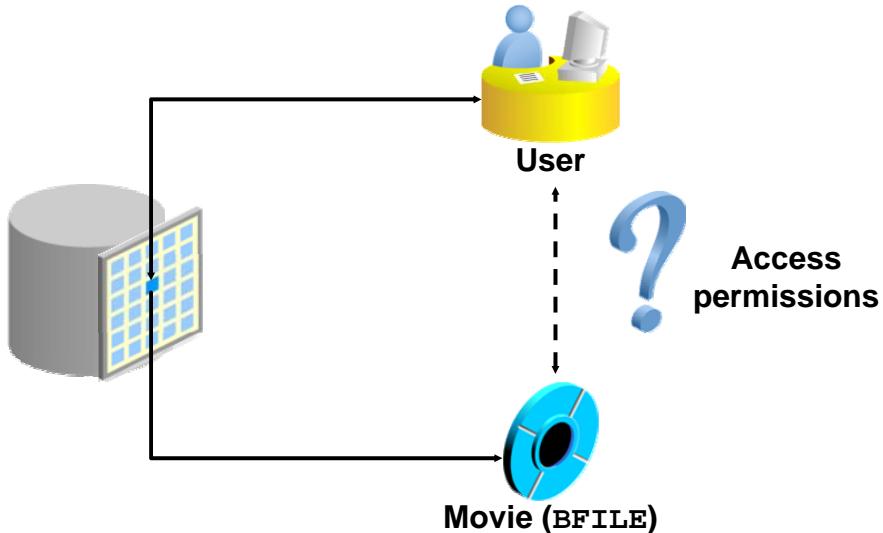
BFILEs are external large objects (LOBs) stored in OS files that are external to database tables. The BFILE data type stores a locator to the physical file. A BFILE can be in GIF, JPEG, MPEG, MPEG2, text, or other formats. The external LOBs may be located on hard disks, CD-ROMs, photo CDs, or other media, but a single LOB cannot extend from one medium or device to another. The BFILE data type is available so that database users can access the external file system. Oracle Database 10g provides:

- Definition of BFILE objects
- Association of BFILE objects with the corresponding external files
- Security for BFILEs

The remaining operations that are required for using BFILEs are possible through the DBMS_LOB package and OCI. BFILEs are read-only; they do not participate in transactions. Support for integrity and durability must be provided by the operating system. The file must be created and placed in the appropriate directory, giving the Oracle process privileges to read the file. When the LOB is deleted, the Oracle server does not delete the file. Administration of the files and the OS directory structures can be managed by the DBA, system administrator, or user. The maximum size of an external large object depends on the operating system but cannot exceed 4 GB.

Note: BFILEs are available with the Oracle8 database and later releases.

Securing BFILES



ORACLE®

7 - 12

Copyright © 2008, Oracle. All rights reserved.

Securing BFILES

Unauthenticated access to files on a server presents a security risk. Oracle Database 10g can act as a security mechanism to shield the operating system from unsecured access while removing the need to manage additional user accounts on an enterprise computer system.

File Location and Access Privileges

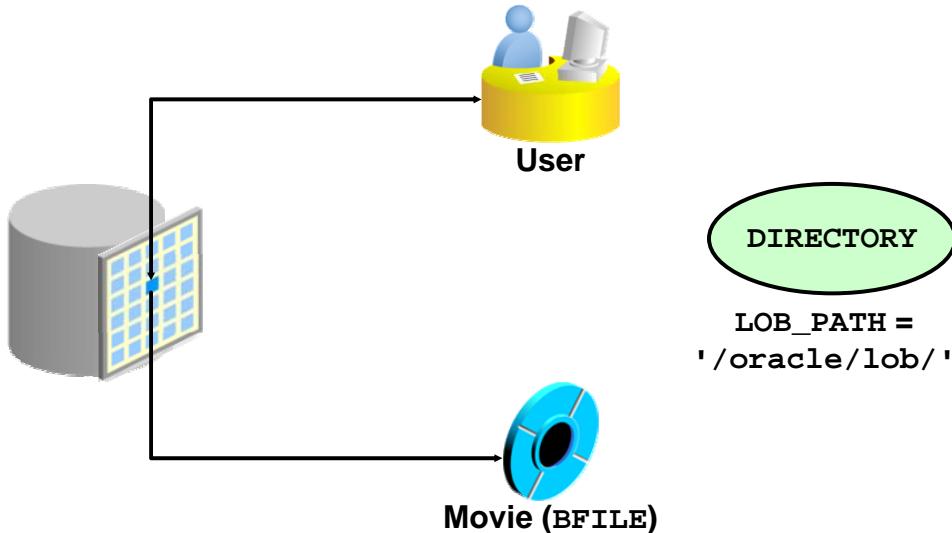
The file must reside on the machine where the database exists. A timeout to read a nonexistent BFILE is based on the OS value.

You can read a BFILE in the same way that you read an internal LOB. However, there could be restrictions related to the file itself, such as:

- Access permissions
- File system space limits
- Non-Oracle manipulations of files
- OS maximum file size

Oracle Database 10g does not provide transactional support on BFILEs. Any support for integrity and durability must be provided by the underlying file system and the OS. Oracle backup and recovery methods support only the LOB locators, not the physical BFILEs.

What Is a DIRECTORY?



What Is a DIRECTORY?

A DIRECTORY is a nonschema database object that enables the administration of access and usage of BFILEs in Oracle Database 10g.

A DIRECTORY specifies an alias for a directory on the file system of the server under which a BFILE is located. By granting users suitable privileges for these items, you can provide secure access to files in the corresponding directories on a user-by-user basis (certain directories can be made read-only, inaccessible, and so on).

Furthermore, these directory aliases can be used while referring to files (open, close, read, and so on) in PL/SQL and OCI. This provides application abstraction from hard-coded path names and gives flexibility in portably managing file locations.

The DIRECTORY object is owned by SYS and created by the DBA (or a user with the CREATE ANY DIRECTORY privilege). The directory objects have object privileges, unlike other nonschema objects. Privileges to the DIRECTORY object can be granted and revoked. Logical path names are not supported.

The permissions for the actual directory depend on the operating system. They may differ from those defined for the DIRECTORY object and could change after creation of the DIRECTORY object.

Guidelines for Creating DIRECTORY Objects

- Do not create DIRECTORY objects on paths with database files.
- Limit the number of people who are given the following system privileges:
 - CREATE ANY DIRECTORY
 - DROP ANY DIRECTORY
- All DIRECTORY objects are owned by SYS.
- Create directory paths and properly set permissions before using the DIRECTORY object so that the Oracle server can read the file.

ORACLE®

7 - 14

Copyright © 2008, Oracle. All rights reserved.

Guidelines for Creating DIRECTORY Objects

To associate an OS file with a BFILE, you should first create a DIRECTORY object that is an alias for the full path name to the OS file.

Create DIRECTORY objects by using the following guidelines:

- Directories should point to paths that do not contain database files, because tampering with these files could corrupt the database. Currently, only the READ privilege can be given for a DIRECTORY object.
- The CREATE ANY DIRECTORY and DROP ANY DIRECTORY system privileges should be used carefully and not granted to users indiscriminately.
- DIRECTORY objects are not schema objects; all are owned by SYS.
- Create the directory paths with appropriate permissions on the OS before creating the DIRECTORY object. Oracle does not create the OS path.

If you migrate the database to a different OS, you may need to change the path value of the DIRECTORY object.

Information about the DIRECTORY object that you create by using the CREATE DIRECTORY command is stored in the DBA_DIRECTORIES and ALL_DIRECTORIES data dictionary views.

Using the DBMS_LOB Package

- Working with LOBs often requires the use of the Oracle-supplied DBMS_LOB package.
- DBMS_LOB provides routines to access and manipulate internal and external LOBs.
- LOB data can be retrieved directly using SQL.
- In PL/SQL, you can define a VARCHAR2 for a CLOB and a RAW for a BLOB.

ORACLE®

7 - 15

Copyright © 2008, Oracle. All rights reserved.

Using the DBMS_LOB Package

To work with LOBs, you may need to use the DBMS_LOB package. The package does not support any concurrency control mechanism for BFILE operations. The user is responsible for locking the row containing the destination internal LOB before calling subprograms that involve writing to the LOB value. These DBMS_LOB routines do not implicitly lock the row containing the LOB.

The two constants, LOBMAXSIZE and FILE_READONLY, that are defined in the package specification are also used in the procedures and functions of DBMS_LOB; for example, use them to achieve the maximum level of purity in SQL expressions.

The DBMS_LOB functions and procedures can be broadly classified into two types: mutators and observers.

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

Using the DBMS_LOB Package

- **Modify LOB values:**

APPEND, COPY, ERASE, TRIM, WRITE, LOADFROMFILE

- **Read or examine LOB values:**

GETLENGTH, INSTR, READ, SUBSTR

- **Specific to BFILES:**

FILECLOSE, FILECLOSEALL, FILEEXISTS,

FILEGETNAME, FILEISOPEN, FILEOPEN



Using the DBMS_LOB Package (continued)

APPEND	Appends the contents of the source LOB to the destination LOB
COPY	Copies all or part of the source LOB to the destination LOB
ERASE	Erases all or part of a LOB
LOADFROMFILE	Loads BFILE data into an internal LOB
TRIM	Trims the LOB value to a specified shorter length
WRITE	Writes data to the LOB from a specified offset
GETLENGTH	Gets the length of the LOB value
INSTR	Returns the matching position of the <i>n</i> th occurrence of the pattern in the LOB
READ	Reads data from the LOB starting at the specified offset
SUBSTR	Returns part of the LOB value starting at the specified offset
FILECLOSE	Closes the file
FILECLOSEALL	Closes all previously opened files
FILEEXISTS	Checks whether the file exists on the server
FILEGETNAME	Gets the directory alias and the file name
FILEISOPEN	Checks whether the file was opened using the input BFILE locators
FILEOPEN	Opens a file

DBMS_LOB Package

- NULL parameters get NULL returns.
- Offsets:
 - BLOB, BFILE: Measured in bytes
 - CLOB, NCLOB: Measured in characters
- There are no negative values for parameters.

ORACLE®

7 - 17

Copyright © 2008, Oracle. All rights reserved.

Using the DBMS_LOB Routines

All functions in the DBMS_LOB package return NULL if any input parameters are NULL. All mutator procedures in the DBMS_LOB package raise an exception if the destination LOB/BFILE is input as NULL.

Only positive, absolute offsets are allowed. They represent the number of bytes or characters from the beginning of the LOB data from which to start the operation. The negative offsets and ranges that are observed in SQL string functions and operators are not allowed. Corresponding exceptions are raised upon violation. The default value for an offset is 1, which indicates the first byte or character in the LOB value.

Similarly, only natural number values are allowed for the amount (BUFSIZ) parameter. Negative values are not allowed.

DBMS_LOB.READ and DBMS_LOB.WRITE

```
PROCEDURE READ (
    lobsrc IN BFILE|BLOB|CLOB ,
    amount IN OUT BINARY_INTEGER,
    offset IN INTEGER,
    buffer OUT RAW|VARCHAR2 )
```

```
PROCEDURE WRITE (
    lobdst IN OUT BLOB|CLOB,
    amount IN OUT BINARY_INTEGER,
    offset IN INTEGER := 1,
    buffer IN RAW|VARCHAR2 ) -- RAW for BLOB
```

ORACLE®

7 - 18

Copyright © 2008, Oracle. All rights reserved.

DBMS_LOB.READ

Call the READ procedure to read and return piecewise a specified AMOUNT of data from a given LOB, starting from OFFSET. An exception is raised when no more data remains to be read from the source LOB. The value returned in AMOUNT is less than the one specified if the end of the LOB is reached before the specified number of bytes or characters can be read. In the case of CLOBs, the character set of data in BUFFER is the same as that in the LOB.

PL/SQL allows a maximum length of 32,767 for RAW and VARCHAR2 parameters. Ensure that the allocated system resources are adequate to support buffer sizes for the given number of user sessions. Otherwise, the Oracle server raises the appropriate memory exceptions.

Note: BLOB and BFILE return RAW; the others return VARCHAR2.

DBMS_LOB.WRITE

Call the WRITE procedure to write piecewise a specified AMOUNT of data into a given LOB, from the user-specified BUFFER, starting from an absolute OFFSET from the beginning of the LOB value.

Make sure (especially with multiple-byte characters) that the amount in bytes corresponds to the amount of buffer data. WRITE has no means of checking whether they match, and it will write AMOUNT bytes of the buffer contents into the LOB.

Managing BFILEs

The DBA or the system administrator:

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

The developer or the user:

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



Managing BFILEs

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

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

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

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

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

Preparing to Use BFILEs

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

```
md D:\Labs\DATA_FILES\MEDIA_FILES
```

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

```
CREATE DIRECTORY data_files  
AS 'D:\Labs\DATA_FILES\MEDIA_FILES';
```

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

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

ORACLE®

7 - 20

Copyright © 2008, Oracle. All rights reserved.

Preparing to Use BFILEs

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

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

Syntax definition: CREATE DIRECTORY *dir_name* AS *os_path*;

In this syntax, *dir_name* is the name of the directory database object, and *os_path* specifies the location of the BFILEs.

The slide examples show the commands to set up:

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

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

Populating BFILE Columns with SQL

- 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 using the BFILENAME function:

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

ORACLE®

7 - 21

Copyright © 2008, Oracle. All rights reserved.

Populating BFILE Columns with SQL

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

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

The BFILENAME function creates a pointer (or LOB locator) to the external file stored in a physical directory, which is assigned a directory alias name that is used in the first parameter of the function. Populate the BFILE column by using the BFILENAME function in either of the following:

- The VALUES clause of an INSERT statement
- The SET clause of an UPDATE statement

An UPDATE operation can be used to change the pointer reference target of the BFILE. A BFILE column can also be initialized to a NULL value and updated later with the BFILENAME function, as shown in the slide.

After the BFILE columns are associated with a file, subsequent read operations on the BFILE can be performed by using the PL/SQL DBMS_LOB package and OCI. However, these files are read-only when accessed through BFILES. Therefore, they cannot be updated or deleted through BFILES.

Populating a BFILE Column with PL/SQL

```
CREATE 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;
```

ORACLE®

7 - 22

Copyright © 2008, Oracle. All rights reserved.

Populating a BFILE Column with PL/SQL

The slide example shows a PL/SQL procedure called `set_video`, which accepts the name of the directory alias referencing the OS file system as a parameter, and a customer ID. The procedure performs the following tasks:

- Uses a cursor `FOR` loop to obtain each customer record
- Sets the `filename` by appending `.gif` to the customer's `first_name`
- Creates an in-memory LOB locator for the BFILE in the `file_ptr` variable
- Calls the `DBMS_LOB.FILEOPEN` procedure to verify whether the file exists, and to determine the size of the file by using the `DBMS_LOB.GETLENGTH` function
- Executes an `UPDATE` statement to write the BFILE locator value to the `video` BFILE column
- Displays the file size returned from the `DBMS_LOB.GETLENGTH` function
- Closes the file by using the `DBMS_LOB.FILECLOSE` procedure

Suppose that you execute the following call:

```
EXECUTE set_video('DATA_FILES', 844)
```

The sample result is:

```
FILE: Alice.gif SIZE: 2619802
```

Using DBMS_LOB Routines with BFILEs

The DBMS_LOB.FILEEXISTS function can check whether the file exists in the OS. The function:

- Returns 0 if the file does not exist
- Returns 1 if the file does exist

```
CREATE OR REPLACE FUNCTION get_filesize(p_file_ptr IN
OUT BFILE)
RETURN NUMBER IS
  v_file_exists BOOLEAN;
  v_length NUMBER:= -1;
BEGIN
  v_file_exists := DBMS_LOB.FILEEXISTS(p_file_ptr) = 1;
  IF v_file_exists THEN
    DBMS_LOB.FILEOPEN(p_file_ptr);
    v_length := DBMS_LOB.GETLENGTH(p_file_ptr);
    DBMS_LOB.FILECLOSE(p_file_ptr);
  END IF;
  RETURN v_length;
END;
/
```

ORACLE®

7 - 23

Copyright © 2008, Oracle. All rights reserved.

Using DBMS_LOB Routines with BFILEs

The set_video procedure on the previous page terminates with an exception if a file does not exist. To prevent the loop from prematurely terminating, you could create a function, such as get_filesize, to determine whether a given BFILE locator references a file that actually exists on the server's file system. The DBMS_LOB.FILEEXISTS function accepts the BFILE locator as a parameter and returns an INTEGER with:

- A value 0 if the physical file does not exist
- A value 1 if the physical file exists

If the BFILE parameter is invalid, one of the following three exceptions may be raised:

- NOEXIST_DIRECTORY if the directory does not exist
- NOPRIV_DIRECTORY if the database processes do not have privileges for the directory
- INVALID_DIRECTORY if the directory was invalidated after the file was opened

In the get_filesize function, the output of the DBMS_LOB.FILEEXISTS function is compared with value 1 and the result of the condition sets the BOOLEAN variable file_exists. The DBMS_LOB.FILEOPEN call is performed only if the file exists, thereby preventing unwanted exceptions from occurring. The get_filesize function returns a value of -1 if a file does not exist; otherwise, it returns the size of the file in bytes. The caller can take appropriate action with this information.

Lesson Agenda

- Introduction to LOBS
- Managing BFILEs by using the DBMS_LOB package
- Migrating LONG data types to LOBS
- Manipulating LOB data
- Using temporary LOBS

ORACLE®

7 - 24

Copyright © 2008, Oracle. All rights reserved.

Migrating from LONG to LOB

Oracle Database 10g enables the migration of LONG columns to LOB columns.

- Data migration consists of the procedure to move existing tables containing LONG columns to use LOBS:

```
ALTER TABLE [<schema>.] <table_name>
  MODIFY (<long_col_name> {CLOB | BLOB | NCLOB})
```

- Application migration consists of changing applications using LONG data types to use LOB data types instead.

ORACLE®

7 - 25

Copyright © 2008, Oracle. All rights reserved.

Migrating from LONG to LOB

Oracle Database 10g supports LONG-to-LOB migration by using an API. In data migration, existing tables that contain LONG columns need to be moved to use LOB columns. This can be done by using the ALTER TABLE command. You can use the syntax shown to:

- Modify a LONG column to a CLOB or an NCLOB column
- Modify a LONG RAW column to a BLOB column

The constraints of the LONG column (NULL and NOT NULL are the only allowed constraints) are maintained for the new LOB columns. The default value specified for the LONG column is also copied to the new LOB column. For example, you have the following table:

```
CREATE TABLE long_tab (id NUMBER, long_col LONG);
```

To change the long_col column in the long_tab table to the CLOB data type, use:

```
ALTER TABLE long_tab MODIFY (long_col CLOB);
```

For information about the limitations on LONG-to-LOB migration, refer to *Oracle Database Application Developer's Guide - Large Objects*. In application migration, the existing LONG applications change to using LOBs. You can use SQL and PL/SQL to access LONGs and LOBs. The LONG-to-LOB migration API is provided for both OCI and PL/SQL.

Migrating from LONG to LOB

- **Implicit conversion:** From LONG (LONG RAW) or a VARCHAR2 (RAW) variable to a CLOB (BLOB) variable, and vice versa.
- **Explicit conversion:**
 - TO_CLOB () converts LONG, VARCHAR2, and CHAR to CLOB.
 - TO_BLOB () converts LONG RAW and RAW to BLOB.
- **Function and procedure parameter passing:**
 - CLOBS and BLOBS are passed as actual parameters.
 - VARCHAR2, LONG, RAW, and LONG RAW are formal parameters, and vice versa.
- **LOB data is acceptable in most of the SQL and PL/SQL operators and built-in functions.**

ORACLE®

7 - 26

Copyright © 2008, Oracle. All rights reserved.

Migrating from LONG to LOB (continued)

With the new LONG-to-LOB API introduced in Oracle Database 10g, data from CLOB and BLOB columns can be referenced by regular SQL and PL/SQL statements.

Implicit assignment and parameter passing: The LONG-to-LOB migration API supports assigning a CLOB (BLOB) variable to a LONG (LONG RAW) or a VARCHAR2(RAW) variable, and vice versa.

Explicit conversion functions: In PL/SQL, the following two new explicit conversion functions were added in Oracle Database 10g to convert other data types to CLOB and BLOB as part of the LONG-to-LOB migration:

- TO_CLOB () converts LONG, VARCHAR2, and CHAR to CLOB.
- TO_BLOB () converts LONG RAW and RAW to BLOB.

Note: TO_CHAR () is enabled to convert a CLOB to a CHAR type.

Function and procedure parameter passing: This enables the use of CLOBS and BLOBS as actual parameters where VARCHAR2, LONG, RAW, and LONG RAW are formal parameters, and vice versa. In SQL and PL/SQL built-in functions and operators, a CLOB can be passed to SQL and PL/SQL VARCHAR2 built-in functions, behaving exactly like a VARCHAR2. Or, the VARCHAR2 variable can be passed into DBMS_LOB APIs, acting like a LOB locator.

Lesson Agenda

- Introduction to LOBS
- Managing BFILEs by using the DBMS_LOB package
- Migrating LONG data types to LOBS
- Manipulating LOB data
- Using temporary LOBS

ORACLE®

Initializing LOB Columns Added to a Table

- Add the LOB columns to an existing table by using ALTER TABLE.

```
ALTER TABLE customers
  ADD (resume CLOB, picture BLOB);
```

- Create a tablespace where you will put a new table with the LOB columns.

```
connect /as sysdba

CREATE TABLESPACE lob_tbs1
  DATAFILE 'lob_tbs1.dbf' SIZE 800M REUSE
  EXTENT MANAGEMENT LOCAL
  UNIFORM SIZE 64M
  SEGMENT SPACE MANAGEMENT AUTO;
```

ORACLE®

7 - 28

Copyright © 2008, Oracle. All rights reserved.

Initializing LOB Columns Added to a Table

The contents of a LOB column are stored in the LOB segment, whereas the column in the table contains only a reference to that specific storage area, called the LOB locator. In PL/SQL, you can define a variable of the LOB type, which contains only the value of the LOB locator. You can initialize the LOB locators by using the following functions:

- EMPTY_CLOB() function to a LOB locator for a CLOB column
- EMPTY_BLOB() function to a LOB locator for a BLOB column

Note: These functions create the LOB locator value and not the LOB content. In general, you use the DBMS_LOB package subroutines to populate the content. The functions are available in Oracle SQL DML, and are not part of the DBMS_LOB package.

LOB columns are defined by using SQL data definition language (DDL). You can add LOB columns to an existing table by using the ALTER TABLE statement.

You can also add LOB columns to a new table. It is recommended that you create a tablespace first, and then create the new table in that tablespace.

Initializing LOB Columns Added to a Table

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

- EMPTY_CLOB() function for a CLOB column
- EMPTY_BLOB() function for a BLOB column

```
connect oe

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);
```

ORACLE®

7 - 29

Copyright © 2008, Oracle. All rights reserved.

Initializing LOB Columns Added to a Table (continued)

The slide example shows that you can use the EMPTY_CLOB() and EMPTY_BLOB() functions in the DEFAULT option in a CREATE TABLE statement. Thus, the LOB locator values are populated in their respective columns when a row is inserted into the table and the LOB columns were not specified in the INSERT statement.

The CUSTOMER_PROFILE table is created. The PICTURE column holds the LOB data in the BasicFile format, because the storage clause identifies the format. You learn about the SecureFile format in the lesson titled “Administering SecureFile LOBs.”

You learn how to use these functions in INSERT and UPDATE statements to initialize the LOB locator values.

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 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;
```

ORACLE®

7 - 30

Copyright © 2008, Oracle. All rights reserved.

Populating LOB Columns

You can insert a value directly into a LOB column by using host variables in SQL or PL/SQL, 3GL-embedded SQL, or OCI. You can use the special EMPTY_BLOB() and EMPTY_CLOB() functions in INSERT or UPDATE statements of SQL DML to initialize a NULL or non-NUL internal LOB to empty. To populate a LOB column, perform the following steps:

1. Initialize the LOB column to a non-NULL value—that is, set a LOB locator pointing to an empty or populated LOB value. This is done by using the EMPTY_BLOB() and EMPTY_CLOB() functions.
2. Populate the LOB contents by using the DBMS_LOB package routines.

However, as shown in the slide examples, the two UPDATE statements initialize the resume LOB locator value and populate its contents by supplying a literal value. This can also be done in an INSERT statement. A LOB column can be updated to:

- Another LOB value
- A NULL value
- A LOB locator with empty contents by using the EMPTY_*LOB() built-in function

You can update the LOB by using a bind variable in embedded SQL. When assigning one LOB to another, a new copy of the LOB value is created. Use a SELECT FOR UPDATE statement to lock the row containing the LOB column before updating a piece of the LOB contents.

Writing Data to a LOB

- Create the procedure to read the MS Word files and load them into the LOB column.
- Call this procedure from the WRITE_LOB procedure (shown on the next page).

```
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;
BEGIN
  DBMS_LOB.OPEN(v_src_loc, DBMS_LOB.LOB_READONLY);
  v_amount := DBMS_LOB.GETLENGTH(v_src_loc);
  DBMS_LOBLOADFROMFILE(p_dest_loc, v_src_loc, v_amount);
  DBMS_LOB CLOSE(v_src_loc);
END loadLOBFromBFILE_proc;
```

ORACLE®

7 - 31

Copyright © 2008, Oracle. All rights reserved.

Writing Data to a LOB

The procedure shown in the slide is used to load data into the LOB column.

Before running the LOADLOBFROMBFILE_PROC procedure, you must set a directory object that identifies where the LOB files are stored externally. In this example, the Microsoft Word documents are stored in the DATA_FILES directory that was created earlier in this lesson.

The LOADLOBFROMBFILE_PROC procedure is used to read the LOB data into the PICTURE column in the CUSTOMER_PROFILES table.

In this example:

- DBMS_LOB.OPEN is used to open an external LOB in read-only mode.
- DBMS_LOB.GETLENGTH is used to find the length of the LOB value.
- DBMS_LOBLOADFROMFILE is used to load the BFILE data into an internal LOB.
- DBMS_LOB CLOSE is used to close the external LOB.

Note: The LOADLOBFROMBFILE_PROC procedure shown in the slide can be used to read both SecureFile and BasicFile formats. SecureFile LOBs is discussed in the lesson titled “Administering SecureFile LOBs.”

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;
```

ORACLE®

7 - 32

Copyright © 2008, Oracle. All rights reserved.

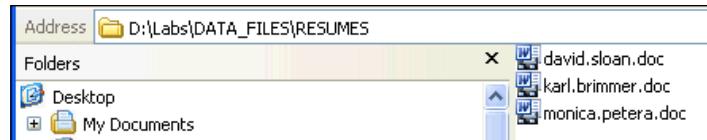
Writing Data to a LOB (continued)

Before you write data to the LOB column, you must make the LOB column non-NULL. The LOB column must contain a locator that points to an empty or populated LOB value. You can initialize a BLOB column value by using the EMPTY_BLOB() function as a default predicate.

The code shown in the slide uses the INSERT statement to initialize the locator. The LOADLOBFROMBFILE routine is then called and the LOB column value is inserted.

The write and read performance statistics for LOB storage is captured through output messages.

Writing Data to a LOB



```
CREATE DIRECTORY resume_files  
AS 'D:\Labs\DATA_FILES\RESUMES';
```

```
set serveroutput on  
set verify on  
set term on  
set linesize 200  
  
timing start load_data  
execute write_lob('karl.brimmer.doc', 'RESUME_FILES')  
execute write_lob('monica.petera.doc', 'RESUME_FILES')  
execute write_lob('david.sloan.doc', 'RESUME_FILES')  
timing stop
```

ORACLE

7 - 33

Copyright © 2008, Oracle. All rights reserved.

2

Writing Data to a LOB (continued)

1. The Microsoft Word files are stored in the D:\Labs\DATA_FILES\RESUMES directory.
2. To read them into the PICTURE column in the CUSTOMER_PROFILES table, the WRITE_LOB procedure is called and the name of the .doc files is passed as a parameter.

Note: This script is run in SQL*Plus, because TIMING is a SQL*Plus option and is not available in SQL Developer.

Writing Data to a LOB (continued)

The output is similar to the following:

```
timing start load_data
execute write_lob('karl.brimmer.doc', 'RESUME_FILES');
Begin inserting rows...
Row 1 inserted.
...
PL/SQL procedure successfully completed.

execute write_lob('monica.petera.doc', 'RESUME_FILES');
Begin inserting rows...
Row 1 inserted.
...
PL/SQL procedure successfully completed.

execute write_lob('david.sloan.doc', 'RESUME_FILES');
Begin inserting rows...
Row 1 inserted.
...
PL/SQL procedure successfully completed.

timing stop
timing for: load_data
Elapsed: 00:00:00.96
```

Reading LOBs from the Table

```
CREATE OR REPLACE PROCEDURE read_lob
IS
    v_lob_loc          BLOB;
    CURSOR profiles_cur IS
        SELECT id, full_name, resume, picture
        FROM customer_profiles;
    v_profiles_rec    customer_profiles%ROWTYPE;
BEGIN
    OPEN profiles_cur;
LOOP
    FETCH profiles_cur INTO v_profiles_rec;
    v_lob_loc := v_profiles_rec.picture;
    DBMS_OUTPUT.PUT_LINE('The length is: ' ||
                         DBMS_LOB.GETLENGTH(v_lob_loc));
    DBMS_OUTPUT.PUT_LINE('The ID is: ' || v_profiles_rec.id);
    DBMS_OUTPUT.PUT_LINE('The blob is read: ' ||
                         UTL_RAW.CAST_TO_VARCHAR2(DBMS_LOB.SUBSTR(v_lob_loc,200,1)));
    EXIT WHEN profiles_cur%NOTFOUND;
END LOOP;
CLOSE profiles_cur;
END;
```

ORACLE®

7 - 35

Copyright © 2008, Oracle. All rights reserved.

Reading LOBs from the Table

To retrieve the records that were inserted, you can call the READ_LOB procedure:

```
set serveroutput on
set verify on
set term on
set linesize 200

timing start read_data
execute read_lob;
timing stop
```

The commands shown in the slide read back the 90 records from the CUSTOMER_PROFILES table. For each record, the size of the LOB value plus the first 200 characters of the LOB are displayed on the screen. A SQL*Plus timer is started to capture the total elapsed time for the retrieval.

Updating LOB by Using DBMS_LOB in PL/SQL

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

ORACLE®

7 - 37

Copyright © 2008, Oracle. All rights reserved.

Updating LOB by Using DBMS_LOB in PL/SQL

In the example in the slide, the LOBLOC variable serves as the LOB locator, and the AMOUNT variable is set to the length of the text that you want to add. The SELECT FOR UPDATE statement locks the row and returns the LOB locator for the RESUME LOB column. Finally, the PL/SQL WRITE package procedure is called to write the text into the LOB value at the specified offset. WRITEAPPEND appends to the existing LOB value.

The example shows how to fetch a CLOB column in releases before Oracle9*i*. In those releases, it was not possible to fetch a CLOB column directly into a character column. The column value must be bound to a LOB locator, which is accessed by the DBMS_LOB package. An example later in this lesson shows that you can directly fetch a CLOB column by binding it to a character variable.

Checking the Space Usage of a LOB Table

```
CREATE OR REPLACE PROCEDURE check_space
IS
  l_fs1_bytes NUMBER;
  l_fs2_bytes NUMBER; ...
BEGIN
  DBMS_SPACE.SPACE_USAGE(
    segment_owner      => 'OE',
    segment_name       => 'CUSTOMER_PROFILES',
    segment_type       => 'TABLE',
    fs1_bytes          => l_fs1_bytes,
    fs1_blocks         => l_fs1_blocks,
    fs2_bytes          => l_fs2_bytes,
    fs2_blocks         => l_fs2_blocks, ...
  );
  DBMS_OUTPUT.ENABLE;
  DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = '||l_fs1_blocks||'
    Bytes = '||l_fs1_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = '||l_fs2_blocks||'
    Bytes = '||l_fs2_bytes); ...
  DBMS_OUTPUT.PUT_LINE('=====
  DBMS_OUTPUT.PUT_LINE('Total Blocks =
    '||to_char(l_fs1_blocks + l_fs2_blocks ...));
END;
/
```

ORACLE®

7 - 38

Copyright © 2008, Oracle. All rights reserved.

Checking the Space Usage of a LOB Table

To check the space usage in the disk blocks allocated to the LOB segment in the CUSTOMER_PROFILES table, use the CHECK_SPACE, as shown above. This procedure calls the DBMS_SPACE package.

To execute the procedure, run the following command:

```
EXECUTE check_space
```

The output is as follows:

```
FS1 Blocks = 1      Bytes = 8192
FS2 Blocks = 0      Bytes = 0
FS3 Blocks = 1      Bytes = 8192
FS4 Blocks = 3      Bytes = 24576
Full Blocks = 0     Bytes = 0
=====
Total Blocks =      5 ||
Total Bytes = 40960
PL/SQL procedure successfully completed.
```

Checking Space Usage of a LOB Table (continued)

Complete Code of the CHECK_SPACE Procedure

```
CREATE OR REPLACE PROCEDURE check_space
IS
    l_fs1_bytes NUMBER;      l_fs2_bytes NUMBER;
    l_fs3_bytes NUMBER;      l_fs4_bytes NUMBER;
    l_fs1_blocks NUMBER;     l_fs2_blocks NUMBER;
    l_fs3_blocks NUMBER;     l_fs4_blocks NUMBER;
    l_full_bytes NUMBER;     l_full_blocks NUMBER;
    l_unformatted_bytes NUMBER;
    l_unformatted_blocks NUMBER;
BEGIN
    DBMS_SPACE.SPACE_USAGE(
        segment_owner      => 'OE',
        segment_name        => 'CUSTOMER_PROFILES',
        segment_type        => 'TABLE',
        fs1_bytes           => l_fs1_bytes,
        fs1_blocks          => l_fs1_blocks,
        fs2_bytes           => l_fs2_bytes,
        fs2_blocks          => l_fs2_blocks,
        fs3_bytes           => l_fs3_bytes,
        fs3_blocks          => l_fs3_blocks,
        fs4_bytes           => l_fs4_bytes,
        fs4_blocks          => l_fs4_blocks,
        full_bytes          => l_full_bytes,
        full_blocks         => l_full_blocks,
        unformatted_blocks  => l_unformatted_blocks,
        unformatted_bytes   => l_unformatted_bytes
    );
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = '||l_fs1_blocks||'
                         Bytes = '||l_fs1_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = '||l_fs2_blocks||'
                         Bytes = '||l_fs2_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS3 Blocks = '||l_fs3_blocks||'
                         Bytes = '||l_fs3_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS4 Blocks = '||l_fs4_blocks||'
                         Bytes = '||l_fs4_bytes);
    DBMS_OUTPUT.PUT_LINE('Full Blocks = '||l_full_blocks||'
                         Bytes = '||l_full_bytes);
    DBMS_OUTPUT.PUT_LINE('=====
=====
=====');
    DBMS_OUTPUT.PUT_LINE('Total Blocks =
       '||to_char(l_fs1_blocks + l_fs2_blocks +
        l_fs3_blocks + l_fs4_blocks + l_full_blocks)||' ||
        Total Bytes = '|| to_char(l_fs1_bytes + l_fs2_bytes
        + l_fs3_bytes + l_fs4_bytes + l_full_bytes));
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*Plus:

ID	FULL_NAME	RESUME
164	Charlotte Kazan	Date of Birth: 8 February 1951 Resigned = 5 June 2000
150	Harry Dean Fonda	Date of Birth: 1 June 1956 Resigned = 30 September 2000

- Output in SQL Developer:

ID	FULL_NAME	RESUME
1	164 Charlotte Kazan	(CLOB) Resigned ...
2	150 Harry Dean Fonda	(CLOB) Date of Bi...

ORACLE®

Selecting CLOB Values by Using SQL

It is possible to see the data in a CLOB column by using a SELECT statement. It is not possible to see the data in a BLOB or BFILE column by using a SELECT statement in SQL*Plus. You must use a tool that can display the binary information for a BLOB, as well as the relevant software for a BFILE—for example, you can use Oracle Forms.

Selecting CLOB Values by Using DBMS_LOB

- DBMS_LOB.SUBSTR (lob, amount, start_pos)
- DBMS_LOB.INSTR (lob, pattern)

```
SELECT DBMS_LOB.SUBSTR (resume, 5, 18),
       DBMS_LOB.INSTR (resume, '=')
FROM   customer_profiles
WHERE  id IN (150, 164);
```

```
DBMS_LOB.SUBSTR(RESUME,5,18)
-----
-----
DBMS_LOB.INSTR(RESUME,'=')
-----
Febru          40
June           36
```

SQL*Plus

	DBMS_LOB.SUBSTR(RESUME,5,18)	DBMS_LOB.INSTR(RESUME,'=')
1	Febru	40
2	June	36

SQL Developer

Selecting CLOB Values by Using DBMS_LOB

DBMS_LOB.SUBSTR

Use DBMS_LOB.SUBSTR to display a part of a LOB. It is similar in functionality to the SUBSTR SQL function.

DBMS_LOB.INSTR

Use DBMS_LOB.INSTR to search for information within the LOB. This function returns the numerical position of the information.

Selecting CLOB Values in PL/SQL

```
SET LINESIZE 50 SERVEROUTPUT ON FORMAT WORD_WRAP

DECLARE
    text VARCHAR2(4001);
BEGIN
    SELECT resume INTO text
    FROM customer_profiles
    WHERE id = 150;
    DBMS_OUTPUT.PUT_LINE('text is: '|| text);
END;
/
```

```
anonymous block completed
text is: Date of Birth: 1 June 1956 Resigned = 30
September 2000
```

ORACLE®

7 - 42

Copyright © 2008, Oracle. All rights reserved.

Selecting CLOB Values in PL/SQL

The slide shows the code for accessing CLOB values that can be implicitly converted to VARCHAR2. When selected, the RESUME column value is implicitly converted from a CLOB to a VARCHAR2 to be stored in the TEXT variable.

Removing LOBS

- Delete a row containing LOBS:

```
DELETE
FROM customer_profiles
WHERE id = 164;
```

- Disassociate a LOB value from a row:

```
UPDATE customer_profiles
SET resume = EMPTY_CLOB()
WHERE id = 150;
```



Removing LOBS

A LOB instance can be deleted (destroyed) by using the appropriate SQL DML statements. The SQL statement `DELETE` deletes a row and its associated internal LOB value. To preserve the row and destroy only the reference to the LOB, you must update the row by replacing the LOB column value with `NULL` or an empty string, or by using the `EMPTY_B/CLOB()` function.

Note: Replacing a column value with `NULL` and using `EMPTY_B/CLOB` are not the same. Using `NULL` sets the value to null; using `EMPTY_B/CLOB` ensures that nothing is in the column.

A LOB is destroyed when the row containing the LOB column is deleted, when the table is dropped or truncated, or when all LOB data is updated.

You must explicitly remove the file associated with a `BFILE` by using the OS commands.

To erase part of an internal LOB, you can use `DBMS_LOB.ERASE`.

Lesson Agenda

- Introduction to LOBS
- Managing BFILEs by using the DBMS_LOB package
- Migrating LONG data types to LOBS
- Manipulating LOB data
- Using temporary LOBS

ORACLE®

7 - 44

Copyright © 2008, Oracle. All rights reserved.

Temporary LOBS

- Temporary LOBS:
 - Provide an interface to support creation of LOBs that act like local variables
 - Can be BLOBS, CLOBs, or NCLOBs
 - Are not associated with a specific table
 - Are created by using the DBMS_LOB.CREATETEMPORARY procedure
 - Use DBMS_LOB routines
- The lifetime of a temporary LOB is a session.
- Temporary LOBS are useful for transforming data in permanent internal LOBS.

ORACLE®

7 - 45

Copyright © 2008, Oracle. All rights reserved.

Temporary LOBS

Temporary LOBs provide an interface to support the creation and deletion of LOBs that act like local variables. Temporary LOBs can be BLOBS, CLOBs, or NCLOBs.

The following are the features of temporary LOBs:

- Data is stored in your temporary tablespace, not in tables.
- Temporary LOBs are faster than persistent LOBs, because they do not generate redo or rollback information.
- Temporary LOBs lookup is localized to each user's own session. Only the user who creates a temporary LOB can access it, and all temporary LOBs are deleted at the end of the session in which they were created.
- You can create a temporary LOB by using DBMS_LOB.CREATETEMPORARY.

Temporary LOBs are useful when you want to perform a transformational operation on a LOB (for example, changing an image type from GIF to JPEG). A temporary LOB is empty when created and does not support the EMPTY_B/CLOB functions.

Use the DBMS_LOB package to use and manipulate temporary LOBs.

Creating a Temporary LOB

The PL/SQL procedure to create and test a temporary LOB:

```
CREATE OR REPLACE PROCEDURE is_templob_open(
    p_lob IN OUT BLOB, p_retval OUT INTEGER) IS
BEGIN
    -- create a temporary LOB
    DBMS_LOB.CREATETEMPORARY (p_lob, TRUE);
    -- see if the LOB is open: returns 1 if open
    p_retval := DBMS_LOB.ISOPEN (p_lob);
    DBMS_OUTPUT.PUT_LINE (
        'The file returned a value...' || p_retval);
    -- free the temporary LOB
    DBMS_LOB.FREETEMPORARY (p_lob);
END;
/
```

ORACLE®

7 - 46

Copyright © 2008, Oracle. All rights reserved.

Creating a Temporary LOB

The example in the slide shows a user-defined PL/SQL procedure, `is_templob_open`, which creates a temporary LOB. This procedure accepts a LOB locator as input, creates a temporary LOB, opens it, and tests whether the LOB is open.

The `is_templob_open` procedure uses the procedures and functions from the `DBMS_LOB` package as follows:

- The `CREATETEMPORARY` procedure is used to create the temporary LOB.
- The `ISOPEN` function is used to test whether a LOB is open: This function returns the value 1 if the LOB is open.
- The `FREETEMPORARY` procedure is used to free the temporary LOB. Memory increases incrementally as the number of temporary LOBs grows, and you can reuse the temporary LOB space in your session by explicitly freeing temporary LOBs.

Summary

In this lesson, you should have learned how to:

- Identify four built-in types for large objects: BLOB, CLOB, NCLOB, and BFILE
- Describe how LOBs replace LONG and LONG RAW
- Describe two storage options for LOBs:
 - Oracle server (internal LOBs)
 - External host files (external LOBs)
- Use the DBMS_LOB PL/SQL package to provide routines for LOB management
- Use temporary LOBs in a session

ORACLE®

7 - 47

Copyright © 2008, Oracle. All rights reserved.

Summary

There are four LOB data types:

- A BLOB is a binary large object.
- A CLOB is a character large object.
- An NCLOB stores multiple-byte national character set data.
- A BFILE is a large object stored in a binary file outside the database.

LOBs can be stored internally (in the database) or externally (in an OS file). You can manage LOBs by using the DBMS_LOB package and its procedures.

Temporary LOBs provide an interface to support the creation and deletion of LOBs that act like local variables.

Practice 7: Overview

This practice covers the following topics:

- Creating object types by using 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



Practice 7: Overview

In this practice, you create a table with both BLOB and CLOB columns. Then, you use the DBMS_LOB package to populate the table and manipulate the data.

Use the OE schema for this practice.

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”

Practice 7

In this practice, you create a table with both BLOB and CLOB columns. Then, you use the DBMS_LOB package to populate the table and manipulate the data.

Working with LOBs

1. Create a table called PERSONNEL by executing the

D:\Labs\labs\lab_07_01.sql script file. The table contains the following attributes and data types:

Column Name	Data Type	Length
ID	NUMBER	6
last_name	VARCHAR2	35
review	CLOB	N/A
picture	BLOB	N/A

2. Insert two rows into the PERSONNEL table, one each for employee 2034 (whose last name is Allen) and employee 2035 (whose last name is Bond). Use the empty function for the CLOB, and provide NULL as the value for the BLOB.
3. Examine and execute the D:\labs\labs\lab_07_03.sql script. The script creates a table named REVIEW_TABLE. This table contains the annual review information for each employee. The script also contains two statements to insert review details about two employees.
4. Update the PERSONNEL table.

- a. Populate the CLOB for the first row by using this subquery in an UPDATE statement:

```
SELECT ann_review
      FROM review_table
     WHERE employee_id = 2034;
```

- b. Populate the CLOB for the second row by using PL/SQL and the DBMS_LOB package. Use the following SELECT statement to provide a value for the LOB locator.

```
SELECT ann_review
      FROM review_table
     WHERE employee_id = 2035;
```

5. Create a procedure that adds a locator to a binary file into the PICTURE column of the PRODUCT_INFORMATION table. The binary file is a picture of the product. The image files are named after the product IDs. You must load an image file locator into all rows in the Printers category (CATEGORY_ID = 12) in the PRODUCT_INFORMATION table.

- a. Create a DIRECTORY object called PRODUCT_PIC that references the location of the binary. These files are available in the

D:\Labs\DATA_FILES\PRODUCT_PIC folder.

```
CREATE DIRECTORY product_pic AS
'D:\Labs\DATA_FILES\PRODUCT_PIC';
```

(Alternatively, use the D:\labs\labs\lab_07_05a.sql script.)

- b. Add the image column to the PRODUCT_INFORMATION table by using:

```
ALTER TABLE product_information ADD (picture BFILE);
```

(Alternatively, use the D:\labs\labs\lab_07_05_b.sql file.)

Practice 7 (continued)

- c. Create a PL/SQL procedure called `load_product_image` that uses `DBMS_LOB.FILEEXISTS` to test whether the product picture file exists. If the file exists, set the BFILE locator for the file in the `PICTURE` column; otherwise, display a message that the file does not exist. Use the `DBMS_OUTPUT` package to report file size information about each image associated with the `PICTURE` column.
(Alternatively, use the `D:\labs\labs\lab_07_05_c.sql` file.)
- d. Invoke the procedure by passing the name of the `PRODUCT_PIC` directory object as a string literal parameter value.
- e. Check the LOB space usage of the `PRODUCT_INFORMATION` table. Use the `D:\labs\labs\lab_07_05_e.sql` file to create the procedure and execute it.

8

Administering SecureFile LOBS

ORACLE®

Copyright © 2008, Oracle. All rights reserved.

Objectives

After completing this lesson, you should be able to:

- Describe SecureFile LOB features
- Enable SecureFile LOB deduplication, compression, and encryption
- Migrate BasicFile LOBs to the SecureFile LOB format
- Analyze the performance of LOBs



Objectives

In this lesson, you learn to migrate the pre-Oracle Database 11g LOB storage format (called BasicFile LOB format) to the SecureFile LOB format. You also compare the performance of LOBs stored in the BasicFile format with the SecureFile format. Finally, you learn how to enable SecureFile LOB deduplication (storage sharing), compression, and encryption.

Lesson Agenda

- SecureFile LOB features
- Deduplication, compression, and encryption
- Migration of BasicFile LOBs to the SecureFile LOB format
- Performance of LOBS

ORACLE®

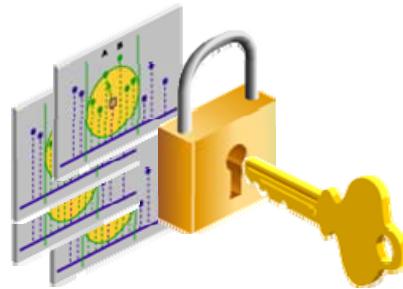
8 - 3

Copyright © 2008, Oracle. All rights reserved.

SecureFile LOBS

Oracle Database 11g offers a reengineered large object (LOB) data type that:

- Improves performance
- Eases manageability
- Simplifies application development
- Offers advanced, next-generation functionality such as intelligent compression and transparent encryption



ORACLE®

8 - 4

Copyright © 2008, Oracle. All rights reserved.

SecureFile LOBS

With SecureFile LOBs, the LOB data type is completely reengineered with dramatically improved performance, manageability, and ease of application development. This implementation, available with Oracle Database 11g, also offers advanced, next-generation functionality such as intelligent compression and transparent encryption. This feature significantly strengthens the native content management capabilities of Oracle Database.

SecureFile LOBs were introduced to supplement the implementation of original BasicFile LOBs that are identified by the BASICFILE SQL parameter.

Storage of SecureFile LOBS

Oracle Database 11g implements a new storage paradigm 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.



Storage of SecureFile LOBS

Starting with Oracle Database 11g, you have the option of using the new SecureFile storage paradigm for LOBs. You can specify to use the new paradigm by using the SECUREFILE keyword in the CREATE TABLE statement. If that keyword is left out, the old storage paradigm for basic file LOBs is used. This is the default behavior.

You can modify the `init.ora` file and change the default behavior for the storage of LOBs by setting the DB_SECUREFILE initialization parameter. The values allowed are:

- ALWAYS : Attempts to create all LOB files as SECUREFILES but creates any LOBs not in ASSM tablespaces as BASICFILE LOBs
- FORCE : All LOBs created in the system are created as SECUREFILE LOBs.
- PERMITTED : The default; allows SECUREFILES to be created when specified with the SECUREFILE keyword in the CREATE TABLE statement
- NEVER : Creates LOBs that are specified as SECUREFILE LOBs as BASICFILE LOBs
- IGNORE : Ignores the SECUREFILE keyword and all SECUREFILE options

Creating a SecureFile LOB

- Create a tablespace for the LOB data:

```
-- have your dba do this:  
CREATE TABLESPACE sf_tbs1  
  DATAFILE 'sf_tbs1.dbf' SIZE 1500M REUSE  
  AUTOEXTEND ON NEXT 200M  
  MAXSIZE 3000M  
  SEGMENT SPACE MANAGEMENT AUTO;
```

1

- Create a table to hold the LOB data:

```
CONNECT oe  
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

ORACLE

8 - 6

Copyright © 2008, Oracle. All rights reserved.

Creating a SecureFile LOB

To create a column to hold a LOB that is a SecureFile, you:

- Create a tablespace to hold the data
- Define a table that contains a LOB column data type that is used to store the data in the SecureFile format

In the example shown in the slide:

1. The sf_tbs1 tablespace is defined. This tablespace stores the LOB data in the SecureFile format. When you define a column to hold SecureFile data, you must have Automatic Segment Space Management (ASSM) enabled for the tablespace in order to support SecureFiles.
2. The CUSTOMER_PROFILE_SF table is created. The PROFILE_INFO column holds the LOB data in the SecureFile format, because the storage clause identifies the format.

Writing Data to the SecureFile LOB

Writing data to a SECUREFILE LOB works in the same way as writing data to a BASICFILE LOB.

- Create the DIRECTORY object in the database that points to the location where the external documents are stored.
- Create a procedure to read the LOB data into the LOB column.
- Create a procedure to insert LOB data into the table (which references the procedure that reads the LOB data).
- Execute the insert procedure and specify the file that you want to insert.

ORACLE®

8 - 7

Copyright © 2008, Oracle. All rights reserved.

Writing Data to the SecureFile LOB

In the previous lesson, you learned how to write data to a LOB. The same procedure is used when writing data to a SecureFile LOB.

Reading Data from the Table

Reading data from a SECUREFILE LOB works in the same way as reading data from a BASICFILE LOB.

- Create a procedure to specify the LOB data that you want to read from the table.
- Execute the procedure to read the table.



Reading Data From a SecureFile LOB

In the previous lesson, you learned how to read data from a LOB. The same procedure is used when reading data from a SecureFile LOB.

Lesson Agenda

- SecureFile LOB features
- Deduplication, compression, and encryption
- Migration of BasicFile LOBs to the SecureFile LOB format
- Performance of LOBS

ORACLE®

8 - 9

Copyright © 2008, Oracle. All rights reserved.

Enabling Deduplication and Compression

To enable deduplication and compression, use the ALTER TABLE statement with the DEDUPLICATE and COMPRESS options.

- By enabling deduplication with SecureFiles, duplicate LOB data is automatically detected and space is conserved by storing only one copy.
- Enabling compression turns on LOB compression.

```
ALTER TABLE tablename  
MODIFY LOB lobcolname  
(DEDUPLICATE option  
COMPRESS option)
```



ORACLE®

8 - 10

Copyright © 2008, Oracle. All rights reserved.

Enabling Deduplication and Compression with the ALTER TABLE Statement

You can enable deduplication and compression of SecureFiles with the ALTER TABLE statement and the DEDUPLICATE and COMPRESS options.

The DEDUPLICATE option enables you to specify that LOB data, which is identical in two or more rows in a LOB column, should share the same data blocks. The opposite of this option is KEEP_DUPLICATES. Using a secure hash index to detect duplication, the database combines LOBs with identical content into a single copy, thereby reducing storage and simplifying storage management. You can also use DBMS_LOB.SETOPTIONS to enable or disable deduplication on individual LOBs.

The options for the COMPRESS clause are:

- COMPRESS HIGH: Provides the best compression but incurs the most work
- COMPRESS MEDIUM: Is the default value
- NOCOMPRESS: Disables compression

You can also use DBMS_LOB.SETOPTIONS to enable or disable compression on individual LOBs.

Enabling Deduplication and Compression: Example

- 1.** Check the space being used by the CUSTOMER_PROFILE_SF table.
- 2.** Enable deduplication and compression on the PROFILE_INFO LOB column with the ALTER TABLE statement.
- 3.** Recheck the space being used by the CUSTOMER_PROFILE_SF table.
- 4.** Reclaim the space.

ORACLE®

8 - 11

Copyright © 2008, Oracle. All rights reserved.

Deduplication and Compression: Example

To demonstrate how efficient deduplication and compression are on SecureFiles, the example follows the set of steps outlined in the slide.

In the first step, you see the space being used by the CUSTOMER_PROFILE_SF table.

In the second step, you enable deduplication and compression for the PROFILE_INFO LOB column in the CUSTOMER_PROFILE_SF table.

In the third step, you examine the space being used after deduplication and compression are enabled.

In the fourth step, you reclaim the space and examine the results.

Step 1: Checking Space Usage

```
CREATE OR REPLACE PROCEDURE check_sf_space
IS
  l_fs1_bytes NUMBER;
  l_fs2_bytes NUMBER;
  ...
BEGIN
  DBMS_SPACE.SPACE_USAGE(
    segment_owner      => 'OE',
    segment_name       => 'CUSTOMER_PROFILES_SF',
    segment_type       => 'TABLE',
    fs1_bytes          => l_fs1_bytes,
    fs1_blocks         => l_fs1_blocks,
    fs2_bytes          => l_fs2_bytes,
    fs2_blocks         => l_fs2_blocks, ...
  );
  DBMS_OUTPUT.ENABLE;
  DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = '||l_fs1_blocks||'
    Bytes = '||l_fs1_bytes);
  DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = '||l_fs2_blocks||'
    Bytes = '||l_fs2_bytes); ...
  DBMS_OUTPUT.PUT_LINE('=====');
  DBMS_OUTPUT.PUT_LINE('Total Blocks =
    '||to_char(l_fs1_blocks + l_fs2_blocks));
  ...
END;
```

ORACLE®

8 - 12

Copyright © 2008, Oracle. All rights reserved.

Checking Space Usage Before Deduplication and Compression

In the previous lesson, you checked the space usage of a BASICFILE LOB. Here, you create another procedure to check the SECUREFILE LOB space usage.

To execute the procedure, run the following command:

```
EXECUTE check_sf_space
```

Note: The full code for the CHECK_SF_SPACE procedure is shown on the next page.

Checking Space Usage Before Deduplication and Compression (continued)

```
CREATE OR REPLACE PROCEDURE check_sf_space
IS
    l_fs1_bytes NUMBER;      l_fs2_bytes NUMBER;
    l_fs3_bytes NUMBER;      l_fs4_bytes NUMBER;
    l_fs1_blocks NUMBER;     l_fs2_blocks NUMBER;
    l_fs3_blocks NUMBER;     l_fs4_blocks NUMBER;
    l_full_bytes NUMBER;    l_full_blocks NUMBER;
    l_unformatted_bytes NUMBER;
    l_unformatted_blocks NUMBER;
BEGIN
    DBMS_SPACE.SPACE_USAGE(
        segment_owner      => 'OE',
        segment_name        => 'CUSTOMER_PROFILE_SF',
        segment_type        => 'TABLE',
        fs1_bytes           => l_fs1_bytes,
        fs1_blocks          => l_fs1_blocks,
        fs2_bytes           => l_fs2_bytes,
        fs2_blocks          => l_fs2_blocks,
        fs3_bytes           => l_fs3_bytes,
        fs3_blocks          => l_fs3_blocks,
        fs4_bytes           => l_fs4_bytes,
        fs4_blocks          => l_fs4_blocks,
        full_bytes          => l_full_bytes,
        full_blocks         => l_full_blocks,
        unformatted_blocks  => l_unformatted_blocks,
        unformatted_bytes   => l_unformatted_bytes
    );
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = '||l_fs1_blocks||'
                         Bytes = '||l_fs1_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = '||l_fs2_blocks||'
                         Bytes = '||l_fs2_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS3 Blocks = '||l_fs3_blocks||'
                         Bytes = '||l_fs3_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS4 Blocks = '||l_fs4_blocks||'
                         Bytes = '||l_fs4_bytes);
    DBMS_OUTPUT.PUT_LINE('Full Blocks = '||l_full_blocks||'
                         Bytes = '||l_full_bytes);
    DBMS_OUTPUT.PUT_LINE('=====');
    DBMS_OUTPUT.PUT_LINE('Total Blocks =
                        '||to_char(l_fs1_blocks + l_fs2_blocks +
                         l_fs3_blocks + l_fs4_blocks + l_full_blocks)||' ||
                         Total Bytes = '|| to_char(l_fs1_bytes + l_fs2_bytes
                         + l_fs3_bytes + l_fs4_bytes + l_full_bytes));
END check_sf_space;
```

Step 1: Checking Space Usage

Execution Results:

```
EXECUTE check_sf_space

FS1 Blocks = 0      Bytes = 0
FS2 Blocks = 1      Bytes = 8192
FS3 Blocks = 0      Bytes = 0
FS4 Blocks = 4      Bytes = 32768
Full Blocks = 0     Bytes = 0
=====
Total Blocks = 5 ||
Total Bytes = 40960

PL/SQL procedure successfully completed.
```

ORACLE®

8 - 14

Copyright © 2008, Oracle. All rights reserved.

Checking Space Usage Before Deduplication and Compression (continued)

You are shown the space usage before enabling deduplication and compression. The amount shown in the slide is used as a baseline for comparison over the next few steps.

Note: You can also compare the space usage with that of the BASICFILE LOB from the previous lesson.

Enabling Deduplication and Compression: Example

Step 2: Enabling deduplication and compression:

```
ALTER TABLE customer_profiles_sf
MODIFY LOB (profile_info)
(DEDUPLICATE LOB
COMPRESS      HIGH);
```

```
Table altered.
```

ORACLE®

8 - 15

Copyright © 2008, Oracle. All rights reserved.

Enabling Deduplication and Compression: Example

To enable deduplication and compression, run the ALTER TABLE statement with the appropriate options.

In this example, deduplication is turned on and the compression rate is set to HIGH.

Enabling Deduplication and Compression: Example

Step 3: Rechecking LOB space usage:

```
EXECUTE check_sf_space

FS1 Blocks = 0      Bytes = 0
FS2 Blocks = 0      Bytes = 0
FS3 Blocks = 0      Bytes = 0
FS4 Blocks = 4      Bytes = 32768
Full Blocks = 1     Bytes = 8192
=====
Total Blocks = 5 ||
Total Bytes = 40960

PL/SQL procedure successfully completed.
```

ORACLE®

8 - 16

Copyright © 2008, Oracle. All rights reserved.

Rechecking LOB Space Usage

The amount of space used should be about 65% less than before deduplication and compression were enabled.

If the total space used appears to be the same as before deduplication and compression were enabled, you need to reclaim the free space before it is usable again.

Enabling Deduplication and Compression: Example

Step 4: Reclaiming the free space:

```
ALTER TABLE customer_profiles_sf ENABLE ROW MOVEMENT; 1  
Table altered.  
  
ALTER TABLE customer_profiles_sf SHRINK SPACE COMPACT 2  
Table altered.  
  
ALTER TABLE customer_profiles_sf SHRINK SPACE; 3  
Table altered.
```

ORACLE®

8 - 17

Copyright © 2008, Oracle. All rights reserved.

Reclaiming the Free Space

1. The first statement enables row movement so that the data can be shifted to save space. Compacting the segment requires row movement.
2. The second statement (ALTER TABLE resumes SHRINK SPACE COMPACT) redistributes the rows inside the blocks resulting in more free blocks under the High Water Mark (HWM)—but the HWM itself is not disturbed.
3. The third statement (ALTER TABLE resumes SHRINK SPACE) returns unused blocks to the database and resets the HWM, moving it to a lower position. Lowering the HWM should result in better full-table scan reads.

Rechecking LOB Space Usage

```
EXECUTE check_sf_space  
FS1 Blocks = 0      Bytes = 0  
FS2 Blocks = 1      Bytes = 8192  
FS3 Blocks = 0      Bytes = 0  
FS4 Blocks = 0      Bytes = 0  
Full Blocks = 0     Bytes = 0  
=====  
Total Blocks = 1 |||  
Total Bytes = 8192
```

Using Encryption

The encryption option enables you to turn the LOB encryption on or off, and optionally select an encryption algorithm.

- Encryption is performed at the block level.
- You can specify the encryption algorithm:
 - 3DES168
 - AES128
 - AES192 (default)
 - AES256
- The column encryption key is derived by using Transparent Data Encryption.
- All LOBs in the LOB column are encrypted.
- DECRYPT keeps the LOBs in cleartext.
- LOBs can be encrypted on a per-column or per-partition basis.



ORACLE®

8 - 18

Copyright © 2008, Oracle. All rights reserved.

Using Encryption

You can create a table or alter a table with encryption enabled or disabled on a LOB column. The current Transparent Data Encryption (TDE) syntax is used for extending encryption to LOB data types.

Using Encryption

1. Create a directory to store the Transparent Data Encryption (TDE) wallet.

```
mkdir d:\etc\oracle\wallets
```

2. Edit the <ORACLE_HOME>\network\admin\sqlnet.ora file to indicate the location of the TDE wallet.

```
ENCRYPTION_WALLET_LOCATION=(SOURCE=(METHOD=FILE)
(METHOD_DATA= (DIRECTORY=d:\etc\oracle\wallets)))
```

3. Stop and start the listener for the change to take effect.

```
LSNRCTL RELOAD
```

4. To open the wallet, log in to SQL*Plus as SYSDBA and execute the following command:

```
ALTER system SET KEY IDENTIFIED BY "welcome1";
```

ORACLE

8 - 19

Copyright © 2008, Oracle. All rights reserved.

Using Encryption (continued)

TDE enables you to encrypt sensitive data in database columns as it is stored in the operating system files.

TDE is a key-based access control system that enforces authorization by encrypting data with a key that is kept secret. There can be only one key for each database table that contains encrypted columns, regardless of the number of encrypted columns in a given table. Each table's column encryption key is, in turn, encrypted with the database server's master key. No keys are stored in the database. Instead, they are stored in an Oracle wallet, which is part of the external security module.

To enable TDE, perform the following:

1. Create a directory to store the TDE wallet.
2. Modify the sqlnet.ora file to identify the location of the TDE wallet, as shown in the slide. Make sure that the wallet location is set to a location outside the Oracle installation to avoid ending up on a backup tape together with encrypted data.
3. Stop and start the listener to have the change take effect: LSNRCTL RELOAD
4. Open the wallet. Log in to SQL*Plus as the SYS user and execute the following command:

```
ALTER system SET KEY IDENTIFIED BY "welcome";
```

Using Encryption: Example

- Enable encryption:

```
ALTER TABLE customer_profiles_sf  
MODIFY (profile_info ENCRYPT USING 'AES192');
```

Table altered.

- Verify that the LOB is encrypted:

```
SELECT *  
FROM user_encrypted_columns;
```

TABLE_NAME	COLUMN_NAME	ENCRYPTION_ALG	SAL
CUSTOMER_PROFILES	PROFILE_INFO	AES 192 bits key	YES

ORACLE®

8 - 20

Copyright © 2008, Oracle. All rights reserved.

Using Encryption: Example

In the example shown in the slide, the CUSTOMER_PROFILES_SF table is modified so that the PROFILE_INFO column uses encryption.

You can query the USER_ENCRYPTED_COLUMNS dictionary view to see the status of the encrypted columns.

Lesson Agenda

- SecureFile LOB features
- Deduplication, compression, and encryption
- Migration of BasicFile LOBs to the SecureFile LOB format
- Performance of LOBS

ORACLE®

8 - 21

Copyright © 2008, Oracle. All rights reserved.

Migrating from BasicFile to SecureFile Format

Check the LOB segment subtype name for the BasicFile format:

```
col segment_name format a30
col segment_type format a13

SELECT segment_name, segment_type, segment_subtype
FROM   dba_segments
WHERE  tablespace_name = 'LOB_TBS1'
AND    segment_type = 'LOBSEGMENT';

SEGMENT_NAME          SEGMENT_TYPE      SEGME
-----
SYS_LOB0000080068C00004$$      LOBSEGMENT      ASSM
```

ORACLE®

8 - 22

Copyright © 2008, Oracle. All rights reserved.

LOB Segment Type for BasicFile Format

By querying the DBA_SEGMENTS view, you can see that the LOB segment subtype name for BasicFile LOB storage is ASSM.

Migrating from BasicFile to SecureFile Format

- The migration from BasicFile to SecureFiles LOB storage format is performed online.
- This means that the CUSTOMER_PROFILES table continues to be accessible during the migration.
- This type of operation is called online redefinition.

```
connect oe
CREATE TABLE customer_profiles_interim
(id NUMBER,
 full_name VARCHAR2 (45),
 resume CLOB,
 picture BLOB)
LOB(picture) STORE AS SECUREFILE
(TABLESPACE lob_tbs1);
```

ORACLE®

8 - 23

Copyright © 2008, Oracle. All rights reserved.

Creating an Interim Table

Online redefinition requires an interim table for data storage.

In this step, the interim table is defined with the SecureFiles LOB storage format, and the LOB is stored in the lob_tbs1 tablespace. After the migration is completed, the PICTURE LOB is stored in the lob_tbs1 tablespace.

Migrating from BasicFile to SecureFile Format

Call the DBMS_REDEFINITION package to perform the online redefinition operation:

```
connect /as sysdba
DECLARE
  error_count PLS_INTEGER := 0;
BEGIN
  DBMS_REDEFINITION.START_REDEF_TABLE
    ('OE', 'customer_profiles', 'customer_profiles_interim',
     'id id, full_name full_name,
      resume resume, picture picture',
     OPTIONS_FLAG => DBMS_REDEFINITION.CONS_USE_ROWID);
  DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS
    ('OE', 'customer_profiles', 'customer_profiles_interim',
     1, true,true,true,false, error_count);
  DBMS_OUTPUT.PUT_LINE('Errors := ' || TO_CHAR(error_count));
  DBMS_REDEFINITION.FINISH_REDEF_TABLE
    ('OE', 'customer_profiles', 'customer_profiles_interim');
END;
```

ORACLE®

8 - 24

Copyright © 2008, Oracle. All rights reserved.

Using DBMS_REDEFINITION to Perform Redefinition

After running the code shown in the slide and completing the redefinition operation, you can drop the interim table:

```
connect oe

DROP TABLE customer_profiles_interim;
```

Now, check the segment type of the migrated LOB. Note that the segment subtype for SecureFile LOB storage is SECUREFILE; for BasicFile format, it is ASSM.

```
SELECT segment_name, segment_type, segment_subtype
FROM dba_segments
WHERE tablespace_name = 'LOB_TBS1'
AND segment_type = 'LOBSEGMENT'
/
```

SEGMENT_NAME	SEGMENT_TYPE	SEGMENT_SU
SYS_LOB0000080071C00004\$\$	LOBSEGMENT	SECUREFILE

Lesson Agenda

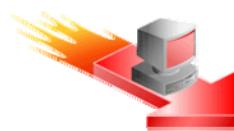
- SecureFile LOB features
- Deduplication, compression, and encryption
- Migration of BasicFile LOBs to the SecureFile LOB format
- Performance of LOBs



Comparing Performance

Compare the performance on loading and reading LOB columns in the SecureFile and BasicFile formats:

Performance Comparison	Loading Data	Reading Data
SecureFile format	00:00:00.96	00:00:01.09
BasicFile format	00:00:01.68	00:00:01.15



ORACLE®

8 - 26

Copyright © 2008, Oracle. All rights reserved.

Performance

In the examples shown in this lesson and the previous lesson, the performance on loading and reading data in the LOB column of the SecureFile format LOB is faster than that of the BasicFile format LOB.

Summary

In this lesson, you should have learned how to:

- Describe SecureFile LOB features
- Enable SecureFile LOB deduplication, compression, and encryption
- Migrate BasicFile LOBs to the SecureFile LOB format
- Analyze the performance of LOBs



Summary

In this lesson, you learned about the new SecureFile format for LOBs. You learned that the SecureFile format offers features such as deduplication, compression, and encryption. You learned how to migrate the older version BasicFile format to the SecureFile format, and also learned that the performance of SecureFile format LOBs is faster than the BasicFile format LOBs.

Practice 8 Overview: Using SecureFile Format LOBS

This practice covers the following topics:

- Setting up the environment for LOBs
- Migrating BasicFile LOBs to SecureFile LOBs
- Enabling deduplication and compression



Practice 8 Overview: Using SecureFile Format LOBS

In this lesson, you practice using the features of SecureFile format LOBs.

Use the OE schema for this practice.

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”

Practice 8

In this lesson, you practice using the features of SecureFile format LOBs.

Working with SecureFile LOBs

1. In this practice, you migrate a BasicFile format LOB to a SecureFile format LOB. You need to set up several supporting structures:

- a. As the OE user, drop your existing PRODUCT_DESCRIPTIONS table and create a new one:

```
DROP TABLE product_descriptions;

CREATE TABLE product_descriptions
  (product_id NUMBER);
```

- b. As the SYS user, create a new tablespace to store the LOB information.

```
CREATE TABLESPACE lob_tbs2
  DATAFILE 'lob_tbs2.dbf' SIZE 1500M REUSE
  AUTOEXTEND ON NEXT 200M
  MAXSIZE 3000M
  SEGMENT SPACE MANAGEMENT AUTO;
```

- c. Create a directory object that identifies the location of your LOBs. In the Oracle classroom, the location is in the Oracle D:\labs\DATA_FILES\PRODUCT_PIC folder. Then, grant read privileges on the directory to the OE user.

```
CREATE OR REPLACE DIRECTORY product_files
  AS 'd:\Labs\DATA_FILES\PRODUCT_PIC';

GRANT READ ON DIRECTORY product_files TO oe;
```

- d. As the OE user, alter the table and add a BLOB column of the BASICFILE storage type.

```
ALTER TABLE product_descriptions ADD
  (detailed_product_info BLOB )
  LOB (detailed_product_info) STORE AS BASICFILE
  (tablespace lob_tbs2);
```

- e. Create the procedure to load the LOB data into the column (You can run the D:\Labs\labs\lab_08_01_e.sql script.):

```
CREATE OR REPLACE PROCEDURE loadLOBFromBFILE_proc (
  p_dest_loc IN OUT BLOB, p_file_name IN VARCHAR2)
IS
  v_src_loc  BFILE := BFILENAME('PRODUCT_FILES', p_file_name);
  v_amount    INTEGER := 4000;
BEGIN
  DBMS_LOB.OPEN(v_src_loc, DBMS_LOB.LOB_READONLY);
  v_amount := DBMS_LOB.GETLENGTH(v_src_loc);
  DBMS_LOBLOADFROMFILE(p_dest_loc, v_src_loc, v_amount);
  DBMS_LOB.CLOSE(v_src_loc);
END loadLOBFromBFILE_proc;
/
```

Practice 8 (continued)

- f. As the OE user, create the procedure to write the LOB data. (You can run the D:\Labs\lab\lab_08_01_f.sql script.)

```
CREATE OR REPLACE PROCEDURE write_lob (p_file IN
    VARCHAR2)
IS
    i      NUMBER;    v_id NUMBER;    v_b    BLOB;
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE('Begin inserting rows... ');
    FOR i IN 1 .. 5 LOOP
        v_id:=SUBSTR(p_file, 1, 4);
        INSERT INTO product_descriptions
            VALUES (v_id, EMPTY_BLOB())
            RETURNING detailed_product_info INTO v_b;
        loadLOBFromBFILE_proc(v_b,p_file);
        DBMS_OUTPUT.PUT_LINE('Row ' || i || ' inserted.');
    END LOOP;
    COMMIT;
END write_lob;
/
```

- g. As the OE user, execute the procedures to load the data. If you are using SQL*Plus, you can set the timing on to observe the time. (You can run the D:\Labs\lab\lab_08_01_g.sql script.)

Note: If you are using SQL Developer, issue only the EXECUTE statements listed as follows. In SQL Developer, some of the SQL*Plus commands are ignored. It is recommended that you use SQL*Plus for this exercise.

```
set serveroutput on
set verify on
set term on
set lines 200

timing start load_data
execute write_lob('1726_LCD.doc');
execute write_lob('1734_RS232.doc');
execute write_lob('1739_SDRAM.doc');
timing stop
```

- h. As the SYSTEM user, check the segment type in the data dictionary.

```
SELECT segment_name, segment_type, segment_subtype
  FROM dba_segments
 WHERE tablespace_name = 'LOB_TBS2'
   AND segment_type = 'LOBSEGMENT';
```

Practice 8 (continued)

- i. As the OE user, create an interim table.

```
CREATE TABLE product_descriptions_interim
(product_id NUMBER,
detailed_product_info BLOB)
LOB(detailed_product_info) STORE AS SECUREFILE
(TABLESPACE lob_tbs2);
```

- j. Connect as the SYSTEM user and run the redefinition script. (You can run the D:\Labs\lab\lab_08_01_j.sql script.)

```
DECLARE
error_count PLS_INTEGER := 0;
BEGIN
DBMS_REDEFINITION.START_REDEF_TABLE
('OE', 'product_descriptions',
'product_descriptions_interim',
'product_id product_id, detailed_product_info
detailed_product_info',
OPTIONS_FLAG => DBMS_REDEFINITION.CONS_USE_ROWID);
DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS
('OE', 'product_descriptions',
'product_descriptions_interim',
1, true,true,true,false, error_count);
DBMS_OUTPUT.PUT_LINE('Errors := ' ||
TO_CHAR(error_count));
DBMS_REDEFINITION.FINISH_REDEF_TABLE
('OE', 'product_descriptions',
'product_descriptions_interim');
END;
/
```

- k. As the OE user, remove the interim table.

```
DROP TABLE product_descriptions_interim;
```

- l. As the SYSTEM user, check the segment type in the data dictionary to make sure it is now set to SECUREFILE.

```
SELECT segment_name, segment_type, segment_subtype
FROM dba_segments
WHERE tablespace_name = 'LOB_TBS2'
AND segment_type = 'LOBSEGMENT';
```

Practice 8 (continued)

- m. As the OE user, check the space of the table by executing the CHECK_SPACE procedure. (You can run the D:\Labs\labs\lab_08_01_m.sql script.)

```
CREATE OR REPLACE PROCEDURE check_space
IS
    l_fs1_bytes NUMBER;
    l_fs2_bytes NUMBER;
    l_fs3_bytes NUMBER;
    l_fs4_bytes NUMBER;
    l_fs1_blocks NUMBER;
    l_fs2_blocks NUMBER;
    l_fs3_blocks NUMBER;
    l_fs4_blocks NUMBER;
    l_full_bytes NUMBER;
    l_full_blocks NUMBER;
    l_unformatted_bytes NUMBER;
    l_unformatted_blocks NUMBER;
BEGIN
    DBMS_SPACE.SPACE_USAGE(
        segment_owner      => 'OE',
        segment_name       => 'PRODUCT_DESCRIPTIONS',
        segment_type       => 'TABLE',
        fs1_bytes          => l_fs1_bytes,
        fs1_blocks         => l_fs1_blocks,
        fs2_bytes          => l_fs2_bytes,
        fs2_blocks         => l_fs2_blocks,
        fs3_bytes          => l_fs3_bytes,
        fs3_blocks         => l_fs3_blocks,
        fs4_bytes          => l_fs4_bytes,
        fs4_blocks         => l_fs4_blocks,
        full_bytes         => l_full_bytes,
        full_blocks        => l_full_blocks,
        unformatted_blocks => l_unformatted_blocks,
        unformatted_bytes  => l_unformatted_bytes
    );
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = ' || l_fs1_blocks || '
                         Bytes = ' || l_fs1_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = ' || l_fs2_blocks || '
                         Bytes = ' || l_fs2_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS3 Blocks = ' || l_fs3_blocks || '
                         Bytes = ' || l_fs3_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS4 Blocks = ' || l_fs4_blocks || '
                         Bytes = ' || l_fs4_bytes);
    DBMS_OUTPUT.PUT_LINE('Full Blocks = ' || l_full_blocks || '
                         Bytes = ' || l_full_bytes);
    DBMS_OUTPUT.PUT_LINE('=====
                         =====');
END;
```

Practice 8 (continued)

```
DBMS_OUTPUT.PUT_LINE('Total Blocks =
'||to_char(l_fs1_blocks + l_fs2_blocks +
l_fs3_blocks + l_fs4_blocks + l_full_blocks)||' ||
Total Bytes ='|| to_char(l_fs1_bytes + l_fs2_bytes
+ l_fs3_bytes + l_fs4_bytes + l_full_bytes));
END;
/

set serveroutput on
execute check_space;
```


9

Performance and Tuning

ORACLE®

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

- Understand and influence the compiler
- Tune PL/SQL code
- Enable intraunit inlining



Objectives

In this lesson, the performance and tuning topics are divided into three main groups:

- Native and interpreted compilation
- Tuning PL/SQL code
- Intraunit inlining

In the compilation section, you learn about native and interpreted compilation.

In the “Tuning PL/SQL Code” section, you learn why it is important to write smaller, executable sections of code, when to use SQL or PL/SQL, how bulk binds can improve performance, how to use the FORALL syntax, how to rephrase conditional statements, about data types and constraint issues.

With inlining, the compiler reviews code to see whether it can be inlined rather than referenced. You can influence the inlining process.

Lesson Agenda

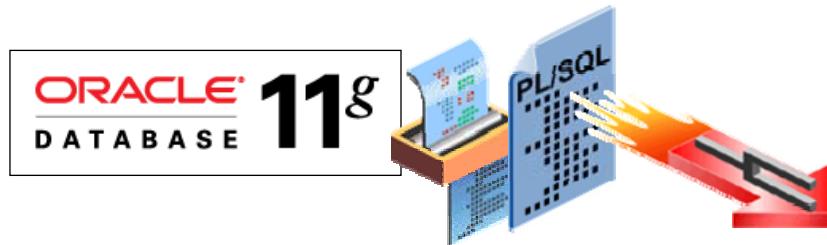
- Using native and interpreted compilation methods
- Tuning PL/SQL code
- Enabling intraunit inlining



Native and Interpreted Compilation

Two compilation methods:

- Interpreted compilation
 - Default compilation method
 - Interpreted at run time
- Native compilation
 - Compiles into native code
 - Stored in the SYSTEM tablespace



ORACLE®

9 - 4

Copyright © 2008, Oracle. All rights reserved.

Native and Interpreted Compilation

You can compile your PL/SQL code by using either native compilation or interpreted compilation.

With interpreted compilation, the PL/SQL statements in a PL/SQL program unit are compiled into an intermediate form, machine-readable code, which is stored in the database dictionary and interpreted at run time. You can use PL/SQL debugging tools on program units compiled for interpreted mode.

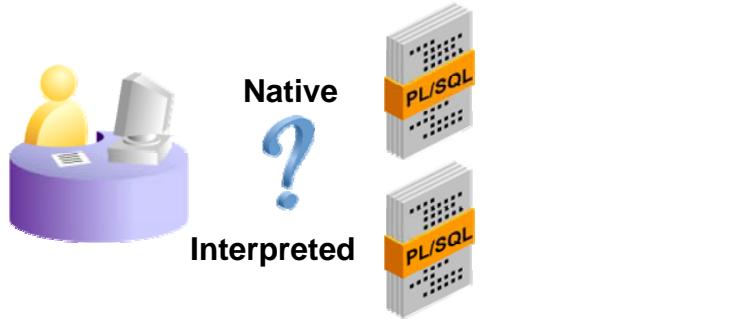
With PL/SQL native compilation, the PL/SQL statements in a PL/SQL program unit are compiled into native code and stored in the SYSTEM tablespace. Because the native code does not have to be interpreted at run time, it runs faster.

Native compilation applies only to PL/SQL statements. If your PL/SQL program contains only calls to SQL statements, it may not run faster when natively compiled, but it will run at least as fast as the corresponding interpreted code. The compiled code and the interpreted code make the same library calls, so their action is the same.

The first time a natively compiled PL/SQL program unit is executed, it is fetched from the SYSTEM tablespace into the shared memory. Regardless of how many sessions call the program unit, the shared memory has only one copy of it. If a program unit is not being used, the shared memory it is using might be freed, to reduce the memory load.

Deciding on a Compilation Method

- Use the interpreted mode when (typically during development):
 - You are using 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 are looking for increased performance in production



ORACLE®

9 - 5

Copyright © 2008, Oracle. All rights reserved.

Deciding on a Compilation Method

When deciding on a compilation method, you need to examine:

- Where you are in the development cycle
- What the program unit does

If you are debugging and recompiling program units frequently, the interpreted mode has these advantages:

- You can use PL/SQL debugging tools on program units compiled for interpreted mode (but not for those compiled for native mode).
- Compiling for interpreted mode is faster than compiling for native mode.

After completing the debugging phase of development, consider the following in determining whether to compile a PL/SQL program unit for native mode:

- The native mode provides the greatest performance gains for computation-intensive procedural operations. Examples are data warehouse applications and applications with extensive server-side transformations of data for display.
- The native mode provides the least performance gains for PL/SQL subprograms that spend most of their time executing SQL.
- When many program units (typically over 15,000) are compiled for native execution, and are simultaneously active, the large amount of shared memory required might affect system performance.

Setting the Compilation Method

- 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 fastest performance, use the following setting:

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

ORACLE®

9 - 6

Copyright © 2008, Oracle. All rights reserved.

Using the Initialization Parameters for PL/SQL Compilation

The PLSQL_CODE_TYPE Parameter

The PLSQL_CODE_TYPE compilation parameter determines whether the PL/SQL code is natively compiled or interpreted.

If you choose INTERPRETED:

- PL/SQL library units are compiled to PL/SQL bytecode format.
- These modules are executed by the PL/SQL interpreter engine.

If you choose NATIVE:

- PL/SQL library units (with the possible exception of top-level anonymous PL/SQL blocks) are compiled to native (machine) code.
- Such modules are executed natively without incurring interpreter overhead.

When the value of this parameter is changed, it has no effect on the PL/SQL library units that have already been compiled. The value of this parameter is stored persistently with each library unit. If a PL/SQL library unit is compiled natively, all subsequent automatic recompilations of that library unit use the native compilation. In Oracle Database 11g, native compilation is easier and more integrated, with fewer initialization parameters to set.

Using the PL/SQL Initialization Parameters (continued)

The `PLSQL_OPTIMIZE_LEVEL` Parameter

This parameter specifies the optimization level that is used to compile the PL/SQL library units. The higher the setting of this parameter, the more effort the compiler makes to optimize the PL/SQL library units. The available values are (0, 1, and 2 were available starting with Oracle 10g, release 2):

0: Maintains the evaluation order and hence the pattern of side effects, exceptions, and package initializations of Oracle9*i* and earlier releases. Also removes the new semantic identity of `BINARY_INTEGER` and `PLS_INTEGER`, and restores the earlier rules for the evaluation of integer expressions. Although the code runs somewhat faster than it did in Oracle9*i*, the use of level 0 forfeits most of the performance gains of PL/SQL starting with Oracle Database 10g.

1: Applies a wide range of optimizations to PL/SQL programs, including the elimination of unnecessary computations and exceptions, but generally does not move source code out of its original source order.

2: Applies a wide range of modern optimization techniques beyond those of level 1, including changes that may move source code relatively far from its original location.

3: This value is available in Oracle Database 11g. It applies a wide range of optimization techniques beyond those of level 2, automatically including techniques not specifically requested. This enables procedure inlining, which is an optimization process that replaces procedure calls with a copy of the body of the procedure to be called. The copied procedure almost always runs faster than the original call. To allow subprogram inlining, either accept the default value of the `PLSQL_OPTIMIZE_LEVEL` initialization parameter (which is 2) or set it to 3. With `PLSQL_OPTIMIZE_LEVEL` = 2, you must specify each subprogram to be inlined. With `PLSQL_OPTIMIZE_LEVEL` = 3, the PL/SQL compiler seeks opportunities to inline subprograms beyond those that you specify.

Generally, setting this parameter to 2 pays off in terms of better execution performance. If, however, the compiler runs slowly on a particular source module or if optimization does not make sense for some reason (for example, during rapid turnaround development), setting this parameter to 1 results in almost as good a compilation with less use of compile-time resources. The value of this parameter is stored persistently with the library unit.

Viewing the Compilation Settings

Use the USER | ALL | DBA_PLSQL_OBJECT_SETTINGS data dictionary views to display the settings for a PL/SQL object:

```
DESCRIBE ALL_PLSQL_OBJECT_SETTINGS
```

Name	Null?	Type
OWNER	NOT NULL	VARCHAR2(30)
NAME	NOT NULL	VARCHAR2(30)
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)

ORACLE®

Displaying the PL/SQL Initialization Parameters

The columns of the USER_PLSQL_OBJECTS_SETTINGS data dictionary view include:

Owner: The owner of the object. This column is not displayed in the USER_PLSQL_OBJECTS_SETTINGS view.

Name: The name of the object

Type: The available choices are PROCEDURE, FUNCTION, PACKAGE, PACKAGE BODY, TRIGGER, TYPE, or TYPE BODY.

PLSQL_OPTIMIZE_LEVEL: The optimization level that was used to compile the object

PLSQL_CODE_TYPE: The compilation mode for the object

PLSQL_DEBUG: Specifies whether or not the object was compiled for debugging

PLSQL_WARNINGS: The compiler warning settings used to compile the object

NLS_LENGTH_SEMANTICS: The national language support (NLS) length semantics used to compile the object

PLSQL_CCFLAGS: The conditional compilation flag used to compile the object

PLSCOPE_SETTINGS: Controls the compile time collection, cross reference, and storage of PL/SQL source code identifier data (new in Oracle Database 11g)

Viewing the Compilation Settings

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

NAME	PLSQL_CODE_TYP	PLSQL_OPTIMIZE_LEVEL
ACTIONS_T	INTERPRETED	2
ACTION_T	INTERPRETED	2
ACTION_V	INTERPRETED	2
ADD_ORDER_ITEMS	INTERPRETED	2
CATALOG_TYP	INTERPRETED	2
CATALOG_TYP	INTERPRETED	2
CATALOG_TYP	INTERPRETED	2
CATEGORY_TYP	INTERPRETED	2
CATEGORY_TYP	INTERPRETED	2
COMPOSITE_CATEGORY_TYP	INTERPRETED	2
...		

ORACLE®

Displaying the PL/SQL Initialization Parameters (continued)

Set the values of the compiler initialization parameters by using the ALTER SYSTEM or ALTER SESSION statements.

The parameters' values are accessed when the CREATE OR REPLACE or ALTER statements are executed.

Setting Up a Database for Native Compilation

- This requires DBA privileges.
- The PLSQL_CODE_TYPE compilation parameter must be set to NATIVE.
- The benefits apply to all the built-in PL/SQL packages that are used for many database operations.

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

ORACLE®

9 - 10

Copyright © 2008, Oracle. All rights reserved.

Setting Up a Database for Native Compilation

If you have DBA privileges, you can set up a new database for PL/SQL native compilation by setting the PLSQL_CODE_TYPE compilation parameter to NATIVE. The performance benefits apply to all built-in PL/SQL packages that are used for many database operations.

Compiling a Program Unit for Native Compilation

```
SELECT name, plsql_code_type, plsql_optimize_level ①
FROM   user_plsql_object_settings
WHERE  name = 'ADD_ORDER_ITEMS';

NAME          PLSQL_CODE_T  PLSQL_OPTIMIZE_LEVEL
-----
ADD_ORDER_ITEMS    INTERPRETED      2
ALTER SESSION SET PLSQL_CODE_TYPE = 'NATIVE'; ②
ALTER PROCEDURE add_order_items COMPILE;        ③

SELECT name, plsql_code_type, plsql_optimize_level ④
FROM   user_plsql_object_settings
WHERE  name = 'ADD_ORDER_ITEMS';

NAME          PLSQL_CODE_T  PLSQL_OPTIMIZE_LEVEL
-----
ADD_ORDER_ITEMS    NATIVE        2
```

ORACLE®

9 - 11

Copyright © 2008, Oracle. All rights reserved.

Changing PL/SQL Initialization Parameters: Example

To change a compiled PL/SQL object from interpreted code type to native code type, you must set the PLSQL_CODE_TYPE parameter to NATIVE (optionally set the other parameters), and then recompile the program.

In the example shown above:

1. The compilation type is checked on the ADD_ORDER_ITEMS program unit.
2. The compilation method is set to NATIVE at the session level.
3. The ADD_ORDER_ITEMS program unit is recompiled.
4. The compilation type is checked again on the ADD_ORDER_ITEMS program unit to verify that it changed.

If you want to compile an entire database for native or interpreted compilation, scripts are provided to help you do so.

- You require DBA privileges.
- Set PLSQL_CODE_TYPE at the system level.
- Run the dbmsupgnv.sql-supplied script that is found in the
 \Oraclehome\product\11.1.0\db_1\RDBMS\ADMIN folder.

For detailed information, see the *Oracle® Database PL/SQL Language Reference 11g* reference manual.

Lesson Agenda

- Using native and interpreted compilation methods
- Tuning PL/SQL code
- Enabling intraunit inlining



Tuning PL/SQL Code

You can tune your PL/SQL code 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
- Comparing SQL with PL/SQL
- Understanding how bulk binds can improve performance
- Using the FORALL support with bulk binding
- Handling and saving exceptions with the SAVE EXCEPTIONS syntax
- Rephrasing conditional statements

ORACLE®

9 - 13

Copyright © 2008, Oracle. All rights reserved.

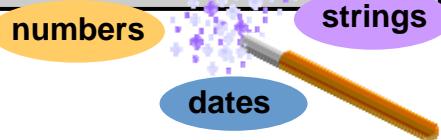
Tuning PL/SQL Code

By tuning your PL/SQL code, you can tailor its performance to best meet your needs. In the following pages, you learn about some of the main PL/SQL tuning issues that can improve the performance of your PL/SQL applications.

Avoiding Implicit Data Type Conversion

- PL/SQL performs implicit conversions between structurally different data types.
- Example: When assigning a PLS_INTEGER variable to a NUMBER variable

```
DECLARE
  n NUMBER;
BEGIN
  n := n + 15;      -- converted
  n := n + 15.0;    -- not converted
  ...
END;
```



ORACLE

9 - 14

Copyright © 2008, Oracle. All rights reserved.

Avoiding Implicit Data Type Conversion

At run time, PL/SQL automatically performs implicit conversions between structurally different data types. By avoiding implicit conversions, you can improve the performance of your code.

The major problems with implicit data type conversion are:

- It is nonintuitive and can result in unexpected results.
- You have no control over the implicit conversion.

In the slide example, assigning a PLS_INTEGER variable to a NUMBER variable or vice versa results in a conversion, because their representations are different. Such implicit conversions can happen during parameter passing as well. The integer literal 15 is represented internally as a signed 4-byte quantity, so PL/SQL must convert it to an Oracle number before the addition. However, the floating-point literal 15.0 is represented as a 22-byte Oracle number, so no conversion is necessary.

To avoid implicit data type conversion, you can use the built-in functions:

- TO_DATE
- TO_NUMBER
- TO_CHAR
- CAST

Understanding the NOT NULL Constraint

```
PROCEDURE calc_m IS
  m NUMBER NOT NULL:=0;
  a NUMBER;
  b NUMBER;
BEGIN
  ...
  m := a + b;
  ...
END;
```

```
PROCEDURE calc_m IS
  m NUMBER; --no
             --constraint
  a NUMBER;
  b NUMBER;
BEGIN
  ...
  m := a + b;
  IF m IS NULL THEN
    -- raise error
  END IF;
END;
```

ORACLE®

9 - 15

Copyright © 2008, Oracle. All rights reserved.

The NOT NULL Constraint

In PL/SQL, using the NOT NULL constraint incurs a small performance cost. Therefore, use it with care. Consider the example on the left in the slide that uses the NOT NULL constraint for *m*. Because *m* is constrained by NOT NULL, the value of the expression *a* + *b* is assigned to a temporary variable, which is then tested for nullity. If the variable is not null, its value is assigned to *m*. Otherwise, an exception is raised. However, if *m* were not constrained, the value would be assigned to *m* directly.

A more efficient way to write the same example is shown on the right in the slide.

Note that the subtypes NATURALN and POSTIVEN are defined as the NOT NULL subtypes of NATURAL and POSITIVE. Using them incurs the same performance cost as seen above.

Using the NOT NULL Constraint

Slower

No extra coding is needed.

When an error is implicitly raised, the value of *m* is preserved.

Not Using the Constraint

Faster

Requires extra coding that is error prone

When an error is explicitly raised, the old value of *m* is lost.

Using the 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.



ORACLE®

9 - 16

Copyright © 2008, Oracle. All rights reserved.

Using the PLS_INTEGER Data Type for All Integer Operations

When you need to declare an integer variable, use the PLS_INTEGER data type, which is the most efficient numeric type. That is because PLS_INTEGER values require less storage than INTEGER or NUMBER values, which are represented internally as 22-byte Oracle numbers. Also, PLS_INTEGER operations use machine arithmetic, so they are faster than BINARY_INTEGER, INTEGER, or NUMBER operations, which use library arithmetic.

Furthermore, INTEGER, NATURAL, NATURALN, POSITIVE, POSITIVEN, and SIGNEDTYPE are constrained subtypes. Their variables require precision checking at run time that can affect the performance.

The BINARY_FLOAT and BINARY_DOUBLE data types are also faster than the NUMBER data type.

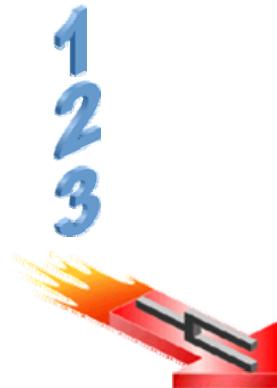
Using the SIMPLE_INTEGER Data Type

- Definition:

- Is a predefined subtype
- Has the range $-2147483648 .. 2147483648$
- 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



ORACLE

9 - 17

Copyright © 2008, Oracle. All rights reserved.

Using the SIMPLE_INTEGER Data Type

The SIMPLE_INTEGER data type is a predefined subtype of the BINARY_INTEGER (or PLS_INTEGER) data type that has the same numeric range as BINARY_INTEGER. It differs significantly from PLS_INTEGER in its overflow semantics. Incrementing the largest SIMPLE_INTEGER value by one silently produces the smallest value, and decrementing the smallest value by one silently produces the largest value. These “wrap around” semantics conform to the Institute of Electrical and Electronics Engineers (IEEE) standard for 32-bit integer arithmetic.

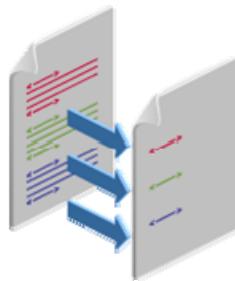
The key features of the SIMPLE_INTEGER predefined subtype are the following:

- Includes the range of $-2147483648.. +2147483648$
- Has a not null constraint
- Wraps rather than overflows
- Is faster than PLS_INTEGER

Without the overhead of overflow checking and nullness checking, the SIMPLE_INTEGER data type provides significantly better performance than PLS_INTEGER when the parameter PLSQL_CODE_TYPE is set to native, because arithmetic operations on the former are performed directly in the machine’s hardware. The performance difference is less noticeable when the parameter PLSQL_CODE_TYPE is set to interpreted but even with this setting, the SIMPLE_INTEGER type is faster than the PLS_INTEGER type.

Modularizing Your Code

- Limit the number of lines of code between a BEGIN and END to about a page or 60 lines of code.
- Use packaged programs to keep each executable section small.
- Use local procedures and functions to hide logic.
- Use a function interface to hide formulas and business rules.



ORACLE®

9 - 18

Copyright © 2008, Oracle. All rights reserved.

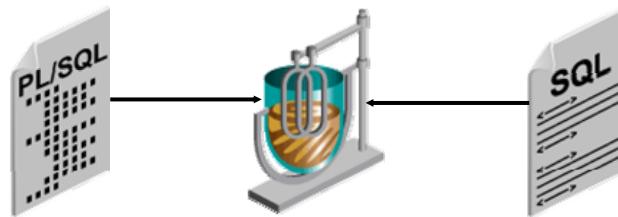
Write Smaller Executable Sections

By writing smaller sections of executable code, you can make the code easier to read, understand, and maintain. When developing an application, use a stepwise refinement. Make a general description of what you want your program to do, and then implement the details in subroutines. Using local modules and packaged programs can help keep each executable section small. This makes it easier for you to debug and refine your code.

Comparing SQL with PL/SQL

Each has its own benefits:

- SQL:
 - Accesses data in the database
 - Treats data as sets
- PL/SQL:
 - Provides procedural capabilities
 - Has more flexibility built into the language



ORACLE®

9 - 19

Copyright © 2008, Oracle. All rights reserved.

SQL Versus PL/SQL

Both SQL and PL/SQL have their strengths. However, there are situations where one language is more appropriate to use than the other.

You use SQL to access data in the database with its powerful statements. SQL processes sets of data as groups rather than as individual units. The flow-control statements of most programming languages are absent in SQL, but present in PL/SQL. When using SQL in your PL/SQL applications, be sure not to repeat a SQL statement. Instead, encapsulate your SQL statements in a package and make calls to the package.

Using PL/SQL, you can take advantage of the PL/SQL-specific enhancements for SQL, such as autonomous transactions, fetching into cursor records, using a cursor FOR loop, using the RETURNING clause for information about modified rows, and using BULK COLLECT to improve the performance of multiple-row queries.

Though there are advantages of using PL/SQL over SQL in several cases, use PL/SQL with caution, especially under the following circumstances:

- Performing high-volume inserts
- Using user-defined PL/SQL functions
- Using external procedure calls
- Using the utl_file package as an alternative to SQL*Plus in high-volume reporting

Comparing SQL with PL/SQL

- Some simple set processing is markedly faster than the equivalent PL/SQL.

```
BEGIN
  INSERT INTO inventories2
    SELECT product_id, warehouse_id
      FROM main_inventories;
END;
```

- Avoid using procedural code when it may be better to use SQL.

```
...FOR I IN 1..5600 LOOP
  counter := counter + 1;
  SELECT product_id, warehouse_id
    INTO v_p_id, v_wh_id
     FROM big_inventories WHERE v_p_id = counter;
  INSERT INTO inventories2 VALUES(v_p_id, v_wh_id);
END LOOP;...
```



ORACLE®

9 - 20

Copyright © 2008, Oracle. All rights reserved.

SQL Versus PL/SQL (continued)

The SQL statement explained in the slide is a great deal faster than the equivalent PL/SQL loop. Take advantage of the simple set processing operations that are implicitly available in the SQL language, as it can run markedly faster than the equivalent PL/SQL loop. Avoid writing procedural code when SQL would work better.

However, there are occasions when you will get better performance from PL/SQL, even when the process could be written in SQL. Correlated updates are slow. With correlated updates, a better method is to access only correct rows by using PL/SQL. The following PL/SQL loop is faster than the equivalent correlated update SQL statement.

```
DECLARE
  CURSOR cv_raise IS
    SELECT deptno, increase
      FROM emp_raise;
BEGIN
  FOR dept IN cv_raise LOOP
    UPDATE big_emp
      SET sal = sal * dept.increase
        WHERE deptno = dept.deptno;
  END LOOP;
  ...

```

Comparing SQL with PL/SQL

- Instead of:

```
...
INSERT INTO order_items
  (order_id, line_item_id, product_id,
   unit_price, quantity)
VALUES (...)
```

- Create a stand-alone procedure:

```
insert_order_item (
  2458, 6, 3515, 2.00, 4);
```

- Or a packaged procedure:

```
orderitems.ins (
  2458, 6, 3515, 2.00, 4);
```



Encapsulating SQL Statements

From a design standpoint, do not embed your SQL statements directly within the application code. It is better if you write procedures to perform your SQL statements.

Pros

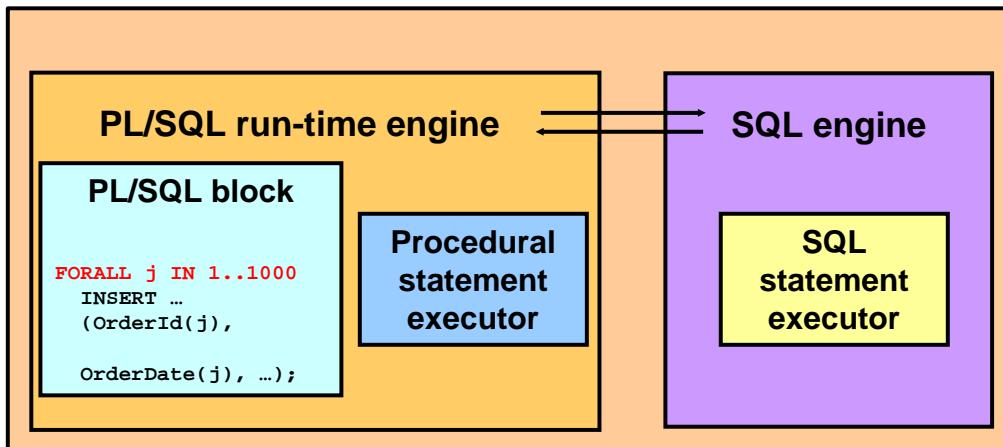
- If you design your application so that all programs that perform an insert on a specific table use the same `INSERT` statement, your application will run faster because of less parsing and reduced demands on the System Global Area (SGA) memory.
- Your program will also handle data manipulation language (DML) errors consistently.

Cons

- You may need to write more procedural code.
- You may need to write several variations of update or insert procedures to handle the combinations of columns that you are updating or inserting into.

Using Bulk Binding

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



Using Bulk Binding

With bulk binds, you can improve performance by decreasing the number of context switches between the SQL and PL/SQL engines. When a PL/SQL program executes, each time a SQL statement is encountered, there is a switch between the PL/SQL engine and the SQL engine. The more the number of switches, the less the efficiency.

Improved Performance

Bulk binding enables you to implement array fetching. With bulk binding, entire collections, not just individual elements, are passed back and forth. Bulk binding can be used with nested tables, varrays, and associative arrays.

The more the rows affected by a SQL statement, the greater is the performance gain with bulk binding.

Using Bulk Binding

Bind whole arrays of values simultaneously, rather than looping to perform fetch, insert, update, and delete on multiple rows.

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

ORACLE®

9 - 23

Copyright © 2008, Oracle. All rights reserved.

Using Bulk Binding (continued)

In the first example shown in the slide, one row at a time is inserted into the target table. In the second example, the FOR loop is changed to a FORALL (which has an implicit loop) and all the immediately subsequent DML statements are processed in bulk. The entire code examples, along with the timing statistics for running each FOR loop example, are as follows.

First, create the demonstration table:

```
CREATE TABLE bulk_bind_example_tbl (
    num_col NUMBER,
    date_col DATE,
    char_col VARCHAR2(40));
```

Second, set the SQL*Plus TIMING variable on. Setting it on enables you to see the approximate elapsed time of the last SQL statement:

```
SET TIMING ON
```

Third, run this block of code that includes a FOR loop to insert 50,000 rows:

```
DECLARE
    TYPE typ_numlist IS TABLE OF NUMBER;
    TYPE typ_datalist IS TABLE OF DATE;
    TYPE typ_charlist IS TABLE OF VARCHAR2(40)
        INDEX BY PLS_INTEGER;
-- continued on the next page
```

Using Bulk Binding (continued)

```
-- continued from previous page
  n typ_numlist := typ_numlist();
  d typ_datelist := typ_datelist();
  c typ_charlist;

BEGIN
  FOR i IN 1 .. 50000 LOOP
    n.extend;
    n(i) := i;
    d.extend;
    d(i) := sysdate + 1;
    c(i) := lpad(1, 40);
  END LOOP;
  FOR I in 1 .. 50000 LOOP
    INSERT INTO bulk_bind_example_tbl
      VALUES (n(i), d(i), c(i));
  END LOOP;
END;
/
```

2.184ms elapsed

Last, run this block of code that includes a FORALL loop to insert 50,000 rows. Note the significant decrease in the timing when using the FORALL processing:

```
DECLARE
  TYPE typ_numlist IS TABLE OF NUMBER;
  TYPE typ_datelist IS TABLE OF DATE;
  TYPE typ_charlist IS TABLE OF VARCHAR2(40)
    INDEX BY PLS_INTEGER;

  n typ_numlist := typ_numlist();
  d typ_datelist := typ_datelist();
  c typ_charlist;

BEGIN
  FOR i IN 1 .. 50000 LOOP
    n.extend;
    n(i) := i;
    d.extend;
    d(i) := sysdate + 1;
    c(i) := lpad(1, 40);
  END LOOP;
  FORALL I in 1 .. 50000
    INSERT INTO bulk_bind_example_tbl
      VALUES (n(i), d(i), c(i));
END;
/
```

828ms elapsed

Using Bulk Binding

Use BULK COLLECT to improve performance:

```
CREATE OR REPLACE PROCEDURE process_customers
  (p_account_mgr customers.account_mgr_id%TYPE)
IS
  TYPE typ_numtab IS TABLE OF
    customers.customer_id%TYPE;
  TYPE typ_chartab IS TABLE OF
    customers.cust_last_name%TYPE;
  TYPE typ_emailtab IS TABLE OF
    customers.cust_email%TYPE;
  v_custnos      typ_numtab;
  v_last_names   typ_chartab;
  v_emails       typ_emailtab;
BEGIN
  SELECT customer_id, cust_last_name, cust_email
    BULK COLLECT INTO v_custnos, v_last_names, v_emails
   FROM customers
  WHERE account_mgr_id = p_account_mgr;
  ...
END process_customers;
```

ORACLE®

9 - 25

Copyright © 2008, Oracle. All rights reserved.

Using BULK COLLECT

When you require a large number of rows to be returned from the database, you can use the BULK COLLECT option for queries. This option enables you to retrieve multiple rows of data in a single request. The retrieved data is then populated into a series of collection variables. This query runs significantly faster than if it were done without the BULK COLLECT.

You can use the BULK COLLECT option with explicit cursors too:

```
BEGIN
  OPEN cv_customers INTO customers_rec;
  FETCH cv_customers BULK COLLECT INTO
    v_custnos, v_last_name, v_mails;
  ...
```

You can also use the LIMIT option with BULK COLLECT. This gives you control over the amount of processed rows in one step.

```
FETCH cv_customers BULK COLLECT
  INTO v_custnos, v_last_name, v_email
  LIMIT 200;
```

Using Bulk Binding

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;
```

ORACLE®

9 - 26

Copyright © 2008, Oracle. All rights reserved.

The RETURNING Clause

Often, applications need information about the row that is affected by a SQL operation; for example, to generate a report or take action. Using the RETURNING clause, you can retrieve information about the rows that you modified with the INSERT, UPDATE, and DELETE statements. This can improve performance, because it enables you to make changes, and at the same time, collect information about the data being changed. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required. Without the RETURNING clause, you need two operations: one to make the change, and a second operation to retrieve information about the change. In the slide example, the order_total information is retrieved from the ORDERS table and collected into the totlist collection. The totlist collection is returned in bulk to the PL/SQL engine.

If you did not use the RETURNING clause, you would need to perform two operations, one for the UPDATE, and another for the SELECT:

```
UPDATE orders SET order_total = .95 * order_total
WHERE sales_rep_id = p_id
AND order_total > c_big_total;

SELECT order_total FROM orders
WHERE sales_rep_id = p_id AND order_total > c_big_total;
```

The RETURNING Clause (continued)

In the following example, you update the credit limit of a customer and at the same time retrieve the customer's new credit limit into a SQL Developer environment variable:

```
CREATE OR REPLACE PROCEDURE change_credit
  (p_in_id      IN    customers.customer_id%TYPE,
   o_credit OUT NUMBER)
IS
BEGIN
  UPDATE customers
  SET    credit_limit = credit_limit * 1.10
  WHERE  customer_id = p_in_id
  RETURNING credit_limit INTO o_credit;
END change_credit;
/
VARIABLE g_credit NUMBER
EXECUTE change_credit(109, :g_credit)
PRINT g_credit
```

Using SAVE EXCEPTIONS

- 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
ERROR_INDEX	Holds the iteration of the <code>FORALL</code> statement where the exception was raised
ERROR_CODE	Holds the corresponding Oracle error code

- Note that the values always refer to the most recently executed `FORALL` statement.

ORACLE®

Handling FORALL Exceptions

To handle the exceptions encountered during a BULK BIND operation, you can add the keyword `SAVE EXCEPTIONS` to your `FORALL` statement. Without it, if a row fails during the `FORALL` loop, the loop execution is terminated. `SAVE_EXCEPTIONS` allows the loop to continue processing and is required if you want the loop to continue.

All exceptions raised during the execution are saved in the `%BULK_EXCEPTIONS` cursor attribute, which stores a collection of records. This cursor attribute is available only from the exception handler.

Each record has two fields. The first field, `%BULK_EXCEPTIONS(i).ERROR_INDEX`, holds the “iteration” of the `FORALL` statement during which the exception was raised. The second field, `BULK_EXCEPTIONS(i).ERROR_CODE`, holds the corresponding Oracle error code.

The values stored by `%BULK_EXCEPTIONS` always refer to the most recently executed `FORALL` statement. The number of exceptions is saved in the count attribute of `%BULK_EXCEPTIONS`; that is, `%BULK_EXCEPTIONS.COUNT`. Its subscripts range from 1 to COUNT. If you omit the `SAVE EXCEPTIONS` keyword, execution of the `FORALL` statement stops when an exception is raised. In that case, `SQL%BULK_EXCEPTIONS.COUNT` returns 1, and `SQL%BULK_EXCEPTIONS` contains just one record. If no exception is raised during the execution, `SQL%BULK_EXCEPTIONS.COUNT` returns 0.

Handling FORALL Exceptions

```
DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    num_tab    NumList :=
        NumList(100,0,110,300,0,199,200,0,400);
    bulk_errors EXCEPTION;
    PRAGMA      EXCEPTION_INIT (bulk_errors, -24381 );
BEGIN
    FORALL i IN num_tab.FIRST..num_tab.LAST
    SAVE EXCEPTIONS
    DELETE FROM orders WHERE order_total < 500000/num_tab(i);
EXCEPTION WHEN bulk_errors THEN
    DBMS_OUTPUT.PUT_LINE('Number of errors is: '
                         || SQL%BULK_EXCEPTIONS.COUNT);
    FOR j in 1..SQL%BULK_EXCEPTIONS.COUNT
    LOOP
        DBMS_OUTPUT.PUT_LINE (
            TO_CHAR(SQL%BULK_EXCEPTIONS(j).error_index) ||
            ' / ' ||
            SQLERRM(-SQL%BULK_EXCEPTIONS(j).error_code));
    END LOOP;
END;
/
```

ORACLE®

9 - 29

Copyright © 2008, Oracle. All rights reserved.

Example

In this example, the EXCEPTION_INIT pragma defines an exception named BULK_ERRORS and associates the name with the ORA-24381 code, which is an “Error in Array DML.” The PL/SQL block raises the predefined exception ZERO_DIVIDE when i equals 2, 5, 8. After the bulk bind is completed, SQL%BULK_EXCEPTIONS.COUNT returns 3, because the code tried to divide by zero three times. To get the Oracle error message (which includes the code), you pass SQL%BULK_EXCEPTIONS(i).ERROR_CODE to the error-reporting function SQLERRM. Here is the output:

```
Number of errors is: 5
Number of errors is: 3
2 / ORA-01476: divisor is equal to zero
5 / ORA-01476: divisor is equal to zero
8 / ORA-01476: divisor is equal to zero
```

Rephrasing Conditional Control Statements

In logical expressions, PL/SQL stops evaluating the expression as soon as the result is determined.

- Scenario 1:

```
IF TRUE | FALSE OR (v_sales_rep_id IS NULL) THEN  
  ...  
  ...  
END IF;
```

- Scenario 2:

```
IF credit_ok(cust_id) AND (v_order_total < 5000) THEN  
  ...  
END IF;
```

ORACLE®

9 - 30

Copyright © 2008, Oracle. All rights reserved.

Rephrasing Conditional Control Statements

In logical expressions, improve performance by carefully tuning conditional constructs.

When evaluating a logical expression, PL/SQL stops evaluating the expression as soon as the result is determined. For example, in the first scenario in the slide, which involves an OR expression, when the value of the left operand yields TRUE, PL/SQL need not evaluate the right operand (because OR returns TRUE if either of its operands is true).

Now, consider the second scenario in the slide, which involves an AND expression. The Boolean function CREDIT_OK is always called. However, if you switch the operands of AND as follows, the function is called only when the expression v_order_total < 5000 is true (because AND returns TRUE only if both its operands are true):

```
IF (v_order_total < 5000 ) AND credit_ok(cust_id) THEN  
  ...  
END IF;
```

Rephrasing 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;
```



ORACLE®

9 - 31

Copyright © 2008, Oracle. All rights reserved.

Mutually Exclusive Conditions

If you have a situation where you are checking a list of choices for a mutually exclusive result, use the ELSIF syntax, as it offers the most efficient implementation. With ELSIF, after a branch evaluates to TRUE, the other branches are not evaluated.

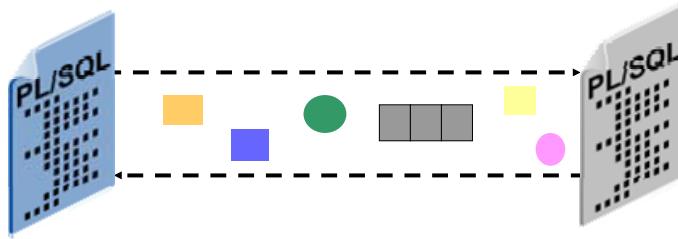
In the example shown on the right in the slide, every IF statement is executed. In the example on the left, after a branch is found to be true, the rest of the branch conditions are not evaluated. Sometimes you do not need an IF statement. For example, the following code can be rewritten without an IF statement:

```
IF date_ordered < sysdate + 7 THEN
    late_order := TRUE;
ELSE
    late_order := FALSE;
END IF;

--rewritten without an IF statement:
late_order := date_ordered < sysdate + 7;
```

Passing Data Between PL/SQL Programs

- The flexibility built into PL/SQL enables you to pass:
 - Simple scalar variables
 - Complex data structures
- You can use the `NOCOPY` hint to improve performance with the `IN OUT` parameters.



ORACLE®

Passing Data Between PL/SQL Programs

You can pass simple scalar data or complex data structures between PL/SQL programs.

When passing collections as parameters, you may encounter a slight decrease in performance as compared with passing scalar data but the performance is still comparable. However, when passing `IN OUT` parameters that are complex (such as collections) to a procedure, you will experience significantly more overhead, because a copy of the parameter value is stored before the routine is executed. The stored value must be kept in case an exception occurs. You can use the `NOCOPY` compiler hint to improve performance in this situation. `NOCOPY` instructs the compiler not to make a backup copy of the parameter that is being passed. However, be careful when you use the `NOCOPY` compiler hint, because your results are not predictable if your program encounters an exception.

Passing Data Between PL/SQL Programs

Pass records as parameters to encapsulate data, as well as, 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);
```

ORACLE®

Passing Records as Arguments

You can declare user-defined records as formal parameters of procedures and functions as shown in the slide. By using records to pass values, you are encapsulating the data being passed. This requires less coding than defining, assigning, and manipulating each record field individually.

When you call a function that returns a record, use the notation:

```
function_name(parameters).field_name
```

For example, the following call to the NTH_HIGHEST_ORD_TOTAL function references the ORDER_TOTAL field in the ORD_INFO record:

```
DECLARE
  TYPE OrdRec IS RECORD (
    v_order_id      NUMBER(6),
    v_order_total   REAL);
    v_middle_total  REAL;
  FUNCTION nth_highest_total (n INTEGER) RETURN OrdRec IS
    order_info OrdRec;
  BEGIN
    ...
    RETURN order_info; -- return record
  END;
  BEGIN
    -- call function
    v_middle_total := nth_highest_total(10).v_order_total;
  ...
  Oracle Database 11g: Advanced PL/SQL 9 - 33
```

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;
```

ORACLE®

9 - 34

Copyright © 2008, Oracle. All rights reserved.

Passing Collections as Arguments

You can declare collections as formal parameters of procedures and functions. In the example in the slide, associative arrays are declared as the formal parameters of two packaged procedures. If you were to use scalar variables to pass the data, you would need to code and maintain many more declarations.

Lesson Agenda

- Using native and interpreted compilation methods
- Tuning PL/SQL code
- Enabling intraunit inlining



Introducing Intraunit Inlining

- 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:
 - Inlining can provide large performance gains when applied judiciously by a factor of 2–10 times.

ORACLE®

9 - 36

Copyright © 2008, Oracle. All rights reserved.

Introducing Inlining

Procedure inlining is an optimization process that replaces procedure calls with a copy of the body of the procedure to be called. The copied procedure almost always runs faster than the original call, because:

- The need to create and initialize the stack frame for the called procedure is eliminated.
- The optimization can be applied over the combined text of the call context and the copied procedure body.
- Propagation of constant actual arguments often causes the copied body to collapse under optimization.

When inlining is achieved, you can see performance gains of 2–10 times.

With Oracle Database 11g, the PL/SQL compiler can automatically find calls that should be inlined, and can do the inlining correctly and quickly. There are some controls to specify where and when the compiler should do this work (using the `PLSQL_OPTIMIZATION_LEVEL` database parameter), but usually, a general request is sufficient.

Using Inlining

- Influence implementing inlining via two methods:
 - Oracle parameter PLSQL_OPTIMIZE_LEVEL
 - PRAGMA INLINE
- Recommend that you:
 - Inline small programs
 - Inline programs that are frequently executed
- Use performance tools to identify the hotspots that are suitable for inline applications:
 - plstimer



ORACLE®

9 - 37

Copyright © 2008, Oracle. All rights reserved.

Using Inlining

When implementing inlining, it is recommended that the process be applied to smaller programs, and/or programs that execute frequently. For example, you may want to inline small helper programs.

To help you identify which programs to inline, you can use the `plstimer` PL/SQL performance tool. This tool specifically analyzes program performance in terms of the time spent in procedures and the time spent on particular call sites. It is important that you identify the procedure calls that may benefit from inlining.

There are two ways to use inlining:

1. Set the `PLSQL_OPTIMIZE_LEVEL` parameter to 3. When this parameter is set to 3, the PL/SQL compiler searches for calls that might profit from inlining and inlines the most profitable calls. Profitability is measured by those calls that help the program speed up the most and keep the compiled object program as short as possible.

```
ALTER SESSION SET plsql_optimize_level = 3;
```

2. Use `PRAGMA INLINE` in your PL/SQL code. This identifies whether a specific call should be inlined. Setting this pragma to “YES” has an effect only if the optimize level is set to two or higher.

Inlining Concepts

Noninlined program:

```
CREATE OR REPLACE PROCEDURE small_pgm
IS
  a NUMBER;
  b NUMBER;

  PROCEDURE touch(x IN OUT NUMBER, y NUMBER)
  IS
  BEGIN
    IF y > 0 THEN
      x := x*x;
    END IF;
  END;

  BEGIN
    a := b;
    FOR i IN 1..10 LOOP
      touch(a, -17);
      a := a*b;
    END LOOP;
  END small_pgm;
```

ORACLE®

9 - 38

Copyright © 2008, Oracle. All rights reserved.

Inlining Concepts

The example shown in the slide will be expanded to show you how a procedure is inlined. The `a := a * b` assignment at the end of the loop looks like it could be moved before the loop. However, it cannot be, because `a` is passed as an `IN OUT` parameter to the `TOUCH` procedure. The compiler cannot be certain what the procedure does to its parameters. This results in the multiplication and in the assignment's being completed 10 times instead of only once, even though multiple executions are not necessary.

Inlining Concepts

Examine the loop after inlining:

```
...
BEGIN
    a := b;
    FOR i IN 1..10 LOOP
        IF -17 > 0 THEN
            a := a*a;
        END IF;
        a := a*b;
    END LOOP;
END small_pgm;
...
```

ORACLE®

9 - 39

Copyright © 2008, Oracle. All rights reserved.

Inlining Concepts (continued)

The code in the slide shows what happens to the loop after inlining.

Inlining Concepts

The loop is transformed in several steps:

```
a := b;
FOR i IN 1..10 LOOP ...
  IF false THEN
    a := a*a;
  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;
```

ORACLE®

9 - 40

Copyright © 2008, Oracle. All rights reserved.

Inlining Concepts (continued)

Because the insides of the procedure are now visible to the compiler, it can transform the loop in several steps, as shown in the slide.

Instead of 11 assignments (one outside of the loop) and 10 multiplications, only one assignment and one multiplication are performed. If the loop ran a million times (instead of 10), the savings would be a million assignments. For code that contains deep loops that are executed frequently, inlining offers tremendous savings.

Inlining: Example

- 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.

- Within a PL/SQL subroutine, use PRAGMA INLINE:

- NO means no inlining occurs regardless of the level and regardless of the YES pragmas.
- YES means inline at level 2 of a particular call and increase the priority of inlining at level 3 for the call.



Inlining Concepts (continued)

To influence the optimizer to use inlining, you can set the PLSQL_OPTIMIZE_LEVEL parameter to a value of 2 or higher. By setting this parameter, you are making a request that inlining be used. It is up to the compiler to analyze the code and determine whether inlining is appropriate. When the optimize level is set to 3, the PL/SQL compiler searches for calls that might profit from inlining and inlines the most profitable calls.

In rare cases, if the overhead of the optimizer makes the compilation of very large applications take too long, you can lower the optimization by setting PLSQL_OPTIMIZE_LEVEL=1 instead of its default value of 2. In even rarer cases, you might see a change in exception action, either an exception that is not raised at all, or one that is raised earlier than expected. Setting PLSQL_OPTIMIZE_LEVEL=1 prevents the code from being rearranged.

To enable inlining within a PL/SQL subroutine, you can use PRAGMA INLINE to suggest that a specific call be inlined.

Inlining: 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;
```

ORACLE®

9 - 42

Copyright © 2008, Oracle. All rights reserved.

Inlining Concepts (continued)

Within a PL/SQL subroutine, you can use PRAGMA INLINE to suggest that a specific call be inlined. When using PRAGMA INLINE, the first argument is the simple name of a subroutine, a function name, a procedure name, or a method name. The second argument is either the constant string ‘NO’ or ‘YES.’ The pragma can go before any statement or declaration. If you put it in the wrong place, you receive a syntax error message from the compiler.

To identify that a specific call should not be inlined, use:

```
PRAGMA INLINE (function_name, 'NO');
```

Setting the PRAGMA INLINE to ‘NO’ always works, regardless of any other pragmas that might also apply to the same statement. The pragma also applies at all optimization levels, and it applies no matter how badly the compiler would like to inline a particular call. If you are certain that you do not want some code inlined (perhaps due to the large size), you can set this to NO.

Setting the PRAGMA INLINE to ‘YES’ strongly encourages the compiler to inline the call. The compiler keeps track of the resources used during inlining and makes the decision to stop inlining when the cost becomes too high.

If inlining is requested and you have the compiler warnings turned on, you see the message:

```
PLW-06004: inlining of call of procedure ADD_IT requested.
```

If inlining is applied, you see the compiler warning (it is more of a message):

```
PLW-06005: inlining of call of procedure 'ADD_IT' was done.
```

Inlining: Guidelines

- 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.
- Be careful about suggesting to inline functions that are deterministic.

ORACLE®

9 - 43

Copyright © 2008, Oracle. All rights reserved.

Inlining: Guidelines

The compiler inlines code automatically, provided that you are using native compilation and have set the PLSQL_OPTIMIZE_LEVEL to 3. If you have set PLSQL_Warnings = 'enable:all', using the SQL*Plus SHOW ERRORS command displays the name of the code that is inlined.

- The PLW-06004 compiler message tells you that a pragma `INLINE('YES')` referring to the named procedure was found. The compiler will, if possible, inline this call.
- The PLW-06005 compiler message tells you the name of the code that is inlined.

Alternatively, you can query the USER/ALL/DBA_ERRORS dictionary view.

Deterministic functions compute the same outputs for the same inputs every time it is invoked and have no side effects. In Oracle Database 11g, the PL/SQL compiler can figure out whether a function is deterministic; it may not find all that truly are, but it finds many of them. It never mistakes a nondeterministic function for a deterministic function.

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/QL code
- Use native compilation for faster PL/SQL execution



Summary

There are several methods that help you tune your PL/SQL application.

When tuning PL/SQL code, consider using the RETURNING clause and/or bulk binds to improve processing. Be aware of conditional statements with an OR clause. Place the fastest processing condition first. There are several data type and constraint issues that can help in tuning an application.

By using native compilation, you can benefit from performance gains for computation-intensive procedural operations.

Practice 9: Overview

This practice covers the following topics:

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



Practice 9: Overview

In this practice, you tune some of the code that you created for the OE application.

- Break a previously built subroutine into smaller executable sections
- Pass collections into subroutines
- Add error handling for BULK INSERT

Use the OE schema for this practice.

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”

Practice 9

In this practice, you measure and examine performance and tuning.

Writing Better Code

1. Open the lab_09_01.sql file and examine the package (the package body is shown below):

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
         VARCHAR2);

    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
         VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;

-- continued on the next page
```

Practice 9 (continued)

```
-- continued from previous page
    IF v_card_info.EXISTS(1) THEN -- cards exist, add more
        i := v_card_info.LAST;
        v_card_info.EXTEND(1);
        v_card_info(i+1) := typ_cr_card(p_card_type,
                                         p_card_no);

        UPDATE customers
            SET credit_cards = v_card_info
            WHERE customer_id = p_cust_id;
    ELSE -- no cards for this customer yet, construct one
        UPDATE customers
            SET credit_cards = typ_cr_card_nst
                (typ_cr_card(p_card_type, p_card_no))
            WHERE customer_id = p_cust_id;
    END IF;
END update_card_info;

PROCEDURE display_card_info
    (p_cust_id NUMBER)
IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
BEGIN
    SELECT credit_cards
        INTO v_card_info
        FROM customers
        WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
        FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
            DBMS_OUTPUT.PUT('Card Type: ' ||
                            v_card_info(idx).card_type || ' ');
            DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
                            v_card_info(idx).card_num );
        END LOOP;
    ELSE
        DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
END display_card_info;
END credit_card_pkg; -- package body
/
```

This code needs to be improved. The following issues exist in the code:

- The local variables use the INTEGER data type.
- The same SELECT statement is run in the two procedures.
- The same IF v_card_info.EXISTS(1) THEN statement is in the two procedures.

Practice 9 (continued)

Using Efficient Data Types

2. To improve the code, make the following modifications:

- a. Change the local INTEGER variables to use a more efficient data type.
- b. Move the duplicated code into a function. The package specification for the modification is:

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    FUNCTION cust_card_info
        (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
        RETURN BOOLEAN;
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
         VARCHAR2 );
    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/
```

- c. Have the function return TRUE if the customer has credit cards. The function should return FALSE if the customer does not have credit cards. Pass an uninitialized nested table into the function. The function places the credit card information into this uninitialized parameter.

3. Test your modified code with the following data:

```
EXECUTE credit_card_pkg.update_card_info -
        (120, 'AM EX', 5555555555)
PL/SQL procedure successfully completed.
```

```
EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
Card Type: DC / Card No: 4444444
Card Type: AM EX / Card No: 5555555555
```

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

```
-- Note: If you did not complete Practice 4, your results
-- will be:
```

```
EXECUTE credit_card_pkg.display_card_info(120)
Card Type: AM EX / Card No: 5555555555
```

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

Practice 9 (continued)

4. You need to modify the UPDATE_CARD_INFO procedure to return information (using the RETURNING clause) about the credit cards being updated. Assume that this information will be used by another application developer in your team, who is writing a graphical reporting utility on customer credit cards.
 - a. Open the lab_09_04_a.sql file. It contains the modified code from the previous question #2.
 - b. Modify the code to use the RETURNING clause to find information about the rows that are affected by the UPDATE statements.
 - c. You can test your modified code with the following procedure (contained in lab_09_04_c.sql):

```
CREATE OR REPLACE PROCEDURE test_credit_update_info
(p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no NUMBER)
IS
    v_card_info typ_cr_card_nst;
BEGIN
    credit_card_pkg.update_card_info
        (p_cust_id, p_card_type, p_card_no, v_card_info);
END test_credit_update_info;
/
```

- d. Test your code with the following statements set in boldface:
EXECUTE test_credit_update_info(125, 'AM EX', 123456789)
PL/SQL procedure successfully completed.

```
SELECT credit_cards FROM customers WHERE customer_id = 125;
CREDIT_CARDS(CARD_TYPE, CARD_NUM)
-----
TYP_CR_CARD_NST(TYP_CR_CARD('AM EX', 123456789))
```

Practice 9 (continued)

Collecting Exception Information

5. In this exercise, you test exception handling with the SAVE EXCEPTIONS clause.

- a. Run the lab_09_05a.sql file to create a test table:

```
CREATE TABLE card_table  
(accepted_cards VARCHAR2(50) NOT NULL);
```

- b. Open the lab_09_05b.sql file and run the contents:

```
DECLARE  
    type typ_cards is table of VARCHAR2(50);  
    v_cards typ_cards := typ_cards  
    ( 'Citigroup Visa', 'Nationscard MasterCard',  
      'Federal American Express', 'Citizens Visa',  
      'International Discoverer', 'United Diners Club' );  
BEGIN  
    v_cards.Delete(3);  
    v_cards.DELETE(6);  
    FORALL j IN v_cards.first..v_cards.last  
        SAVE EXCEPTIONS  
        EXECUTE IMMEDIATE  
        'insert into card_table (accepted_cards) values (  
        :the_card)'  
        USING v_cards(j);  
END;  
/
```

- c. Note the output: _____

Practice 9 (continued)

- d. Open the lab_09_05_d.sql file and run the contents:

```
DECLARE
  type typ_cards is table of VARCHAR2(50);
  v_cards typ_cards := typ_cards
  ( 'Citigroup Visa', 'Nationscard MasterCard',
    'Federal American Express', 'Citizens Visa',
    'International Discoverer', 'United Diners Club' );
  bulk_errors EXCEPTION;
  PRAGMA exception_init (bulk_errors, -24381 );
BEGIN
  v_cards.Delete(3);
  v_cards.DELETE(6);
  FORALL j IN v_cards.first..v_cards.last
    SAVE EXCEPTIONS
    EXECUTE IMMEDIATE
      'insert into card_table (accepted_cards) values (
      :the_card)'
      USING v_cards(j);
EXCEPTION
  WHEN bulk_errors THEN
    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;
  /

```

- e. Note the output:_____

- f. Why is the output different?

Practice 9 (continued)

Timing Performance of SIMPLE_INTEGER and PLS_INTEGER

6. In this exercise, you compare the performance between the PLS_INTEGER and SIMPLE_INTEGER data types with native compilation:

- a. Run the lab_09_06_a.sql file to create a testing procedure that contains conditional compilation:

```
CREATE OR REPLACE PROCEDURE p
IS
    t0      NUMBER := 0;
    t1      NUMBER := 0;

$IF $$Simple $THEN
    SUBTYPE My_Integer_t IS                               SIMPLE_INTEGER;
    My_Integer_t_Name CONSTANT VARCHAR2(30) := 
        'SIMPLE_INTEGER';
$ELSE
    SUBTYPE My_Integer_t IS                               PLS_INTEGER;
    My_Integer_t_Name CONSTANT VARCHAR2(30) := 'PLS_INTEGER';
$END

v00  My_Integer_t := 0;      v01  My_Integer_t := 0;
v02  My_Integer_t := 0;      v03  My_Integer_t := 0;
v04  My_Integer_t := 0;      v05  My_Integer_t := 0;

two   CONSTANT My_Integer_t := 2;
lmt   CONSTANT My_Integer_t := 100000000;

BEGIN
    t0 := DBMS_UTILITY.GET_CPU_TIME();
    WHILE v01 < lmt LOOP
        v00 := v00 + Two;
        v01 := v01 + Two;
        v02 := v02 + Two;
        v03 := v03 + Two;
        v04 := v04 + Two;
        v05 := v05 + Two;
    END LOOP;

    IF v01 <> lmt OR v01 IS NULL THEN
        RAISE Program_Error;
    END IF;

    t1 := DBMS_UTILITY.GET_CPU_TIME();
    DBMS_OUTPUT.PUT_LINE(
        RPAD(LOWER($$PLSQL_Code_Type), 15) ||
        RPAD(LOWER(My_Integer_t_Name), 15) ||
        TO_CHAR((t1-t0), '9999') || ' centiseconds');
END p;
```

Practice 9 (continued)

- b. Open the lab_09_06_b.sql file and run the contents:

```
ALTER PROCEDURE p COMPILE  
PLSQL_Code_Type = NATIVE PLSQL_CCFlags = 'simple:true'  
REUSE SETTINGS;
```

```
EXECUTE p()
```

```
ALTER PROCEDURE p COMPILE  
PLSQL_Code_Type = native PLSQL_CCFlags = 'simple:false'  
REUSE SETTINGS;
```

```
EXECUTE p()
```

- c. Note the output: _____

- d. Explain the output.

Improving Performance with Caching

10

ORACLE

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

- 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



ORACLE

10 - 2

Copyright © 2008, Oracle. All rights reserved.

Objectives

In this lesson, you learn about the Oracle Database 11g caching techniques that can improve performance. You examine the improvement on the performance of queries by caching the results of a query in memory, and then using the cached results in future executions of the query or query fragments. The cached results reside in the result cache memory portion of the shared global area (SGA).

The PL/SQL cross-section function result caching mechanism provides applications with a language-supported and system-managed means for storing the results of PL/SQL functions in an SGA, which is available to every session that runs the application.

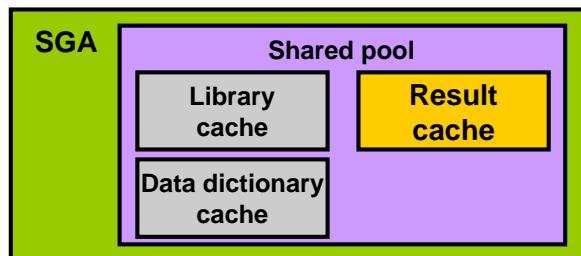
Lesson Agenda

- Improving memory usage by caching SQL result sets
 - Enabling Query Result Cache
 - Using the DBMS_RESULT_CACHE package
- Implementing SQL Query Result Cache
 - Writing queries that use the result cache hint
- Using PL/SQL Function Result Cache
 - Setting up PL/SQL functions to use PL/SQL result caching
 - Implementing PL/SQL Function Result Caching

ORACLE

What Is Result Caching?

- Result Cache allows SQL query and PL/SQL function results to be stored in cache memory.
- 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.
- Cached query results become invalid when the database data accessed by the query is modified.



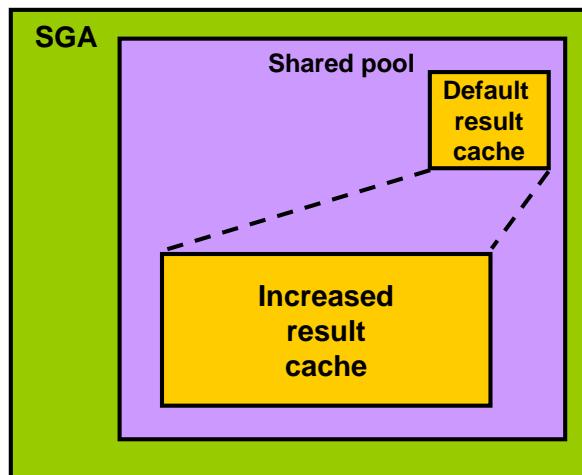
ORACLE

What Is Result Caching?

The new SQL Query Result Cache enables explicit caching of queries and query fragments in an area of the shared pool called Result Cache Memory. When a query is executed, the result cache is built up and the result is returned. The database can then use the cached results for subsequent query executions, thereby resulting in faster response times. Cached query results become invalid when the data in the database objects being accessed by the query are modified.

Increasing Result Cache Memory Size

- You can increase the small, default result cache memory size by using the RESULT_CACHE_MAX_SIZE initialization parameter.



ORACLE

10 - 5

Copyright © 2008, Oracle. All rights reserved.

Increasing Result Cache Memory Size

By default, on database startup, Oracle allocates memory to the result cache in the shared pool. The memory size allocated depends on the memory size of the shared pool as well as the memory management system.

- When using the MEMORY_TARGET initialization parameter to specify the memory allocation, Oracle allocates 0.25% of the memory target to the result cache.
- When you set the size of the shared pool using the SGA_TARGET initialization parameter, Oracle allocates 0.5% of the SGA target to the result cache.
- If you specify the size of the shared pool using the SHARED_POOL_SIZE initialization parameter, Oracle allocates 1% of the shared pool size to the result cache.

Note: Oracle will not allocate more than 75% of the shared pool to the result cache.

Use the RESULT_CACHE_MAX_RESULT initialization parameter to specify the maximum percentage of result cache memory that can be used by any single result. The default value is 5%, but you can specify any percent value between 1 and 100.

Setting Result_Cache_Max_Size

- Set Result_Cache_Max_Size from the command line or in an initialization file by a DBA.
- The cache size is dynamic and can be changed either permanently or until the instance is restarted.

```
SQL> ALTER SYSTEM SET result_cache_max_size = 2M SCOPE =
      MEMORY;

System altered.

SQL> SELECT name, value
  2  FROM  v$parameter
  3 WHERE  name = 'result_cache_max_size';

NAME                                VALUE
-----  -----
result_cache_max_size                2097152

1 row selected.
```

ORACLE

10 - 6

Copyright © 2008, Oracle. All rights reserved.

Setting Result_Cache_Max_Size

By default, the server-side Result Cache is configured to use a very small portion of the shared pool. You can manually set the result cache memory size by using the RESULT_CACHE_MAX_SIZE initialization parameter. Setting RESULT_CACHE_MAX_SIZE to 0 during database startup disables the server-side Result Cache. RESULT_CACHE_MAX_SIZE cannot be dynamically changed if the value is set to 0 during database startup in the spfile or the init.ora file.

Enabling Query Result Cache

- Use the RESULT_CACHE_MODE initialization parameter in the database initialization parameter file.
- 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.

ORACLE

10 - 7

Copyright © 2008, Oracle. All rights reserved.

Enabling Query Result Cache

You can enable Query Result Cache at the database level by using the RESULT_CACHE_MODE initialization parameter in the database initialization parameter file. The same parameter can also be used at the session level by using the ALTER SESSION command. RESULT_CACHE_MODE can be set to:

- **MANUAL (default)**: You must add the RESULT_CACHE hint to your queries for the results to be cached or to be served out of the cache. The RESULT_CACHE hint can also be added in subqueries and inline views.
- **FORCE**: Results are always stored in the Result Cache Memory, if possible.

The use of the SQL Query Result Cache introduces the ResultCache operator in the query execution plan.

Using the DBMS_RESULT_CACHE Package

- The DBMS_RESULT_CACHE package provides an interface for a DBA to manage memory allocation for SQL query result cache and the PL/SQL function result cache.

```
execute dbms_result_cache.memory_report
```

```
Result Cache Memory Report
[Parameters]
Block Size      = 1K bytes
Maximum Cache Size = 1056K bytes (1056 blocks)
Maximum Result Size = 52K bytes (52 blocks)
[Memory]
Total Memory = 5140 bytes [0.003% of the Shared Pool]
... Fixed Memory = 5140 bytes [0.003% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]
```

ORACLE

10 - 8

Copyright © 2008, Oracle. All rights reserved.

Using the DBMS_RESULT_CACHE Package

You can use the DBMS_RESULT_CACHE package to perform various operations such as bypassing the cache, retrieving statistics on the cache memory usage, and flushing the cache. For example, to view the memory allocation statistics, use `dbms_result_cache.memory_report`. The output of this command is similar to the following:

```
Result Cache Memory Report
[Parameters]
Block Size      = 1K bytes
Maximum Cache Size = 1056K bytes (1056 blocks)
Maximum Result Size = 52K bytes (52 blocks)
[Memory]
Total Memory = 5140 bytes [0.003% of the Shared Pool]
... Fixed Memory = 5140 bytes [0.003% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]
```

Lesson Agenda

- Improving memory usage by caching SQL result sets
 - Enabling Query Result Cache
 - Using the DBMS_RESULT_CACHE package
- Implementing SQL Query Result Cache
 - Writing queries that use the result cache hint
- Using PL/SQL Function Result Cache
 - Setting up PL/SQL functions to use PL/SQL result caching
 - Implementing PL/SQL Function Result Caching

ORACLE

10 - 9

Copyright © 2008, Oracle. All rights reserved.

Lesson Agenda

An example of setting the result cache hint is provided on the next few pages.

SQL Query Result Cache

- Definition:
 - Cache the results of the current query or query fragment in memory, and then use the cached results in future executions of the query or query fragments.
 - Cached results reside in the result cache memory portion of the SGA.
- Benefits:
 - Improved performance



ORACLE

10 - 10

Copyright © 2008, Oracle. All rights reserved.

SQL Query Result Cache

You can improve the performance of your queries by caching the results of a query in memory, and then using the cached results in future executions of the query or query fragments. The cached results reside in the result cache memory portion of the SGA. This feature is designed to speed up query execution on systems with large memories.

SQL Query Result Cache

- Scenario:

- You need to find the greatest average value of credit limit grouped by state over the whole population.
- The query results in a large number of rows being analyzed to yield a few 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;
```



Usage

SQL result caching is useful when your queries need to analyze a large number of rows to return a small number of rows or a single row.

Two new optimizer hints are available to turn on and turn off SQL result caching:

```
/*+ result_cache */
/*+ no_result_cache */
```

These hints enable you to override the settings of the `RESULT_CACHE_MODE` initialization parameter.

You can execute `DBMS_RESULT_CACHE.MEMORY_REPORT` to produce a memory usage report of the result cache.

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.
connect / as sysdba
SET ECHO ON
SET FEEDBACK 1
SET NUMWIDTH 10
SET LINESIZE 150
SET TRIMSPPOOL ON
SET TAB OFF
SET PAGESIZE 1000
SET SERVEROUTPUT ON

execute dbms_result_cache.flush
alter system flush shared_pool
/
execute dbms_result_cache.memory_report
```



Clearing the Shared Pool and Result Cache

To understand the use of Query Result Cache, ensure that you are using clean, new data. The slide example uses SQL*Plus. Connect to the database as SYS.

Clear the shared pool and the result cache by executing the code as shown in the slide.

Examining the Memory Cache

```
--- flush.sql
--- Start with a clean slate. Flush the cache and shared pool.
--- Verify that memory was released.
connect / as sysdba
SET ECHO ON
SET FEEDBACK 1
SET NUMWIDTH 10
SET LINESIZE 150
SET TRIMSPPOOL ON
SET TAB OFF
SET PAGESIZE 1000
SET SERVEROUTPUT ON

execute dbms_result_cache.flush
alter system flush shared_pool
/
execute dbms_result_cache.memory_report
```

```
Result Cache Memory Report
[Parameters]
Block Size      = 1K bytes
Maximum Cache Size = 1056K bytes (1056 blocks)
Maximum Result Size = 52K bytes (52 blocks)
[Memory]
Total Memory = 5140 bytes [0.003% of the Shared Pool]
... Fixed Memory = 5140 bytes [0.003% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]

PL/SQL procedure successfully completed.
```

ORACLE

10 - 13

Copyright © 2008, Oracle. All rights reserved.

Examining the Memory Cache

Examine the memory cache by executing the code as shown in the slide. Cache memory is 0 bytes, because nothing has yet been cached.

Examining the Execution Plan for a Query

```
--- plan_query1.sql
connect oe

--- Generate the execution plan.
--- (The query name Q1 is optional)
explain plan for
select /*+ result_cache q_name(Q1) */ * from orders;

set echo off
--- Display the execution plan. Verify that the query result
--- is placed in the Result Cache.
@$ORACLE_HOME/rdbms/admin/utlxpls
```

```
PLAN_TABLE_OUTPUT
-----
Plan hash value: 1275100350

-----  
| Id  | Operation          | Name           | Rows | Bytes | Cost (|  
| %CPU | Time      |               |       |       |       |  
-----  
| 0   | SELECT STATEMENT |                | 105  | 3885  |    3 |  
| (0) | 00:00:01          |               |       |       |       |  
| 1   | RESULT CACHE     | 5f0x59rau680n60zdua0mztzku |       |       |       |  
| 2   | TABLE ACCESS FULL | ORDERS        | 105  | 3885  |    3 |  
| (0) | 00:00:01          |               |       |       |       |  
-----  
Result Cache Information (identified by operation id):  
-----  
1  column count=8, dependencies=(OE.ORDERS),  
   name="select /*+ result_cache q_name(Q1) */ * from orders"  
14 rows selected.
```

ORACLE

10 - 14

Copyright © 2008, Oracle. All rights reserved.

Examining the Execution Plan for a Query

You examine the execution plan for two queries, and then execute both queries. After executing both queries, you view the memory allocation and usage statistics.

First, execute the code as shown in the slide and examine the execution plan for the first query. The query uses the RESULT_CACHE optimizer hint.

Examining Another Execution Plan

```
--- plan_query2.sql
set echo on
connect oe

--- Generate the execution
--- (The query name Q2 is
explain plan for
select c.customer_id, o.order_id
from (select /*+ result_cache */ customer_id, count(*)
       from orders
      group by customer_id
     where o.customer_id = c.customer_id) o
      ,oe
     where o.customer_id = c.customer_id
set echo off

--- Display the execution plan
--- using the code in
--- ORACLE_HOME/rdbms/admin/xplan.sql
set markup html preformat on

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

PLAN_TABLE_OUTPUT						

Plan hash value: 2892511806						

Id (%CPU)	Operation Time	Name		Rows	Bytes	Cost
0 (0)	SELECT STATEMENT 00:00:01			47	1692	1
1 (0)	NESTED LOOPS 00:00:01			47	1692	1
* 2 (0)	VIEW 00:00:01			47	1316	1
3 (0)	RESULT CACHE 00:00:01	2za8xzq06lxnu8q8m2r14d1s23				
4 (0)	HASH GROUP BY 00:00:01			47	188	1
5 (0)	INDEX FULL SCAN ORD_CUSTOMER_IX 00:00:01		105	420	1	
* 6 (0)	INDEX UNIQUE SCAN CUSTOMERS_PK 00:00:01		1	8	0	

Predicate Information (identified by operation id):					

2 - filter("O"."CUSTOMER_ID">>0)					
6 - access("O"."CUSTOMER_ID"="C"."CUSTOMER_ID") filter("C"."CUSTOMER_ID">>0)					
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"					

ORACLE

10 - 15

Copyright © 2008, Oracle. All rights reserved.

Examining Another Execution Plan

Execute the code as shown in the slide and examine the execution plan for the second query. This query also uses the RESULT_CACHE optimizer hint.

Executing Both Queries

```
--- query3.sql
--- Cache result of both queries, then use the cached result.

connect oe
set echo on

select /*+ result_cache q_name(Q1) */
/

select c.customer_id, o.ord_count
  from (select /*+ result_cache q_name(
            customer_id, count(*) ord_count
          from orders
        group by customer_id) o, customers c
       where o.customer_id = c.customer_id
      /
      set echo off

SQL> select c.customer_id, o.ord_count
  2  from (select /*+ result_cache q_name(Q3) */
  3    customer_id, count(*) ord_count
  4    from orders
  5    group by customer_id) o, customers c
  6   where o.customer_id = c.customer_id;

CUSTOMER_ID  ORD_COUNT
-----  -----
      123          1
      151          1
      152          1
      153          1
      167          1
      168          1
      116          4
      120          1
      121          1
      144          5
```

ORACLE

10 - 16

Copyright © 2008, Oracle. All rights reserved.

Executing Both Queries

You have examined the execution plan for both queries. Now execute both queries by executing the code as shown in the slide.

Viewing Cache Results Created

```
SQL> connect / as sysdba
Connected.
SQL>
SQL> col name format a55
SQL>
SQL> select *
2 from v$result_cache_statistics
3 /

```

ID	NAME	VALUE
1	Block Size (Bytes)	1024
2	Block Count Maximum	1056
3	Block Count Current	32
4	Result Size Maximum (Blocks)	52
5	Create Count Success	2
6	Create Count Failure	0
7	Find Count	0
8	Invalidation Count	0
9	Delete Count Invalid	0
10	Delete Count Valid	0

```
10 rows selected.

SQL>
```

Number of cache results
successfully created

ORACLE

10 - 17

Copyright © 2008, Oracle. All rights reserved.

Viewing Cache Results Created

The next step is to run a query against V\$RESULT_CACHE_STATISTICS to view the memory allocation and usage statistics. You can view the memory allocation by executing the code as shown in the slide.

Note that the CREATE COUNT SUCCESS column has a value of 2, which is the number of cache results that were successfully created (one for each query statement).

In the next steps, you re-execute the queries and view the cache results found.

Re-Executing Both Queries

```
--- query3.sql
--- Cache result of both queries, then use the cached result.

connect oe
set echo on

select /*+ result_cache q_name(Q1) */
/

select c.customer_id, o.ord_count
  from (select /*+ result_cache q_name(
            customer_id, count(*) ord_count
          from orders
        group by customer_id) o, customers c
       where o.customer_id = c.customer_id
      /
      set echo off

      . . .
      2453 04-OCT-99 06.53.34.362632 PM
      direct           116          0          129          153
      2456 07-NOV-98 06.53.25.989889 PM
      direct           117          0          3878.4         163
      2457 31-OCT-99 08.22.16.162632 PM
      direct           118          5          21586.2         159
      105 rows selected.

Elapsed: 00:00:00.02

SQL> select c.customer_id, o.ord_count
  2   from (select /*+ result_cache q_name(Q3) */
  3           customer_id, count(*) ord_count
  4         from orders
  5       group by customer_id) o, customers c
  6     where o.customer_id = c.customer_id;

CUSTOMER_ID  ORD_COUNT
-----  -----
      123          1
      151          1
      152          1
      153          1
      167          1
      168          1
      116          4
      120          1
      121          1
      144          5
      . . .
```

ORACLE

10 - 18

Copyright © 2008, Oracle. All rights reserved.

Re-Executing Both Queries

Re-execute both the queries as shown in the slide.

Note that the query runs faster after caching, with an elapsed time of 00:00:00.02, compared to the previous time of 00:00:06.00.

Viewing Cache Results Found

```
SQL> connect / as sysdba
Connected.
SQL>
SQL> col name format a55
SQL>
SQL> select *
2  from v$result_cache_statistics
3 /

```

ID	NAME	VALUE
1	Block Size (Bytes)	1024
2	Block Count Maximum	1056
3	Block Count Current	32
4	Result Size Maximum (Blocks)	52
5	Create Count Success	2
6	Create Count Failure	0
7	Find Count	2
8	Invalidation Count	0
9	Delete Count Invalid	0
10	Delete Count Valid	0

10 rows selected.

**Number of cache results
successfully found**

ORACLE

Viewing Cache Results Found

Query V\$RESULT_CACHE_STATISTICS again to view memory allocation and usage statistics. Do this by again executing the code as shown in the slide.

Note that the FIND COUNT column now has a value of 2. This is the number of cache results that were successfully found (one for each query statement).

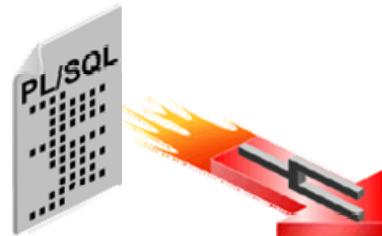
Lesson Agenda

- Improving memory usage by caching SQL result sets
 - Enabling Query Result Cache
 - Using the DBMS_RESULT_CACHE package
- Implementing SQL Query Result Cache
 - Writing queries that use the result cache hint
- Using PL/SQL Function Result Cache
 - Setting up PL/SQL functions to use PL/SQL result caching
 - Implementing PL/SQL Function Result Caching

ORACLE

PL/SQL Function Result Cache

- Definition:
 - Enables data that is stored in cache to be shared across sessions
 - Stores the function result cache in an SGA, making it available to any session that runs your application
- Benefits:
 - Improved performance
 - Improved scalability



ORACLE

10 - 21

Copyright © 2008, Oracle. All rights reserved.

PL/SQL Function Result Cache

Starting in Oracle Database 11g, you can use the PL/SQL cross-section function result caching mechanism. This caching mechanism provides you with a language-supported and system-managed means for storing the results of PL/SQL functions in an SGA, which is available to every session that runs your application. The caching mechanism is both efficient and easy to use, and it relieves you of the burden of designing and developing your own caches and cache-management policies.

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.



ORACLE

10 - 22

Copyright © 2008, Oracle. All rights reserved.

Marking PL/SQL Function Results to Be Cached

To enable result caching for a function, use the RESULT_CACHE clause in your PL/SQL function. Using this clause results in the following:

- If a result-cached function is called, the system checks the cache.
- If the cache contains the result from a previous call to the function with the same parameter values, the system returns the cached result to the caller and does not re-execute the function body.
- If the cache does not contain the result, the system executes the function body and adds the result (for these parameter values) to the cache before returning control to the caller.

The cache can accumulate many results—one result for every unique combination of parameter values with which each result-cached function was called. If the system needs more memory, it ages out (deletes) one or more cached results.

You can specify the database objects that are used to compute a cached result, so that if any of them is updated, the cached result becomes invalid and must be recomputed.

The best candidates for result caching are functions that are called frequently but depend on information that changes infrequently or never.

Clearing the Shared Pool and Result Cache

```
--- flush.sql
--- Start with a clean slate. Flush the cache and shared pool.
--- Verify that memory was released.
connect / as sysdba
SET ECHO ON
SET FEEDBACK 1
SET NUMWIDTH 10
SET LINESIZE 150
SET TRIMSPPOOL ON
SET TAB OFF
SET PAGESIZE 1000
SET SERVEROUTPUT ON

execute dbms_result_cache.flush
alter system flush shared_pool
/
execute dbms_result_cache.memory_report
```

```
Result Cache Memory Report
[Parameters]
Block Size      = 1K bytes
Maximum Cache Size = 1056K bytes (1056 blocks)
Maximum Result Size = 52K bytes (52 blocks)
[Memory]
Total Memory = 5140 bytes [0.003% of the Shared Pool]
... Fixed Memory = 5140 bytes [0.003% of the Shared Pool]
... Dynamic Memory = 0 bytes [0.000% of the Shared Pool]
```

ORACLE

10 - 23

Copyright © 2008, Oracle. All rights reserved.

Clearing the Shared Pool and Result Cache

To understand the use of PL/SQL Function Result Cache, clear the shared pool and the result cache again by performing the steps as shown in the slide.

Lesson Agenda

- Improving memory usage by caching SQL result sets
 - Enabling Query Result Cache
 - Using the DBMS_RESULT_CACHE package
- Implementing SQL Query Result Cache
 - Writing queries that use the result cache hint
- Using PL/SQL Function Result Cache
 - Setting up PL/SQL functions to use PL/SQL result caching
 - Implementing PL/SQL Function Result Caching

ORACLE

Creating a PL/SQL Function 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 relies

ORACLE

10 - 25

Copyright © 2008, Oracle. All rights reserved.

Creating a PL/SQL Function Using the RESULT_CACHE Clause

When writing code for the PL/SQL result cache option, you need to:

- Include the RESULT_CACHE option in the function declaration section of a package
- Include the RESULT_CACHE option in the function definition
- Optionally include the RELIES_ON clause to specify tables or views on which the function results depend

In the example shown in the slide, the ORD_COUNT function has result caching enabled through the RESULT_CACHE option in the function declaration. In this example, the RELIES_ON clause is used to identify the ORDERS table on which the function results depend.

You can also run DBMS_RESULT_CACHE.MEMORY_REPORT to view the result cache memory results.

For more information about result caching in Oracle Database 11g, review the Oracle by Example tutorial, which is available at:

http://stcontent.oracle.com/content/dav/oracle/Libraries/ST%20Curriculum/ST%20Curriculum-Public/Courses/OBE/11gr1_db/manage/res_cache/results_cache.htm

Lesson Agenda

- Improving memory usage by caching SQL result sets
 - Enabling Query Result Cache
 - Using the DBMS_RESULT_CACHE package
- Implementing SQL Query Result Cache
 - Writing queries that use the result cache hint
- Using PL/SQL Function Result Cache
 - Setting up PL/SQL functions to use PL/SQL result caching
 - Implementing PL/SQL Function Result Caching

ORACLE

Calling the PL/SQL Function Inside a Query

```
SQL> connect oe
Connected.
SQL> select cust_last_name, ord_count(customer_id)
      no_of_orders
  2  from customers
  3  where cust_last_name = 'MacGraw';

CUST_LAST_NAME      NO_OF_ORDERS
-----  -----
MacGraw                      4

1 row selected.
```

ORACLE

10 - 27

Copyright © 2008, Oracle. All rights reserved.

Calling the PL/SQL Function Inside a Query

Call the ORD_COUNT PL/SQL function inside a query by performing the steps as shown in the slide.

Verifying Memory Allocation

```
SQL> SET ECHO ON
SQL> SET FEEDBACK 1
SQL> SET NUMWIDTH 10
SQL> SET LINESIZE 150
SQL> SET TRIMSPPOOL ON
SQL> SET TAB OFF
SQL> SET PAGESIZE 1000
SQL>
SQL> connect / as sysdba
Connected.
SQL>
SQL> --- Establish the cache content
SQL> set serveroutput on
SQL> execute dbms_result_cache.memory_report
R e s u l t   C a c h e   M e m o r y   R e p o r t
[Parameters]
Block Size      = 1K bytes
Maximum Cache Size = 1056K bytes (1056 blocks)
Maximum Result Size = 52K bytes (52 blocks)
[Memory]
Total Memory = 103536 bytes [0.055% of the Shared Pool]
... Fixed Memory = 5140 bytes [0.003% of the Shared Pool]
... Dynamic Memory = 98396 bytes [0.052% of the Shared Pool]
..... Overhead = 65628 bytes
..... Cache Memory = 32K bytes (32 blocks)
..... Unused Memory = 29 blocks
..... Used Memory = 3 blocks
..... Dependencies = 2 blocks (2 count)
..... Results = 1 blocks
..... PLSQL      = 1 blocks (1 count)
PL/SQL procedure successfully completed.
```

ORACLE

Verifying Memory Allocation

To verify that memory was allocated, execute `dbms_result_cache.memory_report` as shown in the slide, to view the memory allocation statistics.

Viewing Cache Results Created

```
SQL> connect / as sysdba
Connected.
SQL>
SQL> col name format a55
SQL>
SQL> select *
  2  from v$result_cache_statistics
  3  /


| ID | NAME                         | VALUE |
|----|------------------------------|-------|
| 1  | Block Size (Bytes)           | 1024  |
| 2  | Block Count Maximum          | 1056  |
| 3  | Block Count Current          | 32    |
| 4  | Result Size Maximum (Blocks) | 52    |
| 5  | Create Count Success         | 1     |
| 6  | Create Count Failure         | 0     |
| 7  | Find Count                   | 0     |
| 8  | Invalidation Count           | 0     |
| 9  | Delete Count Invalid         | 0     |
| 10 | Delete Count Valid           | 0     |



10 rows selected.



Elapsed: 00:00:05.00


SQL>
```

ORACLE

10 - 29

Copyright © 2008, Oracle. All rights reserved.

Viewing Cache Results Created

The next step is to query V\$RESULT_CACHE_STATISTICS to view the memory allocation and usage statistics. Do this by performing the steps as shown in the slide.

Note that the CREATE COUNT SUCCESS column has a value of 1. This is the number of cache results that were successfully created (one for each query statement).

Calling the PL/SQL Function Again

```
SQL> connect oe
Connected.
SQL> select cust_last_name, ord_count(customer_id)
      no_of_orders
  2  from customers
  3  where cust_last_name = 'MacGraw';

CUST_LAST_NAME      NO_OF_ORDERS
-----  -----
MacGraw                      4

1 row selected.
```



Calling the PL/SQL Function Again

Run the query again, as shown in the slide.

Viewing Cache Results Found

```
SQL> connect / as sysdba
Connected.
SQL>
SQL> col name format a55
SQL>
SQL> select *
  2  from v$result_cache_statistics
  3 /
-----  
 ID NAME          VALUE  
-----  
 1 Block Size (Bytes)    1024  
 2 Block Count Maximum  1056  
 3 Block Count Current  32  
 4 Result Size Maximum (Blocks) 52  
 5 Create Count Success 1  
 6 Create Count Failure 0  
 7 Find Count           1  
 8 Invalidation Count   0  
 9 Delete Count Invalid 0  
10 Delete Count Valid   0  

10 rows selected.  
  
Elapsed: 00:00:00.01  
  
SQL>
```

ORACLE

Viewing Cache Results Found

Query V\$RESULT_CACHE_STATISTICS again to view the memory allocation and usage statistics. Do this by again performing the steps as shown in the slide.

Note that the FIND COUNT column now has a value of 1. This is the number of cache results that were successfully found (one for each query statement).

Note also that the elapsed time for the same query has reduced from 00:00:05.00 to 00:00:00.01.

Confirming That the Cached Result Was Used

```
SQL> connect / as sysdba
Connected.
SQL>
SQL> col name format a55
SQL>
SQL> select type, namespace,status, scan_count,name
  2  from v$result_cache_objects
  3 /

```

TYPE	NAMES	STATUS	SCAN_COUNT	NAME
Dependency		Published	0	OE.ORD_COUNT
Dependency		Published	0	OE.ORDERS
Result	PLSQL	Published	1	

```
"OE"."ORD_COUNT":::8 ."ORD_COUNT"#fac892c7867b54c6 #1
3 rows selected.

SQL>
```

ORACLE

10 - 32

Copyright © 2008, Oracle. All rights reserved.

Confirming That the Cached Result Was Used

Query V\$RESULT_CACHE_OBJECTS to confirm that the cached result was used. Do this by performing the steps as shown in the slide.

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



Summary

In this lesson, you saw the Oracle Database 11g caching techniques that can improve performance.

Practice 10 Overview: Examining SQL and PL/SQL Result Caching

This practice covers the following topics:

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



Practice 10: Overview

In this practice, you implement SQL query result caching and PL/SQL result function caching. You run scripts to measure the cache memory values, manipulate queries and functions to turn caching on and off, and then examine cache statistics. This practice uses the OE schema.

Practice 10

In this practice, you examine the Explain Plan for a query, add the RESULT_CACHE hint to the query, and reexamine the Explain Plan results. You also execute some code and modify the code so that PL/SQL result caching is turned on.

Examining SQL and PL/SQL Result Caching

1. Use SQL Developer to connect to the OE schema. Examine the Explain Plan for this query, which is found in the lab_10_01.sql lab file. You can see the Explain Plan by clicking the Execute Explain Plan button on the toolbar in the Code Editor window.

```
SELECT count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
  FROM inventories natural join product_information
 GROUP BY product_id, product_name;
```

Examine the Explain Plan results.

2. Add the RESULT_CACHE hint to the query and reexamine the Explain Plan results.

```
SELECT /*+ result_cache */
       count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
  FROM inventories natural join product_information
 GROUP BY product_id, product_name;
```

Examine the Explain Plan results, compared to the previous results.

3. The following code is used to generate a list of warehouse names for pick lists in applications. The WAREHOUSES table is fairly stable and is not modified often.

Click the Run Script button to compile this code (You can use the lab_10_03.sql file):

Practice 10 (continued)

Examining SQL and PL/SQL Result Caching (continued)

```
CREATE OR REPLACE TYPE list_typ IS TABLE OF VARCHAR2(35);
/

CREATE OR REPLACE FUNCTION get_warehouse_names
RETURN list_typ
IS
    v_count BINARY_INTEGER;
    v_wh_names list_typ;
BEGIN
    SELECT count(*)
        INTO v_count
        FROM warehouses;
    FOR i in 1..v_count LOOP
        SELECT warehouse_name
        INTO v_wh_names(i)
        FROM warehouses;
    END LOOP;
    RETURN v_wh_names;
END get_warehouse_names;
```

4. Because the function is called frequently, and because the content of the data returned does not change frequently, this code is a good candidate for PL/SQL result caching.

Modify the code so that PL/SQL result caching is turned on. Click the Run Script button to compile this code again.

Analyzing PL/SQL Code

11

ORACLE®

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

- Use the supplied packages and dictionary views to find coding information
- Determine identifier types and usages with PL/Scope
- Use the DBMS_METADATA package to obtain metadata from the data dictionary as XML or creation DDL that can be used to re-create the objects



Objectives

In this lesson, you learn how to write PL/SQL routines that analyze the PL/SQL applications. You learn about the dictionary views and the packages that you can use to find information within your code and to generate information about your code.

Lesson Agenda

- Running reports on source code
- Determining identifier types and usages
- Using DBMS_METADATA to retrieve object definitions



Finding Coding Information

- Use the dictionary views:

- ALL_ARGUMENTS
- ALL_OBJECTS
- ALL_SOURCE
- ALL PROCEDURES
- ALL_DEPENDENCIES



- Use the supplied packages:

- dbms_describe
- dbms_utility



Finding Information About Your PL/SQL Code

The Oracle dictionary views store information about your compiled PL/SQL code. You can write SQL statements against the views to find information about your code.

Dictionary View	Description
ALL_SOURCE	Includes the lines of source code for all programs that you modify
ALL_ARGUMENTS	Includes information about the parameters for the procedures and functions that you can call
ALL PROCEDURES	Contains the list of procedures and functions that you can execute
ALL_DEPENDENCIES	Is one of the several views that give you information about the dependencies between database objects

You can also use the Oracle-supplied DBMS_DESCRIBE package to obtain information about a PL/SQL object. The package contains the DESCRIBE PROCEDURE procedure, which provides a brief description of a PL/SQL stored procedure. It takes the name of a stored procedure and returns information about each parameter of that procedure.

You can use the DBMS UTILITY supplied package to follow a call stack and an exception stack.

Finding Coding Information

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;

NAME          LINE TEXT
-----
CUST_ADDRESS_TYP    6      , country_id      CHAR(2)
...
```

ORACLE®

11 - 5

Copyright © 2008, Oracle. All rights reserved.

Finding Data Types

You may want to find all occurrences of the CHAR data type. The CHAR data type is fixed in length and can cause false negatives on comparisons with VARCHAR2 strings. By finding the CHAR data type, you can modify the object, if appropriate, and change it to VARCHAR2.

Finding Coding Information

Create a package with various queries that you can easily call:

```
CREATE OR REPLACE PACKAGE query_code_pkg
AUTHID CURRENT_USER
IS
  PROCEDURE find_text_in_code (str IN VARCHAR2);
  PROCEDURE encap_compliance ;
END query_code_pkg;
/
```



Creating a Package to Query Code

A better idea is to create a package to hold various queries that you can easily call. The QUERY_CODE_PKG will hold two validation procedures:

The FIND_TEXT_IN_CODE procedure displays all programs with a specified character string. It queries USER_SOURCE to find occurrences of any text string. The text string is passed as a parameter. For efficiency, the BULK COLLECT statement is used to retrieve all matching rows into the collection variable.

The ENCAP_COMPLIANCE procedure identifies those programs that reference a table directly. This procedure queries the ALL_DEPENDENCIES view to find the PL/SQL code objects that directly reference a table or a view.

You can also include a procedure to validate a set of standards for exception handling.

Creating a Package to Query Code (continued)

QUERY_CODE_PKG Code

```
CREATE OR REPLACE PACKAGE BODY query_code_pkg IS
    PROCEDURE find_text_in_code (str IN VARCHAR2)
    IS
        TYPE info_rt IS RECORD (NAME user_source.NAME%TYPE,
                               text user_source.text%TYPE );
        TYPE info_aat IS TABLE OF info_rt INDEX BY PLS_INTEGER;
        info_aa info_aat;
    BEGIN
        SELECT NAME || '-' || line, text
        BULK COLLECT INTO info_aa FROM user_source
        WHERE UPPER (text) LIKE '%' || UPPER (str) || '%'
        AND NAME != 'VALSTD' AND NAME != 'ERRNUMS';
        DBMS_OUTPUT.PUT_LINE ('Checking for presence of ' ||
                              str || ':');
        FOR indx IN info_aa.FIRST .. info_aa.LAST LOOP
            DBMS_OUTPUT.PUT_LINE (
                info_aa (indx).NAME || ',' || info_aa (indx).text);
        END LOOP;
    END find_text_in_code;

    PROCEDURE encaps_compliance IS
        SUBTYPE qualified_name_t IS VARCHAR2 (200);
        TYPE refby_rt IS RECORD (NAME qualified_name_t,
                               referenced_by qualified_name_t );
        TYPE refby_aat IS TABLE OF refby_rt INDEX BY PLS_INTEGER;
        refby_aa refby_aat;
    BEGIN
        SELECT owner || '.' || NAME refs_table
        , referenced_owner || '.' || referenced_name
        AS table_referenced
        BULK COLLECT INTO refby_aa
        FROM all_dependencies
        WHERE owner = USER
        AND TYPE IN ('PACKAGE', 'PACKAGE BODY',
                     'PROCEDURE', 'FUNCTION')
        AND referenced_type IN ('TABLE', 'VIEW')
        AND referenced_owner NOT IN ('SYS', 'SYSTEM')
        ORDER BY owner, NAME, referenced_owner, referenced_name;
        DBMS_OUTPUT.PUT_LINE ('Programs that reference
                                tables or views');
        FOR indx IN refby_aa.FIRST .. refby_aa.LAST LOOP
            DBMS_OUTPUT.PUT_LINE (refby_aa (indx).NAME || ',' ||
                                  refby_aa (indx).referenced_by);
        END LOOP;
    END encaps_compliance;
END query_code_pkg;
/
```

Finding Coding Information

```
EXECUTE query_code_pkg.encap_compliance  
Programs that reference tables or views  
OE.CREDIT_CARD_PKG,OE.CUSTOMERS  
OE.LOAD_PRODUCT_IMAGE,OE.PRODUCT_INFORMATION  
OE.SET_VIDEO,OE.CUSTOMERS  
OE.WRITE_LOB,OE.PRODUCT_DESCRIPTIONS  
...  
  
PL/SQL procedure successfully completed.
```

1

```
EXECUTE query_code_pkg.find_text_in_code('customers')  
  
Checking for presence of customers:  
CREDIT_CARD_PKG-12,      FROM customers  
CREDIT_CARD_PKG-36,      UPDATE customers  
CREDIT_CARD_PKG-41,      UPDATE customers  
SET_VIDEO-6,             SELECT cust_first_name FROM customers  
SET_VIDEO-13,            UPDATE customers SET video = file_ptr  
...  
  
PL/SQL procedure successfully completed.
```

2

11 - 8

Copyright © 2008, Oracle. All rights reserved.

ORACLE®

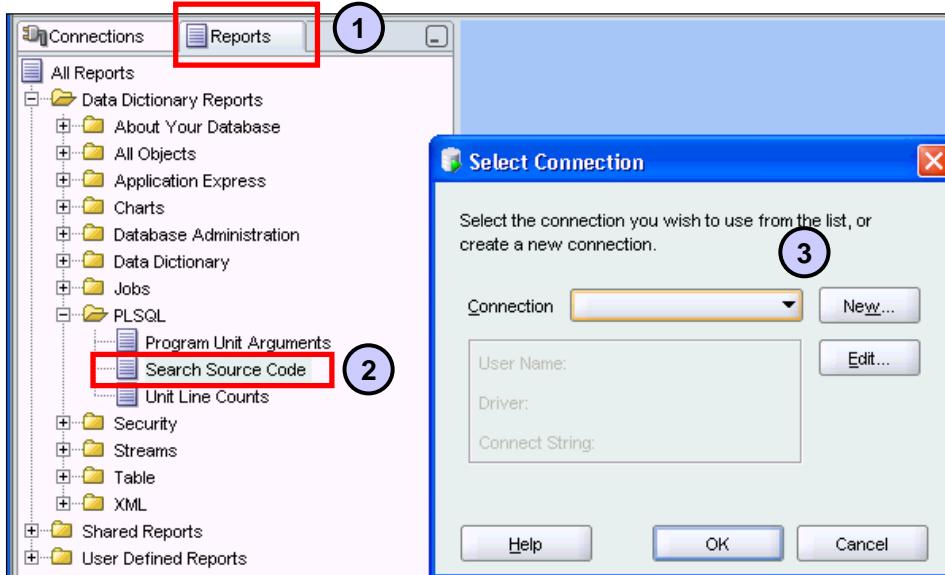
QUERY_CODE_PKG: Examples

In the first example, the ENCAP_COMPLIANCE procedure displays all PL/SQL code objects that reference a table or view directly. Both the code name and table or view name are listed in the output.

In the second example, the FIND_TEXT_IN_CODE procedure returns all PL/SQL code objects that contain the “customers” text string. The code name, line number, and line are listed in the output.

Using SQL Developer to Find Coding Information

Use Reports:



Using SQL Developer to Find Coding Information

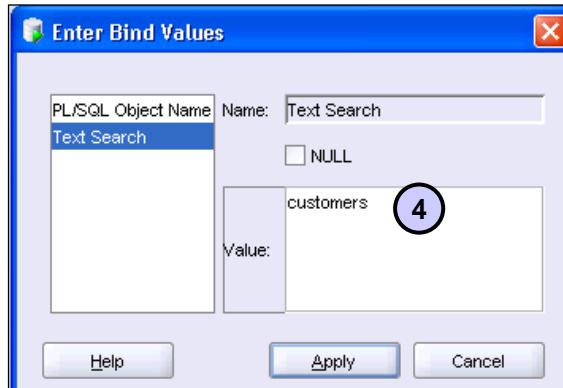
SQL Developer comes with predefined reports that you can use to find information about PL/SQL coding.

If you want to find the occurrence of a text string or an object name, use the Reports feature.

1. Select the Reports tabbed page in SQL Developer.
2. Expand the PL/SQL node and select the Search Source Code option.
3. Enter the connection information.

Using SQL Developer to Find Coding Information

Enter text string or object name



Results

Owner	PL/SQL Object Name	Type	Line	Text
OE	CREDIT_CARD_PKG	PACKAGE BODY	12	FROM customers
OE	CREDIT_CARD_PKG	PACKAGE BODY	36	UPDATE customers
OE	CREDIT_CARD_PKG	PACKAGE BODY	41	UPDATE customers
OE	SET_VIDEO	PROCEDURE	6	SELECT cust_first_name FROM customers
OE	SET_VIDEO	PROCEDURE	13	UPDATE customers SET video = file_ptr

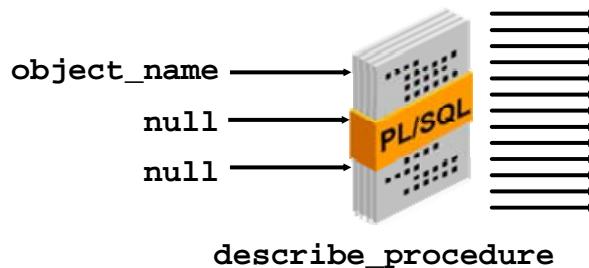
ORACLE®

Using SQL Developer to Find Coding Information (continued)

4. Select either an Object Name or a Text Search for the search type. In the Value field, enter the text string that you want found in your PL/SQL source code for the connection that you specified in step 3.
5. View the results of your search.

Using DBMS_DESCRIBE

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



ORACLE®

11 - 11

Copyright © 2008, Oracle. All rights reserved.

The DBMS_DESCRIBE Package

You can use the DBMS_DESCRIBE package to find information about your procedures. It contains one procedure named DESCRIBE_PROCEDURE. The DESCRIBE_PROCEDURE routine accepts the name of the procedure that you are enquiring about. There are two other IN parameters. Both must be either NULL or an empty string. These two parameters are reserved.

The DBMS_DESCRIBE returns detailed parameter information in a set of associative arrays. The details are numerically coded. You can find the following information from the results returned:

- **Overload:** If overloaded, it holds a value for each version of the procedure.
- **Position:** Position of the argument in the parameter list. 0 is reserved for the RETURN information of a function.
- **Level:** For composite types only; it holds the level of the data type.
- **Argument name:** Name of the argument.
- **Data type:** A numerically coded value representing a data type.
- **Default value:** 0 for no default value, 1 if the argument has a default value.
- **Parameter mode:** 0 = IN, 1 = OUT, 2 = IN OUT .

Note: This is not the complete list of values returned from the DESCRIBE_PROCEDURE routine. For a complete list, see the *PL/SQL Packages and Types Reference 11g Release 1* reference manual.

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;
/
```

1

```
EXEC use_dbms_describe.get_data('ORDERS_APP_PKG.THE_PREDICATE')
```

2

Name	Mode	Position	Datatype
This is the RETURN data for the function:	1	0	1
P_SCHEMA	0	1	1
P_NAME	0	2	1

PL/SQL procedure successfully completed.

ORACLE®

11 - 12

Copyright © 2008, Oracle. All rights reserved.

The DESCRIBE PROCEDURE Routine

Because the DESCRIBE PROCEDURE returns information about your parameters in a set of associative arrays, it is easiest to define a package to call and handle the information returned from it.

In the first example in the slide, the specification for the USE_DBMS_DESCRIBE package is defined. This package holds one procedure, GET_DATA. The GET_DATA routine calls the DBMS_DESCRIBE . DESCRIBE PROCEDURE routine. The implementation of the USE_DBMS_DESCRIBE package is shown on the next page. Note that several associative array variables are defined to hold the values returned via the OUT parameters from the DESCRIBE PROCEDURE routine. Each of these arrays uses the predefined package types:

```
TYPE VARCHAR2_TABLE IS TABLE OF VARCHAR2(30)
INDEX BY BINARY_INTEGER;
TYPE NUMBER_TABLE IS TABLE OF NUMBER INDEX BY BINARY_INTEGER;
```

In the call to the DESCRIBE PROCEDURE routine, you need to pass three parameters: the name of the procedure that you are enquiring about and two null values. These null values are reserved for future use.

In the second example in the slide, the results are displayed for the parameters of the ORDERS_APP_PKG . THE_PREDICATE function. A data type of 1 indicates that it is a VARCHAR2 data type.

The DESCRIBE_PROCEDURE Routine (continued)

Calling DBMS_DESCRIBE.DESCRIBE_PROCEDURE

```
CREATE OR REPLACE PACKAGE use_dbms_describe IS
    PROCEDURE get_data (p_obj_name VARCHAR2);
END use_dbms_describe;
/
CREATE OR REPLACE PACKAGE BODY use_dbms_describe IS
    PROCEDURE get_data (p_obj_name VARCHAR2)
    IS
        v_overload      DBMS_DESCRIBE.NUMBER_TABLE;
        v_position      DBMS_DESCRIBE.NUMBER_TABLE;
        v_level         DBMS_DESCRIBE.NUMBER_TABLE;
        v_arg_name      DBMS_DESCRIBE.VARCHAR2_TABLE;
        v_datatype      DBMS_DESCRIBE.NUMBER_TABLE;
        v_def_value     DBMS_DESCRIBE.NUMBER_TABLE;
        v_in_out        DBMS_DESCRIBE.NUMBER_TABLE;
        v_length        DBMS_DESCRIBE.NUMBER_TABLE;
        v_precision     DBMS_DESCRIBE.NUMBER_TABLE;
        v_scale         DBMS_DESCRIBE.NUMBER_TABLE;
        v_radix         DBMS_DESCRIBE.NUMBER_TABLE;
        v_spare         DBMS_DESCRIBE.NUMBER_TABLE;
    BEGIN
        DBMS_DESCRIBE.DESCRIBE_PROCEDURE
        (p_obj_name, null, null, -- these are the 3 in parameters
         v_overload, v_position, v_level, v_arg_name,
         v_datatype, v_def_value, v_in_out, v_length,
         v_precision, v_scale, v_radix, v_spare, null);
        IF v_in_out.FIRST IS NULL THEN
            DBMS_OUTPUT.PUT_LINE ('No arguments to report.');
        ELSE
            DBMS_OUTPUT.PUT
            ( 'Name'                                Mode');
            DBMS_OUTPUT.PUT_LINE(' Position   Datatype ');
            FOR i IN v_arg_name.FIRST .. v_arg_name.LAST LOOP
                IF v_position(i) = 0 THEN
                    DBMS_OUTPUT.PUT('This is the RETURN data for
                     the function: ');
                ELSE
                    DBMS_OUTPUT.PUT (
                        rpad(v_arg_name(i), LENGTH(v_arg_name(i)) +
                            42-LENGTH(v_arg_name(i)), ' '));
                END IF;
                DBMS_OUTPUT.PUT( '      ' ||
                    v_in_out(i) || '      ' || v_position(i) ||
                    '      ' || v_datatype(i));
                DBMS_OUTPUT.NEW_LINE;
            END LOOP;
        END IF;
    END get_data;
END use_dbms_describe;
```

Using ALL_ARGUMENTS

Query the ALL_ARGUMENTS view to find information about arguments for procedures and functions:

```
SELECT object_name, argument_name, in_out, position, data_type
FROM   all_arguments
WHERE  package_name = 'CREDIT_CARD_PKG';

OBJECT_NAME          ARGUMENT_NAME      IN_OUT    POSITION DATA_TYPE
-----              -----           -----
DISPLAY_CARD_INFO   P_CUST_ID          IN        1 NUMBER
UPDATE_CARD_INFO    OUT                1 OBJECT
UPDATE_CARD_INFO    O_CARD_INFO        OUT       4 TABLE
UPDATE_CARD_INFO    P_CARD_NO          IN        3 VARCHAR2
UPDATE_CARD_INFO    P_CARD_TYPE        IN        2 VARCHAR2
UPDATE_CARD_INFO    P_CUST_ID          IN        1 NUMBER
CUST_CARD_INFO      IN/OUT             1 OBJECT
CUST_CARD_INFO      P_CARD_INFO        IN/OUT    2 TABLE
CUST_CARD_INFO      P_CUST_ID          IN        1 NUMBER
CUST_CARD_INFO      OUT                0 PL/SQL BOOLEAN

10 rows selected.
```

ORACLE®

11 - 14

Copyright © 2008, Oracle. All rights reserved.

Using the ALL_ARGUMENTS Dictionary View

You can also query the ALL_ARGUMENTS dictionary view to find information about the arguments of procedures and functions to which you have access. Like DBMS_DESCRIBE, the ALL_ARGUMENTS view returns information in textual rather than numeric form. Though there is overlap between the two, there is unique information to be found both in DBMS_DESCRIBE and ALL_ARGUMENTS.

In the example shown in the slide, the argument name, mode, position, and data type are returned for ORDERS_APP_PKG. Note the following:

- A position of 1 and a sequence and level of 0 indicates that the procedure has no arguments.
- For a function that has no arguments, it is displayed as a single row for the RETURN clause, with a position of 0.
- The argument name for the RETURN clause is NULL.
- If the programs are overloaded, the OVERLOAD column (not shown in the slide) indicates the Nth overloading; otherwise, it is NULL.
- The DATA_LEVEL column (not shown in the slide) value of 0 identifies a parameter as it appears in the program specification.

Using ALL_ARGUMENTS

Other column information:

- Details about the data type are found in the DATA_TYPE and TYPE_ columns.
- All arguments in the parameter list are at level 0.
- For composite parameters, the individual elements of the composite are assigned levels, starting at 1.
- The POSITION-DATA_LEVEL column combination is unique only for a level 0 argument (the actual parameter, not its subtypes if it is a composite).

ORACLE®

11 - 15

Copyright © 2008, Oracle. All rights reserved.

Using the ALL_ARGUMENTS Dictionary View (continued)

The DATA_TYPE column holds the generic PL/SQL data type. To find more information about the data type, query the TYPE_ columns.

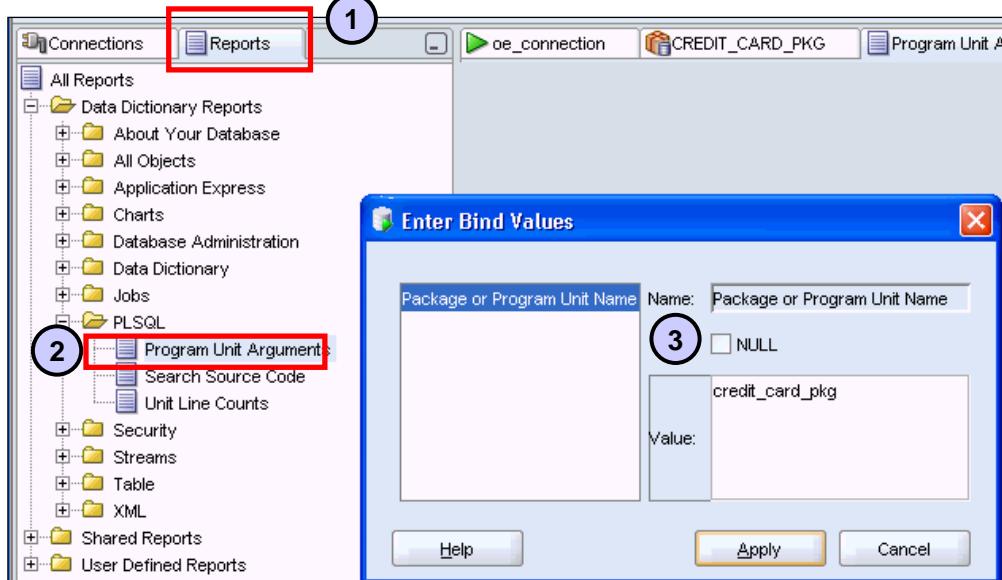
- TYPE_NAME: Holds the name of the type of the argument. If the type is a package local type (that is, it is declared in a package specification), this column displays the name of the package.
- TYPE_SUBNAME: Is relevant only for package local types. Displays the name of the type declared in the package identified in the TYPE_NAME column. For example, if the data type is a PL/SQL table, you can find out the type of the table only by looking at the TYPE_SUBNAME column.

Note: The DEFAULT_VALUE and DEFAULT_LENGTH columns are reserved for future use and do not currently contain information about a parameter's default value. However, you can use DBMS_DESCRIBE to find some default value information. In this package, the parameter DEFAULT_VALUE returns 1 if there is a default value; otherwise, it returns 0.

By combining the information from DBMS_DESCRIBE and ALL_ARGUMENTS, you can find valuable information about parameters, as well as about how your PL/SQL routines are overloaded.

Using SQL Developer to Report on Arguments

Use Reports:



Using SQL Developer to Find Argument Information

SQL Developer comes with predefined reports that you can use to find PL/SQL coding information on your program unit arguments.

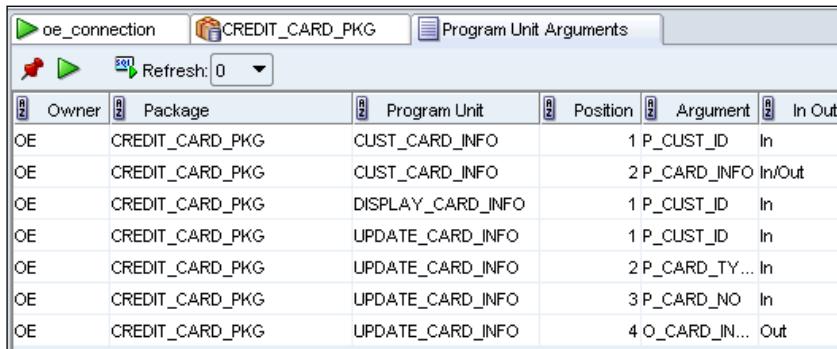
Use the Reports feature to find information on Program Unit Arguments.

1. Select the Reports tabbed page in SQL Developer.
2. Expand the PL/SQL node and select Program Unit Arguments. If prompted, enter the connection information.
3. Enter the name of the program unit on which you want the arguments reported.

Using SQL Developer to Report on Arguments

Results:

4



Owner	Package	Program Unit	Position	Argument	In Out
OE	CREDIT_CARD_PKG	CUST_CARD_INFO	1	P_CUST_ID	In
OE	CREDIT_CARD_PKG	CUST_CARD_INFO	2	P_CARD_INFO	In/Out
OE	CREDIT_CARD_PKG	DISPLAY_CARD_INFO	1	P_CUST_ID	In
OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	1	P_CUST_ID	In
OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	2	P_CARD_TY...	In
OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	3	P_CARD_NO	In
OE	CREDIT_CARD_PKG	UPDATE_CARD_INFO	4	O_CARD_IN...	Out

ORACLE®

11 - 17

Copyright © 2008, Oracle. All rights reserved.

Using SQL Developer to Find Argument Information (continued)

- View the results of your search.

In the example shown above, the output displays the owner of the stored PL/SQL package, the name of the PL/SQL package, the names of the subroutines within the PL/SQL package, the position of the arguments within the package, the argument names, and the argument types (IN, OUT, or IN OUT).

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
-----
PL/SQL Call Stack -----
object      line  object
handle     number  name
0x566ce8e0          4  procedure OE.FIRST_ONE
0x5803f7a8          5  procedure OE.SECOND_ONE
0x569c3770          6  procedure OE.THIRD_ONE
0x567ee3d0          1  anonymous block

PL/SQL procedure successfully completed.
```

ORACLE®

11 - 18

Copyright © 2008, Oracle. All rights reserved.

The DBMS_UTILITY.FORMAT_CALL_STACK Function

Another tool that is available to you is the FORMAT_CALL_STACK function within the DBMS_UTILITY supplied package. It returns the call stack in a formatted character string. The results shown above were generated based on the following routines:

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

CREATE OR REPLACE PROCEDURE second_one
IS
BEGIN
    null;
    first_one;
END;
/
-- continued on next page
```

The DBMS.Utility.Format_Call_Stack Function (continued)

-- continued from previous page

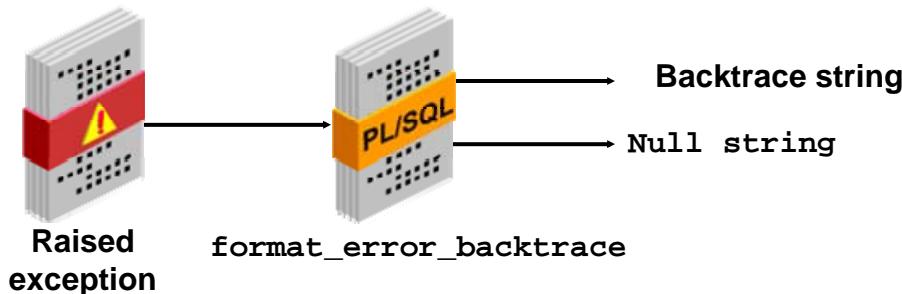
```
CREATE OR REPLACE PROCEDURE third_one
IS
BEGIN
    null;
    null;
    second_one;
END;
/
```

The output from the FORMAT_CALL_STACK function shows the object handle number, the line number where a routine is called from, and the routine that is called. Note that the NULL ; statements added to the procedures are used to emphasize the line number where the routine is called from.

Finding Error Information

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



ORACLE®

11 - 20

Copyright © 2008, Oracle. All rights reserved.

Using DBMS_UTILITY.FORMAT_ERROR_BACKTRACE

You can use this function to display the call stack at the point where an exception was raised, even if the procedure is called from an exception handler in an outer scope. The output returned is similar to the output of the SQLERRM function but not subject to the same size limitation.

Using DBMS_UTILITY.FORMAT_ERROR_STACK

You can use this function to format the current error stack. It can be used in exception handlers to view the full error stack. The function returns the error stack up to 2,000 bytes.

Finding Error Information

```
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_UTILITY.FORMAT_ERROR_STACK() );
    log_errors ( 'Error_Backtrace...' || CHR(10) ||
      DBMS_UTILITY.FORMAT_ERROR_BACKTRACE() );
    DBMS_OUTPUT.PUT_LINE ( '-----' );
END top_with_logging;
/
```

ORACLE®

11 - 21

Copyright © 2008, Oracle. All rights reserved.

Using FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE

To show you the functionality of the FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE functions, a TOP_WITH_LOGGING procedure is created. This procedure calls the LOG_ERRORS procedure and passes to it the results of the FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE functions.

The LOG_ERRORS procedure is shown on the next page.

Finding Error Information

```
CREATE OR REPLACE PROCEDURE log_errors ( i_buff IN VARCHAR2 ) IS
  g_start_pos PLS_INTEGER := 1;
  g_end_pos   PLS_INTEGER;
  FUNCTION output_one_line RETURN BOOLEAN IS
  BEGIN
    g_end_pos := INSTR ( i_buff, CHR(10), g_start_pos );
    CASE g_end_pos > 0
      WHEN TRUE THEN
        DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff,
                                         g_start_pos, g_end_pos-g_start_pos ) );
        g_start_pos := g_end_pos+1;
        RETURN TRUE;
      WHEN FALSE THEN
        DBMS_OUTPUT.PUT_LINE ( SUBSTR ( i_buff, g_start_pos,
                                         (LENGTH(i_buff)-g_start_pos)+1 ) );
        RETURN FALSE;
    END CASE;
  END output_one_line;
BEGIN
  WHILE output_one_line() LOOP NULL;
  END LOOP;
END log_errors;
```

ORACLE®

11 - 22

Copyright © 2008, Oracle. All rights reserved.

The LOG_ERRORS Example

This procedure takes the return results of the FORMAT_ERROR_STACK and FORMAT_ERROR_BACKTRACE functions as an IN string parameter, and reports it back to you using DBMS_OUTPUT.PUT_LINE. The LOG_ERRORS procedure is called twice from the TOP_WITH_LOGGING procedure. The first call passes the results of FORMAT_ERROR_STACK, and the second procedure passes the results of FORMAT_ERROR_BACKTRACE .

Note: You can use UTL_FILE instead of DBMS_OUTPUT to write and format the results to a file.

The LOG_ERRORS Example (continued)

Next, several procedures are created and one procedure calls another so that a stack of procedures is built. The P0 procedure raises a zero divide exception when it is invoked. The call stack is:

```
TOP_WITH_LOGGING > P5 > P4 > P3 > P2 > P1 > P0
```

```
SET DOC OFF
SET FEEDBACK OFF
SET ECHO OFF

CREATE OR REPLACE PROCEDURE P0 IS
  e_01476 EXCEPTION;
  pragma exception_init ( e_01476, -1476 );
BEGIN
  RAISE e_01476; -- this is a zero divide error
END P0;
/
CREATE OR REPLACE PROCEDURE P1 IS
BEGIN
  P0();
END P1;
/
CREATE OR REPLACE PROCEDURE P2 IS
BEGIN
  P1();
END P2;
/
CREATE OR REPLACE PROCEDURE P3 IS
BEGIN
  P2();
END P3;
/
CREATE OR REPLACE PROCEDURE P4 IS
  BEGIN P3();
END P4;
/
CREATE OR REPLACE PROCEDURE P5 IS
  BEGIN P4();
END P5;
/
CREATE OR REPLACE PROCEDURE top IS
BEGIN
  P5(); -- this procedure is used to show the results
        -- without using the TOP_WITH_LOGGING routine.
END top;
/
SET FEEDBACK ON
```

Finding Error Information

Results:

```
EXECUTE top_with_logging
Error_Stack...
ORA-01476: divisor is equal to zero
Error_Backtrace...
ORA-06512: at "OE.P0", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP_WITH_LOGGING", line 7
-----
```

ORACLE®

11 - 24

Copyright © 2008, Oracle. All rights reserved.

Finding Error Information Results

The results from executing the TOP_WITH_LOGGING procedure is shown. Note that the error stack displays the exception encountered. The backtrace information traces the flow of the exception to its origin.

If you execute the TOP procedure without using the TOP_WITH_LOGGING procedure, these are the results:

```
EXECUTE top
BEGIN top; END;
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at "OE.P0", line 5
ORA-06512: at "OE.P1", line 3
ORA-06512: at "OE.P2", line 3
ORA-06512: at "OE.P3", line 3
ORA-06512: at "OE.P4", line 2
ORA-06512: at "OE.P5", line 2
ORA-06512: at "OE.TOP", line 3
ORA-06512: at line 1
```

Note that the line number reported is misleading.

Lesson Agenda

- Running reports on source code
- Determining identifier types and usages
- Using DBMS_METADATA to retrieve object definitions



PL/Scope Concepts

- Definition:
 - 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
 - Gathers data on scoping and overloading
- Benefits
 - Can potentially increase developer productivity
 - Is a valuable resource in helping developers understand source code
 - Can be used to build a PL/SQL IDE

ORACLE®

11 - 26

Copyright © 2008, Oracle. All rights reserved.

PL/Scope Concepts

With PL/Scope, you can produce a cross-referenced repository of PL/SQL identifiers to gather information about your PL/SQL applications.

PL/Scope has two targeted audiences:

- PL/SQL application developers: Targeted to use PL/Scope through IDEs such as JDeveloper and SQL Developer
- IDE and Tools developers: Provides the means to build a comprehensive cross-referenced repository by extracting, categorizing, organizing, and storing all identifiers discovered in the PL/SQL source code

Collecting PL/Scope Data

Collected data includes:

- Identifier types
- Usages
 - Declaration
 - Definition
 - Reference
 - Call
 - Assignment
- Location of usage



ORACLE®

11 - 27

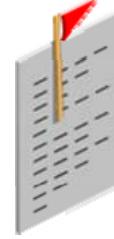
Copyright © 2008, Oracle. All rights reserved.

Collecting PL/Scope Data

You can use PL/Scope to collect data about the user-defined identifiers found in the PL/SQL source code. The data is collected at compilation time and made available in the static data dictionary views.

Using PL/Scope

- Set the PL/SQL compilation parameter PLSCOPE_SETTINGS.
- Valid values are IDENTIFIERS:
 - ALL – Collect all PL/SQL identifier actions found in compiled source
 - NONE – Do not collect any identifier actions (the default)
- Set at the session, system, or per library unit basis:
 - 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_IDENTIFIER catalog view holds the collected identifier values.



ORACLE®

11 - 28

Copyright © 2008, Oracle. All rights reserved.

Using PL/Scope

To use PL/Scope, you need to set the PL/SQL compilation parameter PLSCOPE_SETTINGS to either 'IDENTIFIERS: ALL' or 'IDENTIFIERS: NONE'. You can use the ALTER SESSION, ALTER SYSTEM, or ALTER COMPILE statements to set this parameter. After the parameter is set, any code that you compile is analyzed for PL/Scope. The information collected is gathered in the USER/ALL/DBA_IDENTIFIER dictionary view.

The identifier action describes how the identifier is used, such as in a declaration, definition, reference, assignment, or call.

Query the USER | ALL | DBA_PLSQL_OBJECT_SETTINGS views to view what the PLSCOPE_SETTINGS are set to. This view contains a column called PLSCOPE_SETTINGS. When you query this column, by default all objects are set to 'IDENTIFIERS: NONE', unless you reset the PL/SQL compilation parameter and recompile the code.

The USER/ALL/DBA_IDENTIFIERs Catalog View

Column	Description
OWNER	Owner of the identifier
NAME	Name of the identifier (may not be unique)
SIGNATURE	A unique signature for this identifier
TYPE	The type of the identifier, such as variable, formal, varray
OBJECT_NAME	The object name where the action occurred
OBJECT_TYPE	The type of object where the action occurred
USAGE	The action performed on the identifier, such as declaration, definition, reference, assignment, or call
USAGE_ID	The unique key for the identifier usage
LINE	The line where the identifier action occurred
COL	The column where the identifier action occurred
USAGE_CONTEXT_ID	The context USAGE_ID of the identifier action

ORACLE®

11 - 29

Copyright © 2008, Oracle. All rights reserved.

New Dictionary View

The USER/ALL/DBA_IDENTIFIERs dictionary view is added in Oracle Database 11g to collect information about your identifiers for any code that is compiled or altered when the PL/SQL compilation parameter is set to PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL'.

Sample Data for PL/Scope

Sample data for scoping:

```
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
```

ORACLE®

11 - 30

Copyright © 2008, Oracle. All rights reserved.

Sample Data

The CREDIT_CARD_PKG created earlier is used for PL/SQL scoping. The complete code is shown on the following page.

The code is a simple package that contains one type, two procedures, and one function. Identifier information will be collected on this package in the following pages.

Sample Data (continued)

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2);

    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;

    PROCEDURE display_card_info
        (p_cust_id NUMBER)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN
            FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
                DBMS_OUTPUT.PUT('Card Type: ' || v_card_info(idx).card_type
                                || '\n');
                DBMS_OUTPUT.PUT_LINE(' Card No: ' || v_card_info(idx).card_num );
            END LOOP;
        ELSE
            DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
        END IF;
    END display_card_info;
END credit_card_pkg; -- package body
/
```

Collecting Identifiers

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

- Identifier information is collected in the `USER_IDENTIFIER`s dictionary view, where you can:
 - Perform a basic identifier search
 - Use contexts to describe identifiers
 - Find identifier actions
 - Describe identifier actions



Collecting Identifiers

In the example shown above, the PL/SQL compilation parameter `PLSCOPE_SETTINGS` is turned on to collect information about all identifiers. The `CREDIT_CARD_PKG` is then recompiled. After it is recompiled, the identifier information is available in the `ALL | USER | DBA_IDENTIFIER` views.

To verify that you have enabled identifier information to be collected on the `CREDIT_CARD_PKG` package, you can issue the following statement:

```
SELECT PLSCOPE_SETTINGS  
FROM USER_PLSQL_OBJECT_SETTINGS  
WHERE NAME='CREDIT_CARD_PKG' AND TYPE='PACKAGE BODY';  
  
PLSCOPE_SETTINGS  
-----  
IDENTIFIERS:ALL
```

Viewing Identifier Information

Create a hierarchical report of identifier information:

```
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;
```

ORACLE®

11 - 33

Copyright © 2008, Oracle. All rights reserved.

Viewing Identifier Information

The sample SQL statement shown above retrieves and formats the identifier information that is collected for the CREDIT_CARD_PKG package body. Note that the inline view retrieves values from the Name, Type, and Usage columns in the USER_IDENTIFIERS dictionary view.

Viewing Identifier Information

Results:

```
IDENTIFIER_USAGE_CONTEXTS
-----
Credit_Card_Pkg..... package           definition
Cust_Card_Info.... function            definition
  P_Cust_Id..... formal in           declaration
  P_Card_Info.... formal in out      declaration
  V_Card_Info_Exis variable          declaration
  P_Card_Info.... formal in out      assignment
  P_Cust_Id..... formal in           reference
  Plitblm..... synonym              call
    P_Card_Info... formal in out     reference
    V_Card_Info_Ex variable          assignment
    V_Card_Info_Exis variable         assignment
...
57 rows selected.
```

ORACLE®

11 - 34

Copyright © 2008, Oracle. All rights reserved.

Viewing Identifier Information (continued)

The results are formatted and display the name of the identifier, type, and how the identifier is used.

Note: PLITBLM is an Oracle-supplied package that contains subprograms to help implement the basic language features. PLITBLM handles index-table operations.

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_IDENTIFIER
WHERE USAGE= 'DECLARATION'
ORDER BY OBJECT_TYPE, USAGE_ID;
```

NAME	SIGNATURE	TYPE
CREDIT_CARD_PKG	66A51BCDE66BBF590EDEB7610811A453	PACKAGE
CUST_CARD_INFO	D003ACE1F0208A3FF2C208F869E76C7E	FUNCTION
P_CUST_ID	27A70F461EABB58444168E9628032EBD	FORMAL IN
P_CARD_INFO	01BF97F7BEB616D9A0E98594DB4DABE3	FORMAL IN OUT
UPDATE_CARD_INFO	96AE51FF4F2739AECFC92550B1799E00	PROCEDURE
P_CUST_ID	6A1CF30248AFB9EE21690B96495154F6	FORMAL IN
P_CARD_TYPE	641F9E0C76DB22D0E9674B465CFFEB48	FORMAL IN
...		
I	1E6FA1EA4515EFCE69FD37307943F08D	VARIABLE
IDX	4EEC75280E2FC886ECB3E76E1710C955	ITERATOR
24 rows selected.		

ORACLE®

Performing a Basic Identifier Search

You can display the unique identifiers in your schema by querying for all 'DECLARATION' identifier actions. Valid actions are:

- DECLARATION: All identifiers have one and only one declaration. Each declaration may also have an associated type.
- TYPE: Has the value of either packages, function or procedures, object types, triggers, or exceptions
- SIGNATURE: A unique value that distinguishes the identifier from other identifiers with the same name, whether they are defined in the same program unit or different program units

Using USER_IDENTIFIERS to Find All Local Variables

Find all local variables:

```
SELECT a.NAME variable_name, b.NAME context_name, a.SIGNATURE
FROM USER_IDENTIFIERS a, USER_IDENTIFIERS b
WHERE a.USAGE_CONTEXT_ID = b.USAGE_ID
  AND a.TYPE = 'VARIABLE'
  AND a.USAGE = 'DECLARATION'
  AND a.OBJECT_NAME = 'CREDIT_CARD_PKG'
  AND a.OBJECT_NAME = b.OBJECT_NAME
  AND a.OBJECT_TYPE = b.OBJECT_TYPE
  AND (b.TYPE = 'FUNCTION' or b.TYPE = 'PROCEDURE')
ORDER BY a.OBJECT_TYPE, a.USAGE_ID;
```

VARIABLE_NAME	CONTEXT_NAME	SIGNATURE
V_CARD_INFO_EXISTS	CUST_CARD_INFO	6B962B4BBE1B5B8C0A3EEF2A41880C21
V_CARD_INFO	UPDATE_CARD_INFO	80CB5EC1F0C2CEA4171128FD462D338D
I	UPDATE_CARD_INFO	CE0781AFC31E0A2EC8582386F708E0F9
V_CARD_INFO	DISPLAY_CARD_INFO	7EADE5CABDE4FEA1515227728D336144
I	DISPLAY_CARD_INFO	1E6FA1EA4515EFCE69FD37307943F08D

ORACLE®

Finding Information About Local Variables

In the example shown above, all local variables belonging to procedures or functions are found for the CREDIT_CARD_PKG package.

Finding Identifier Actions

Find all usages performed on the local variable:

```
SELECT USAGE, USAGE_ID, OBJECT_NAME, OBJECT_TYPE  
FROM USER_IDENTIFIERS  
WHERE SIGNATURE='1E6FA1EA4515EFCE69FD37307943F08D'  
ORDER BY OBJECT_TYPE, USAGE_ID;
```

USAGE	USAGE_ID	OBJECT_NAME	OBJECT_TYPE
DECLARATION	42	CREDIT_CARD_PKG	PACKAGE BODY

ORACLE®

11 - 37

Copyright © 2008, Oracle. All rights reserved.

Finding Identifier Actions

You can find all actions performed on an identifier. In the example in the slide, all actions performed on a variable are found by searching on the signature of the variable. The variable is called *i*, and it is used in the DISPLAY_CARD_INFO procedure. Variable *i*'s signature is found in the previous query. It is available to you in the SIGNATURE column of the USER_IDENTIFIERS dictionary view.

The different types of the identifier usage are:

- DECLARATION
- DEFINITION
- CALL
- REFERENCE
- ASSIGNMENT

Describing 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='CE0781AFC31E0A2EC8582386F708E0F9'  
AND USAGE='ASSIGNMENT';
```

LINE	COL	OBJECT_NAME	OBJECT_TYPE
33	7	CREDIT_CARD_PKG	PACKAGE BODY

ORACLE®

11 - 38

Copyright © 2008, Oracle. All rights reserved.

Describing Identifier Actions (continued)

The local identifier *i* is found on line 33 of the CREDIT_CARD_PKG body. Note that this *i* variable is different from the one shown on the previous page.

Lesson Agenda

- Running reports on source code
- Determining identifier types and usages
- Using DBMS_METADATA to retrieve object definitions



DBMS_METADATA Package

The DBMS_METADATA package provides a centralized facility for the extraction, manipulation, and resubmission of dictionary metadata.



ORACLE®

11 - 40

Copyright © 2008, Oracle. All rights reserved.

DBMS_METADATA Package

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.

You can use DBMS_METADATA for extracting metadata from the dictionary, manipulating the metadata (adding columns, changing column data types, and so on), and then converting the metadata to data definition language (DDL) so that the object can be re-created on the same or another database. In the past, you needed to do this programmatically with problems resulting in each new release.

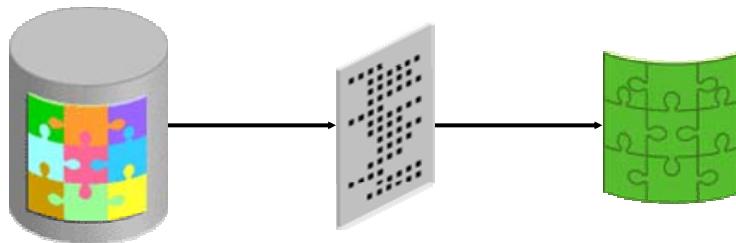
The DBMS_METADATA functionality is used for the Oracle 10g Export/Import replacement, commonly called “the Data Pump.”

Note: For more information about the DBMS_DATAPUMP package, refer to the online course titled *Oracle Database 10g: Reduce Management - Tools and Utilities*.

Metadata API

Processing involves the following steps:

1. Fetch an object's metadata as XML.
2. Transform the XML in a variety of ways (including transforming it into SQL DDL).
3. Submit the XML to re-create the object.



ORACLE®

11 - 41

Copyright © 2008, Oracle. All rights reserved.

Metadata API

Every entity in the database is modeled as an object that belongs to an object type. For example, the ORDERS table is an object; its object type is TABLE. When you fetch an object's metadata, you must specify the object type.

Every object type is implemented by using three entities:

- A user-defined type (UDT) whose attributes comprise all metadata for objects of the type.
An object's XML representation is a translation of a type instance into XML with the XML tag names derived from the type attribute names. (In the case of tables, several UDTs are needed to represent the different varieties of the object type.)
- An object view of the UDT that populates the instances of the object type
- An Extensible Style Sheet Language (XSL) script that converts the XML representation of an object into SQL DDL

Subprograms in DBMS_METADATA

Name	Description
OPEN	Specifies the type of object to be retrieved, the version of its metadata, and the object model. The return value is an opaque context handle for the set of objects.
SET_FILTER	Specifies restrictions on the objects to be retrieved, such as the object name or schema.
SET_COUNT	Specifies the maximum number of objects to be retrieved in a single FETCH_xxx call.
GET_QUERY	Returns the text of the queries that will be used by FETCH_xxx.
SET_PARSE_ITEM	Enables output parsing and specifies an object attribute to be parsed and returned.
ADD_TRANSFORM	Specifies a transform that FETCH_xxx applies to the XML representation of the retrieved objects.
SET_TRANSFORM_PARAM, SET_REMAP_PARAM	Specifies parameters to the XSLT stylesheet identified by transform_handle.
FETCH_XXX	Returns metadata for objects that meet the criteria established by OPEN, SET_FILTER.
CLOSE	Invalidates the handle returned by OPEN and cleans up the associated state.

ORACLE®

Subprograms in DBMS_METADATA

The table provides an overview of the procedures and functions that are available in the DBMS_METADATA package. To retrieve metadata, you can specify the:

- Kind of object to be retrieved—either an object type (a table, index, procedure) or a heterogeneous collection of object types that form a logical unit (such as database export and schema export)
- Selection criteria (owner, name, and so on)
- “parse items” attributes of objects to be parsed and returned separately
- Transformations on the output, implemented by XSLT scripts

The package provides two types of retrieval interfaces for two types of usage:

- **For programmatic use:** OPEN, SET_FILTER, SET_COUNT, GET_QUERY, SET_PARSE_ITEM, ADD_TRANSFORM, SET_TRANSFORM_PARAM, SET_REMAP_PARAM (new in Oracle Database 10g), FETCH_xxx, and CLOSE. These enable a flexible selection criteria and the extraction of a stream of objects.
- **For use in SQL queries and for ad hoc browsing:** The GET_xxx interfaces (GET_XML and GET_DDL) return metadata for a single named object. The GET_DEPENDENT_xxx and GET_GRANTED_xxx interfaces return metadata for one or more dependent or granted objects. None of these APIs supports heterogeneous object types.

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.

ORACLE®

11 - 43

Copyright © 2008, Oracle. All rights reserved.

FETCH_xxx Subprograms

These functions and procedures return metadata for objects meeting the criteria established by the call to the OPEN function that returned the handle, and the subsequent calls to SET_FILTER, SET_COUNT, ADD_TRANSFORM, and so on. Each call to FETCH_xxx returns the number of objects specified by SET_COUNT (or a smaller number, if fewer objects remain in the current cursor) until all objects are returned.

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');
...
```

ORACLE

11 - 44

Copyright © 2008, Oracle. All rights reserved.

SET_FILTER Procedure

You use the SET_FILTER procedure to identify restrictions on the objects that are to be retrieved. For example, you can specify restrictions on an object or schema that is being retrieved. This procedure is overloaded with the parameters that have the following meanings:

- handle is the handle returned from the OPEN function.
- name is the name of the filter. For each filter, the object type applies to its name, data type (text or Boolean), and meaning or effect (including its default value, if there is one).
- value is the value of the filter. It can be text, Boolean, or a numeric value.
- object_type_path is a path name designating the object types to which the filter applies. By default, the filter applies to the object type of the OPEN handle.

If you use an expression filter, it is placed to the right of a SQL comparison, and the value is compared with it. The value must contain parentheses and quotation marks where appropriate. A filter value is combined with a particular object attribute to produce a WHERE condition in the query that fetches the objects.

Filters

There are over 70 filters that are organized into object type categories such as:

- Named objects
- Tables
- Objects dependent on tables
- Index
- Dependent objects
- Granted objects
- Table data
- Index statistics
- Constraints
- All object types
- Database export

ORACLE®

11 - 45

Copyright © 2008, Oracle. All rights reserved.

Filters

There are over 70 filters that you can specify when using the SET_FILTER procedure. These filters are organized into object type categories. Some of the new object type categories in Oracle Database 10g are listed in the slide.

When using the SET_FILTER procedure, you specify the name of the filter and its respective value.

For example, you can use the SCHEMA filter with a value to identify the schema whose objects are selected. Then use a second call to the SET_FILTER procedure and use a filter named INCLUDE_USER that has a Boolean data type for its value. If it is set to TRUE, objects that contain privileged information about the user are retrieved.

```
DBMS_METADATA.SET_FILTER(handle, SCHEMA, 'OE');
DBMS_METADATA.SET_FILTER(handle, INCLUDE_USER, TRUE);
```

Each call to SET_FILTER causes a WHERE condition to be added to the underlying query that fetches the set of objects. The WHERE conditions are combined by using an AND operator, so that you can use multiple SET_FILTER calls to refine the set of objects to be returned.

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');
```

ORACLE

11 - 46

Copyright © 2008, Oracle. All rights reserved.

Examples of Setting Filters

The example shown in the slide calls the SET_FILTER procedure several times to create a WHERE condition that identifies which object types are to be fetched. First, the objects in the PAYROLL and OE schemas are identified as object types to be fetched. Subsequently, the SET_FILTER procedure identifies certain object types (functions, procedures, and packages) and view object names that are to be excluded.

Programmatic Use: Example 1

```
CREATE PROCEDURE example_one IS
    v_hdl      NUMBER; v_th1  NUMBER; v_th2  NUMBER;
    v_doc      sys.ku$_ddls; ←①
BEGIN
    v_hdl := DBMS_METADATA.OPEN('SCHEMA_EXPORT'); ←②
    DBMS_METADATA.SET_FILTER(v_hdl,'SCHEMA','OE'); ←③
    v_th1 := DBMS_METADATA.ADD_TRANSFORM(v_hdl, ←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');
    DBMS_METADATA.SET_TRANSFORM_PARAM(v_th2, ←
        'SQLTERMINATOR', TRUE);
    DBMS_METADATA.SET_TRANSFORM_PARAM(v_th2, ←⑥
        '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;
```

ORACLE®

11 - 47

Copyright © 2008, Oracle. All rights reserved.

Programmatic Use: Example 1

In this example, all objects are retrieved from the HR schema as creation DDL. The MODIFY transform is used to change the tablespaces for the tables.

1. The DBMS_METADATA package has several predefined types that are owned by SYS. The sys.ku\$_ddls type is defined in the DBMS_METADATA package. It is a table type that holds the CLOB type of data.
2. You use the OPEN function to specify the type of object to be retrieved, the version of its metadata, and the object model. It returns a context handle for the set of objects.
In this example, 'SCHEMA_EXPORT' is the object type, which indicates all metadata objects in a schema. There are 85 predefined types of objects for the model that you can specify for this parameter. Both the version of metadata and the object model parameters are not identified in this example. The version of metadata parameter defaults to 'COMPATIBLE'. You can also specify 'LATEST' or a specific database version.
3. The SET_FILTER procedure identifies restrictions on the objects that are to be retrieved.

Programmatic Use: Example 1 (continued)

4. The ADD_TRANSFORM function specifies a transform that FETCH_XXX applies to the XML representation of the retrieved objects. You can have multiple transforms. In the example, two transforms occur, one for each of the th1 and th2 program variables. The ADD_TRANSFORM function accepts four parameters and returns a number representing the opaque handle to the transform. The parameters are the handle returned from the OPEN statement, the name of the transform (DDL, DROP, or MODIFY), the encoding name (which is the name of the national language support [NLS] character set in which the style sheet pointed to by the name is encoded), and the object type. If the object type is omitted, the transform applies to all objects; otherwise, it applies only to the object type specified. The first transform shown in the program code is the handle returned from the OPEN function. The second transform shown in the code has two parameter values specified. The first parameter is the handle identified from the OPEN function. The second parameter value is the DDL, which means the document is transformed to the DDL that creates the object. The output of this transform is not an XML document. The third and fourth parameters are not specified. Both take the default values for the encoding and object type parameters.
5. The SET_REMAP_PARAM procedure identifies the parameters to the XSLT style sheet identified by the transform handle, which is the first parameter passed to the procedure. In the example, the second parameter value 'REMAP_TABLESPACE' means that the objects have their tablespaces renamed from an old value to a new value. In the ADD_TRANSFORM function, the choices are DDL, DROP, or MODIFY. For each of these values, the SET_REMAP_PARAM identifies the name of the parameter. REMAP_TABLESPACE means the objects in the document will have their tablespaces renamed from an old value to a new value. The third and fourth parameters identify the old value and new value. In this example, the old tablespace name is SYSTEM, and the new tablespace name is TBS1.
6. SET_TRANSFORM_PARAM works similarly to SET_REMAP_PARAM. In the code shown, the first call to SET_TRANSFORM_PARAM identifies the parameters for the th2 variable. The SQLTERMINATOR and TRUE parameter values cause the SQL terminator (; or /) to be appended to each DDL statement. The second call to SET_TRANSFORM_PARAM identifies more characteristics for the th2 variable. REF_CONSTRAINTS, FALSE, and TABLE means that the referential constraints on the tables are not copied to the document.
7. The FETCH_DDL function returns the metadata for objects that meet the criteria established by the OPEN, SET_FILTER, ADD_TRANSFORM, SET_REMAP_PARAM, and SET_TRANSFORM_PARAM subroutines.
8. The CLOSE function invalidates the handle returned by the OPEN function and cleans up the associated state. Use this function to terminate the stream of objects established by the OPEN function.

Programmatic Use: Example 2

```
CREATE FUNCTION get_table_md RETURN CLOB IS
  v_hdl  NUMBER; -- returned by 'OPEN'
  v_th   NUMBER; -- returned by 'ADD_TRANSFORM'
  v_doc  CLOB;
BEGIN
  -- specify the OBJECT TYPE
  v_hdl := DBMS_METADATA.OPEN('TABLE');
  -- use FILTERS to specify the objects desired
  DBMS_METADATA.SET_FILTER(v_hdl , 'SCHEMA','OE');
  DBMS_METADATA.SET_FILTER
    (v_hdl , 'NAME','ORDERS');
  -- request to be TRANSFORMED into creation DDL
  v_th := DBMS_METADATA.ADD_TRANSFORM(v_hdl,'DDL');
  -- FETCH the object
  v_doc := DBMS_METADATA.FETCH_CLOB(v_hdl);
  -- release resources
  DBMS_METADATA CLOSE(v_hdl);
  RETURN v_doc;
END;
/
```

ORACLE®

11 - 49

Copyright © 2008, Oracle. All rights reserved.

Programmatic Use: Example 2

This example returns the metadata for the ORDERS table. The result is:

```
set pagesize 0
set long 1000000
SELECT get_table_md FROM dual;

CREATE TABLE "OE"."ORDERS"
  (
    "ORDER_ID" NUMBER(12,0),
    "ORDER_DATE" TIMESTAMP (6) WITH LOCAL TIME ZONE
    CONSTRAINT "ORDER_DATE_NN" NOT
    NULL ENABLE,
    "ORDER_MODE" VARCHAR2(8),
    "CUSTOMER_ID" NUMBER(6,0) CONSTRAINT
    "ORDER_CUSTOMER_ID_NN" NOT NULL ENABLE,
    "ORDER_STATUS" NUMBER(2,0),
    "ORDER_TOTAL" NUMBER(8,2),
    "SALES_REP_ID" NUMBER(6,0),
    "PROMOTION_ID" NUMBER(6,0),
    CONSTRAINT "ORDER_MODE_LOV" CHECK (order_mode in
    ...
  )
```

Programmatic Use: Example 2 (continued)

```
...
CONSTRAINT "ORDER_TOTAL_MIN" CHECK (order_total >= 0) ENABLE,
CONSTRAINT "ORDER_PK" PRIMARY KEY ("ORDER_ID")
    USING INDEX PCTFREE 10 INITTRANS 2 MAXTRANS 255 NOLOGGING
        COMPUTE STATISTICS
STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS
2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL
DEFAULT)
TABLESPACE "EXAMPLE" ENABLE,
CONSTRAINT "ORDERS_CUSTOMER_ID_FK" FOREIGN KEY
( "CUSTOMER_ID" )
    REFERENCES "OE"."CUSTOMERS" ("CUSTOMER_ID")
    ON DELETE SET NULL ENABLE,
CONSTRAINT "ORDERS_SALES REP_FK" FOREIGN KEY
( "SALES REP_ID" ) REFERENCES "HR"."EMPLOYEES"
( "EMPLOYEE_ID" ) ON DELETE SET NULL ENABLE
)
PCTFREE 10 PCTUSED 40 INITTRANS 1 MAXTRANS 255
NOCOMPRESS NOLOGGING
STORAGE(INITIAL 65536 NEXT 1048576
MINEXTENTS 1 MAXEXTENTS 2147483645
PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1
BUFFER_POOL DEFAULT)
TABLESPACE "EXAMPLE"
```

You can accomplish the same effect with the browsing interface:

```
SELECT dbms_metadata.get_ddl
    ('TABLE', 'ORDERS', 'OE')
FROM dual;
```

Browsing APIs

Name	Description
GET_XXX	The GET_XML and GET_DDL functions return metadata for a single named object.
GET_DEPENDENT_XXX	This function returns metadata for a dependent object.
GET_GRANTED_XXX	This function returns metadata for a granted object.
Where xxx is:	DDL or XML

ORACLE®

11 - 51

Copyright © 2008, Oracle. All rights reserved.

Browsing APIs

The browsing APIs are designed for use in SQL queries and ad hoc browsing. These functions allow you to fetch metadata for objects with a single call. They encapsulate calls to OPEN, SET_FILTER, and so on. The function that you use depends on the characteristics of the object type and whether you want XML or DDL.

For some object types, you can use multiple functions. You can use GET_XXX to fetch an index by name, or GET_DEPENDENT_XXX to fetch the same index by specifying the table on which it is defined.

GET_XXX returns a single object name.

For GET_DEPENDENT_XXX and GET_GRANTED_XXX, an arbitrary number of granted or dependent objects may match the input criteria. However, you can specify an object count when fetching these objects.

If you invoke these functions from SQL*Plus, you should use the SET LONG and SET PAGESIZE commands to retrieve the complete, uninterrupted output.

```
SET LONG 2000000  
SET PAGESIZE 300
```

Browsing APIs: Examples

1. Get the XML representation of OE.ORDERS:

```
SELECT DBMS_METADATA.GET_XML
      ('TABLE', 'ORDERS', 'OE')
FROM   dual;
```

2. Fetch the DDL for all object grants on OE.ORDERS:

```
SELECT DBMS_METADATA.GET_DEPENDENT_DDL
      ('OBJECT_GRANT', 'ORDERS', 'OE')
FROM   dual;
```

3. Fetch the DDL for all system grants granted to OE:

```
SELECT DBMS_METADATA.GET_GRANTED_DDL
      ('SYSTEM_GRANT', 'OE')
FROM   dual;
```

ORACLE®

11 - 52

Copyright © 2008, Oracle. All rights reserved.

Browsing APIs: Examples

1. Results for fetching the XML representation of OE.ORDERS:

```
DBMS_METADATA.GET_XML('TABLE', 'ORDERS', 'OE')
-----
<?xml version="1.0"?>
<ROWSET>
  <ROW>
    <TABLE_T>
      <VERS_MAJOR>1</VERS_MAJOR>
```

2. Results for fetching the DDL for all object grants on OE.ORDERS:

```
DBMS_METADATA.GET_DEPENDENT_DDL
  ('OBJECT_GRANT', 'ORDERS', 'OE')
-----
GRANT SELECT ON "OE"."ORDERS" TO "PM"
GRANT SELECT ON "OE"."ORDERS" TO "BI"
```

3. Results for fetching the DDL for all system grants granted to OE:

```
DBMS_METADATA.GET_GRANTED_DDL('SYSTEM_GRANT', 'OE')
-----
GRANT QUERY REWRITE TO "OE"
...
```

Browsing APIs: Examples

```
BEGIN
  DBMS_METADATA.SET_TRANSFORM_PARAM(
    DBMS_METADATA.SESSION_TRANSFORM,
    'STORAGE', false);
END;
/
SELECT DBMS_METADATA.GET_DDL('TABLE', u.table_name)
FROM user_all_tables u
WHERE u.nested = 'NO'
AND (u.iot_type IS NULL OR u.iot_type = 'IOT');

BEGIN
  DBMS_METADATA.SET_TRANSFORM_PARAM(
    DBMS_METADATA.SESSION_TRANSFORM, 'DEFAULT');
END;
/
```

- 1
- 2
- 3

ORACLE®

11 - 53

Copyright © 2008, Oracle. All rights reserved.

Browsing APIs: Examples (continued)

The example in the slide shows how to fetch creation DDL for all “complete” tables in the current schema, filtering out nested tables and overflow segments. The steps shown in the slide are as follows:

1. The SET_TRANSFORM_PARAM function specifies that the storage clauses are not to be returned in the SQL DDL. The SESSION_TRANSFORM function is interpreted to mean “for the current session.”
2. Use the GET_DDL function to retrieve DDL on all nonnested and non-IOT (index-organized table) tables.

```
CREATE TABLE "HR"."COUNTRIES"
( "COUNTRY_ID" CHAR(2)
  CONSTRAINT "COUNTRY_ID_NN" NOT NULL  ENABLE,
  "COUNTRY_NAME" VARCHAR2(40),
  "REGION_ID" NUMBER,
  CONSTRAINT "COUNTRY_C_ID_PK"
  PRIMARY KEY ( "COUNTRY_ID" )  ENABLE,
  CONSTRAINT "COUNTR_REG_FK" FOREIGN KEY
  ...
  );
```

3. Reset the session-level parameters to their defaults.

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



Summary

In this lesson, you learned how to use the dictionary views and supplied PL/SQL packages to analyze your PL/SQL applications.

Use the OE schema for this practice.

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”

Practice 11: Overview

This practice covers the following topics:

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



Practice 11: Overview

Using the OE application that you have created, write code to analyze your application. You will perform the following:

- Find coding information
- Use PL/Scope
- Use DBMS_METADATA

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”

Practice 11

Finding Coding Information

In this practice, you use PL/SQL and SQL Developer to analyze your code.

1. Create the QUERY_CODE_PKG package to search your source code.
 - a. Run the lab_11_01_a.sql script to create the QUERY_CODE_PKG package.
 - b. Run the ENCAP_COMPLIANCE procedure to see which of your programs reference tables or views. (Note: Your results might differ slightly.)
 - c. Run the FIND_TEXT_IN_CODE procedure to find all references to ‘ORDERS.’ (Note: Your results might differ slightly.)
 - d. Use the SQL Developer Reports feature to find the same results for step C shown above.

Practice 11 (continued)

Using PL/Scope

2. In this practice, you use PL/Scope.

- a. Enable your session to collect identifiers.

```
ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL' ;
```

- b. Recompile your CREDIT_CARD_PKG code.

```
ALTER PACKAGE credit_card_pkg COMPILE;
```

- c. Verify that your PLSCOPE_SETTING is set correctly by issuing the following statement:

```
SELECT PLSCOPE_SETTINGS
  FROM USER_PLSQL_OBJECT_SETTINGS
 WHERE NAME='CREDIT_CARD_PKG' AND TYPE='PACKAGE BODY' ;
```

- d. Execute the following statement to create a hierarchical report on the identifier information about the CREDIT_CARD_PKG code. You can run the lab_11_02_d.sql script file.

```
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;
```

Practice 11 (continued)

Using DBMS_METADATA

3. Use DBMS_METADATA to find the metadata for the ORDER_ITEMS table.
 - a. Create the GET_TABLE_MD function. You can run the lab_11_03_a.sql script.

```
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' , 'ORDER_ITEMS');
    -- 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;
/
```

- b. Issue the following statements to view the metadata generated from the GET_TABLE_MD function:

```
set pagesize 0
set long 1000000

SELECT get_table_md FROM dual;
```

- c. Generate an XML representation of the ORDER_ITEMS table by using the DBMS_METADATA.GET_XML function. Spool the output to a file named ORDER_ITEMS_XML.txt in the D:\Labs folder.
 - d. Verify that the ORDER_ITEMS_XML.txt file in the D:\Labs folder is created.

Profiling and Tracing PL/SQL Code

12

ORACLE®

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

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



Objectives

In this lesson, you learn how to write PL/SQL routines that analyze the PL/SQL applications. You are introduced to tracing PL/SQL code and profiling PL/SQL code.

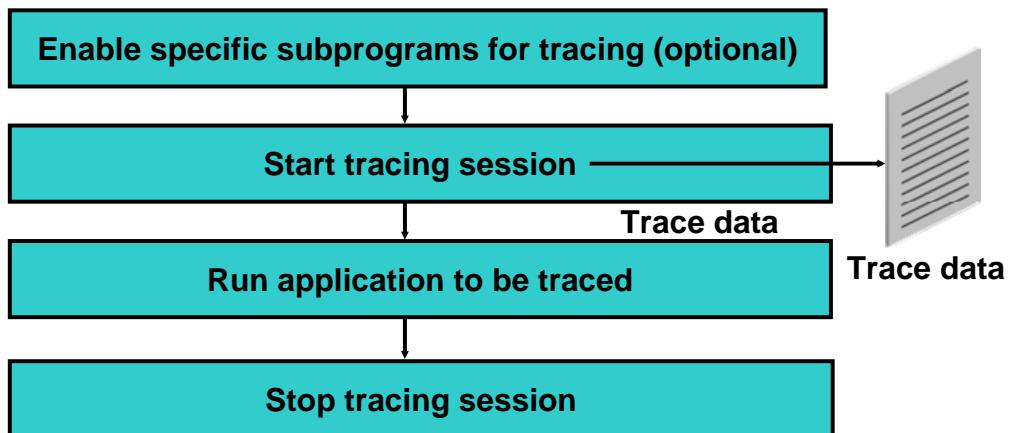
Lesson Agenda

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



Tracing PL/SQL Execution

Tracing PL/SQL execution provides you with a better understanding of the program execution path, and is possible by using the `dbms_trace` package.



Tracing PL/SQL Execution

In large and complex PL/SQL applications, it can sometimes become difficult to keep track of subprogram calls when a number of them call each other. By tracing your PL/SQL code, you can get a clearer idea of the paths and order in which your programs execute.

Though a facility to trace your SQL code has been around for a while, Oracle now provides an API for tracing the execution of PL/SQL programs on the server. You can use the Trace API, implemented on the server as the `dbms_trace` package, to trace PL/SQL subprogram code.

Note: You cannot use PL/SQL tracing with the multithreaded server (MTS).

Tracing PL/SQL Execution

The dbms_trace package contains:

- set_plsql_trace (*trace_level* INTEGER)
- clear_plsql_trace
- plsql_trace_version

ORACLE®

12 - 5

Copyright © 2008, Oracle. All rights reserved.

The dbms_trace Programs

dbms_trace provides subprograms to start and stop PL/SQL tracing in a session. The trace data is collected as the program executes, and it is written out to data dictionary tables.

Procedure	Description
set_plsql_trace	Starts tracing data dumping in a session (You provide the trace level at which you want your PL/SQL code traced as an IN parameter.)
clear_plsql_trace	Stops trace data dumping in a session
plsql_trace_version	Returns the version number of the trace package as an OUT parameter

A typical trace session involves:

- Enabling specific subprograms for trace data collection (optional)
- Starting the PL/SQL tracing session (dbms_trace.set_plsql_trace)
- Running the application that is to be traced
- Stopping the PL/SQL tracing session (dbms_trace.clear_plsql_trace)

Tracing PL/SQL Execution

- Using `set_plsql_trace`, select a trace level to identify how to trace calls, exceptions, SQL, and lines of code.
- Trace-level constants:
 - `trace_all_calls`
 - `trace_enabled_calls`
 - `trace_all_sql`
 - `trace_enabled_sql`
 - `trace_all_exceptions`
 - `trace_enabled_exceptions`
 - `trace_enabled_lines`
 - `trace_all_lines`
 - `trace_stop`
 - `trace_pause`
 - `trace_resume`

ORACLE®

12 - 6

Copyright © 2008, Oracle. All rights reserved.

Specifying a Trace Level

During the trace session, there are two levels that you can specify to trace calls, exceptions, SQL, and lines of code.

Trace Calls

- **Level 1:** Trace all calls. This corresponds to the constant `trace_all_calls`.
- **Level 2:** Trace calls only to enabled program units. This corresponds to the constant `trace_enabled_calls`.

Trace Exceptions

- **Level 1:** Trace all exceptions. This corresponds to `trace_all_exceptions`.
- **Level 2:** Trace exceptions raised only in enabled program units. This corresponds to `trace_enabled_exceptions`.

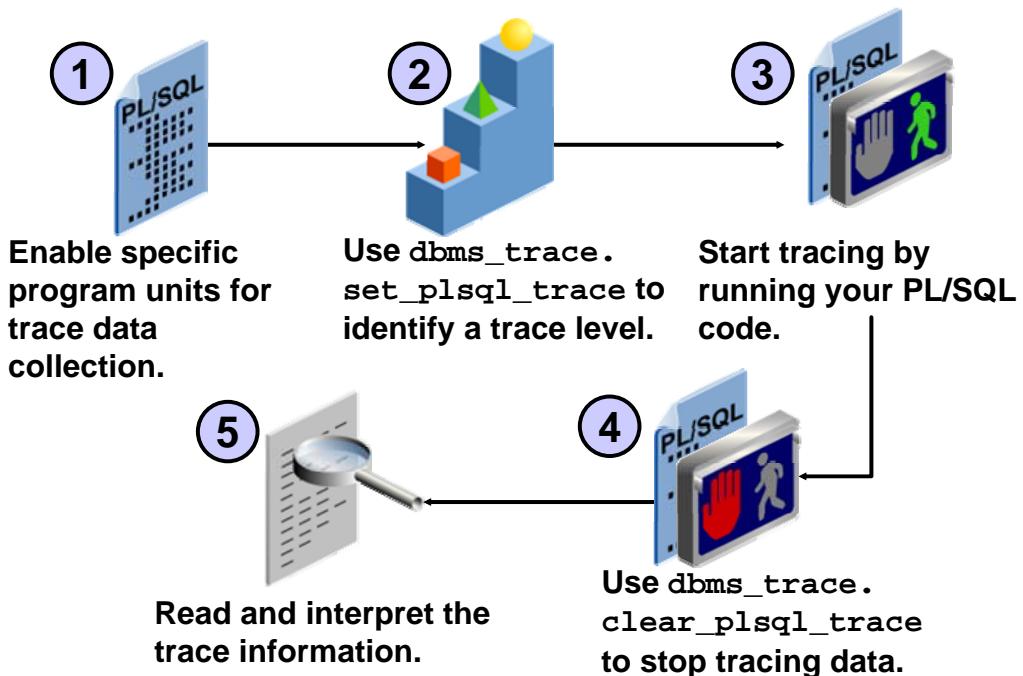
Trace SQL

- **Level 1:** Trace all SQL. This corresponds to the constant `trace_all_sql`.
- **Level 2:** Trace SQL in only enabled program units. This corresponds to the constant `trace_enabled_sql`.

Trace Lines

- **Level 1:** Trace all lines. This corresponds to the constant `trace_all_lines`.
- **Level 2:** Trace lines only in enabled program units. This corresponds to the constant `trace_enabled_lines`.

Tracing PL/SQL: Steps



ORACLE®

Steps to Trace PL/SQL Code

There are five steps to trace PL/SQL code using the `dbms_trace` package:

1. Enable specific program units for trace data collection.
2. Use `dbms_trace.set_plsql_trace` to identify a trace level.
3. Run your PL/SQL code.
4. Use `dbms_trace.clear_plsql_trace` to stop tracing data.
5. Read and interpret the trace information.

The next few pages demonstrate the steps to accomplish PL/SQL tracing.

Step 1: Enable Specific Subprograms

Enable specific subprograms with one of the two methods:

- Enable a subprogram by compiling it with the debug option:

```
ALTER SESSION SET PLSQL_DEBUG=true;
```

```
CREATE OR REPLACE ....
```

- Recompile a specific subprogram with the debug option:

```
ALTER [PROCEDURE | FUNCTION | PACKAGE]
      <subprogram-name> COMPILE DEBUG [BODY];
```

ORACLE®

12 - 8

Copyright © 2008, Oracle. All rights reserved.

Step 1: Enable Specific Subprograms

Profiling large applications may produce a huge volume of data that can be difficult to manage. Before turning on the trace facility, you have the option to control the volume of data collected by enabling a specific subprogram for trace data collection. You can enable a subprogram by compiling it with the debug option. You can do this in one of two ways:

- Enable a subprogram by compiling it with the ALTER SESSION debug option, and then compile the program unit by using the CREATE OR REPLACE syntax:

```
ALTER SESSION SET PLSQL_DEBUG = true;
CREATE OR REPLACE ...
```

- Recompile a specific subprogram with the debug option:

```
ALTER [PROCEDURE | FUNCTION | PACKAGE]
      <subprogram-name> COMPILE DEBUG [BODY];
ALTER PROCEDURE P5 COMPILE DEBUG;
```

Note: The second method cannot be used for anonymous blocks.

Enabling specific subprograms enables you to:

- Limit and control the amount of trace data, especially in large applications.
- Obtain additional trace information that is otherwise not available. For example, during the tracing session, if a subprogram calls another subprogram, the name of the called subprogram is included in the trace data if the calling subprogram was enabled by compiling it in debug mode.

Steps 2 and 3: Identify a Trace Level and Start Tracing

- Specify the trace level by using dbms_trace.set_plsql_trace:

```
EXECUTE DBMS_TRACE.SET_PLSQL_TRACE -
  (tracelevel1 + tracelevel2 ...)
```

- Execute the code that is to be traced:

```
EXECUTE my_program
```



Steps 2 and 3: Specify a Trace Level and Start Tracing

To trace PL/SQL code execution by using dbms_trace, perform these steps:

- Start the trace session by using the syntax shown in the slide. For example:

```
EXECUTE -
DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.trace_all_calls)
```

- Execute the PL/SQL code. The trace data is written to the data dictionary views.

Note: To specify additional trace levels in the argument, use the “+” symbol between each trace level value.

Step 4: Turn Off Tracing

Remember to turn tracing off by using the
`dbms_trace.clear_plsql_trace` procedure.

```
EXECUTE DBMS_TRACE.CLEAR_PLSQL_TRACE
```



Step 4: Turn Off Tracing

After tracing the PL/SQL program unit, turn tracing off by executing `dbms_trace.clear_plsql_trace`. This stops any further writing to the trace tables.

To avoid the overhead of writing the trace information, it is recommended that you turn tracing off when you are not using it.

Step 5: Examine the Trace Information

Examine the trace information:

- Call tracing writes out the program unit type, name, and stack depth.
- Exception tracing writes out the line number.



ORACLE®

12 - 11

Copyright © 2008, Oracle. All rights reserved.

Step 5: Examine the Trace Information

- Lower trace levels supersede higher levels when tracing is activated for multiple tracing levels.
- If tracing is requested only for enabled subprograms and if the current subprogram is not enabled, no trace data is written.
- If the current subprogram is enabled, call tracing writes out the subprogram type, name, and stack depth.
- If the current subprogram is not enabled, call tracing writes out the subprogram type, line number, and stack depth.
- Exception tracing writes out the line number. Raising the exception shows information about whether the exception is user-defined or predefined and, in the case of predefined exceptions, the exception number.

Note: An enabled subprogram is compiled with the debug option.

plsql_trace_runs and plsql_trace_events

- Trace information is written to the following dictionary views:
 - plsql_trace_runs dictionary view
 - plsql_trace_events dictionary view
- Run the `tracetab.sql` script to create the dictionary views.
- You need privileges to view the trace information in the dictionary views.



ORACLE®

12 - 12

Copyright © 2008, Oracle. All rights reserved.

The plsql_trace_runs and plsql_trace_events Dictionary Views

All trace information is written to the dictionary views `plsql_trace_runs` and `plsql_trace_events`. These views are created (typically by a DBA) by running the `tracetab.sql` script. The script is located in the `D:\app\Administrator\product\11.1.0\db_1\RDBMS\ADMIN` folder. Run the script as `SYS`. After the script is run, you need the `SELECT` privilege to view information from these dictionary views.

```
CONNECT /as sysdba
@D:\app\Administrator\product\11.1.0\db_1\RDBMS\ADMIN
\tracetab
GRANT SELECT ON plsql_trace_runs TO oe;
GRANT SELECT ON plsql_trace_events TO oe;
CONNECT oe
```

plsql_trace_runs and plsql_trace_events

```
ALTER SESSION SET PLSQL_DEBUG=TRUE;
ALTER PROCEDURE P5 COMPILE DEBUG;

EXECUTE DBMS_TRACE.SET_PLSQL_TRACE(DBMS_TRACE.trace_all_calls)
EXECUTE p5
EXECUTE DBMS_TRACE.CLEAR_PLSQL_TRACE

SELECT proc_name, proc_line,
       event_proc_name, event_comment
FROM sys.plsql_trace_events
WHERE event_proc_name = 'P5'
OR PROC_NAME = 'P5';
```

PROC_NAME	PROC_LINE	EVENT_PROC_NAME	EVENT_COMMENT
P5	1		Procedure Call
P4	1 P5		Procedure Call

2 rows selected.

ORACLE®

12 - 13

Copyright © 2008, Oracle. All rights reserved.

Query the plsql_trace_runs and plsql_trace_events Views

Use the dictionary views `plsql_trace_runs` and `plsql_trace_events` to view the trace information generated by using the `dbms_trace` facility. `plsql_trace_runs` holds generic information about traced programs, such as the date, time, owner, and name of the traced stored program. `dbms_trace_events` holds more specific information about the traced subprograms.

Lesson Agenda

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



Hierarchical Profiling Concepts

- Definition:
 - 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
- Benefits:
 - Provides more information than a flat profiler
 - Can be used to understand the structure and control flow of complex programs

ORACLE®

12 - 15

Copyright © 2008, Oracle. All rights reserved.

Hierarchical Profiling Concepts

Starting in Oracle Database 11g, the hierarchical profiler is available to help you identify the hotspots and performance tuning opportunities in your PL/SQL applications. This feature enables you to view reports of how a PL/SQL program is executed, organized by function call, as well as SQL and PL/SQL execution times.

You can view function level summaries that include:

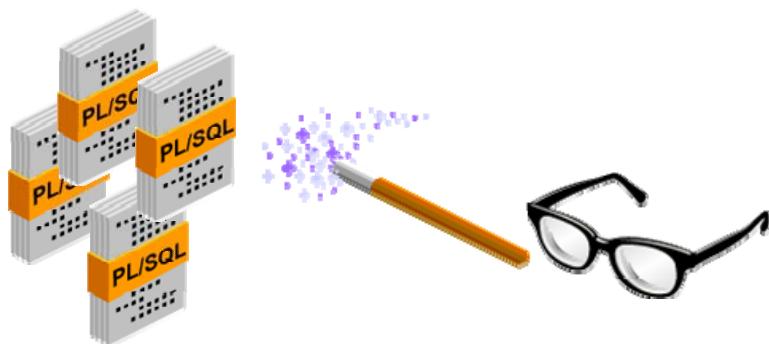
- Number of calls to a function
- Time spent in the function itself
- Time spent in the entire subtree under the function
- Detailed parent-children information for each function

You can use this information to tune your PL/SQL applications, and understand the structure, flow, and control of complex programs (especially those written by someone else).

Hierarchical Profiling Concepts

The PL/SQL hierarchical profiler consists of the:

- Data collection component
- Analyzer component



ORACLE®

12 - 16

Copyright © 2008, Oracle. All rights reserved.

Hierarchical Profiling Concepts (continued)

The PL/SQL hierarchical profiler consists of two subcomponents:

- The data collection component is an intrinsic part of the PL/SQL Virtual Machine. The new DBMS_HPROF package provides APIs to turn hierarchical profiling on and off. The raw profiler output is written to a file.
- The analyzer component, which is also exposed by the DBMS_HPROF package, is used to process the raw profiler output and upload the results of the profiling into database tables that can then be queried. You can use the plshprof command-line utility to generate simple HTML reports directly from the raw profiler data.

Using the PL/SQL Profiler

By using the PL/SQL profiler, you can find:

- The number of calls to a function
- The function time, not including descendants
- The subtree time, including descendants
- Parent-children information for each function
 - 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?

ORACLE®

12 - 17

Copyright © 2008, Oracle. All rights reserved.

Using the PL/SQL Profiler

By using the hierarchical PL/SQL profiler, you can find both function-level execution summary information, as well as detailed parent-children information for each function.

When profiling is turned on, function entry and exit operations are logged to a file along with time information. Fine-grained elapsed information is collected.

You do not need to recompile PL/SQL modules to use the hierarchical profiler. You can analyze both interpreted and natively compiled PL/SQL modules.

Using the PL/SQL Profiler

Sample data for profiling:

```
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
        ...
    END update_card_info;
    PROCEDURE display_card_info
        (p_cust_id NUMBER)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
        ...
    END display_card_info;
END credit_card_pkg; -- package body
```

ORACLE®

12 - 18

Copyright © 2008, Oracle. All rights reserved.

Sample Data

The CREDIT_CARD_PKG that was created earlier is used to demonstrate hierarchical profiling. The full code is shown on the following page.

Sample Data (continued)

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2);

    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
            UPDATE customers
                SET credit_cards = v_card_info
                WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
    END update_card_info;

    PROCEDURE display_card_info
        (p_cust_id NUMBER)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN
            FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
                DBMS_OUTPUT.PUT('Card Type: ' || v_card_info(idx).card_type
                                || '\n');
                DBMS_OUTPUT.PUT_LINE(' Card No: ' || v_card_info(idx).card_num );
            END LOOP;
        ELSE
            DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
        END IF;
    END display_card_info;
END credit_card_pkg; -- package body
/
```

Using the PL/SQL Profiler

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

1

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

2

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

3

ORACLE®

12 - 20

Copyright © 2008, Oracle. All rights reserved.

Steps for Using the PL/SQL Profiler

You use the new DBMS_HPROF package to hierarchically profile PL/SQL code. You need to be granted the privilege to execute the routines in the DBMS_HPROF package:

```
GRANT EXECUTE ON sys.dbms_hprof TO OE;
```

You also need to identify the location of the profiler files. Create a DIRECTORY object to identify this information:

```
CREATE DIRECTORY profile_data AS 'D:\labs\labs';
```

1. The first step is to turn profiling on. Use the DBMS_HPROF.START_PROFILING procedure. When calling this procedure, pass these two parameters:
 - Directory object: An alias for a file system path name. You need to have WRITE privileges to this location.
 - File name: The name of the file to which you want the output written.
 - You can optionally pass a third parameter, max_depth. When max_depth value is NULL (the default), profile information is gathered for all functions irrespective of their call depth. When a non-NULL value is specified, the profiler collects data only for functions up to a call depth level of max_depth.
2. The second step is to run the code that you want profiled.
3. The third step is to turn off profiling. Use the DBMS_HPROF.STOP_PROFILING procedure to stop the profiling.

```
EXECUTE DBMS_HPROF.STOP_PROFILING
```

Understanding Raw Profiler Data

```
P#! PL/SQL Timer Started
P#C PLSQL."""._plsql_vm"
P#X 3
P#C PLSQL."""._anonymous_block"
P#X 1634
P#C PLSQL."OE"."CREDIT_CARD_PKG":11."UPDATE_CARD_INFO">#71749359b90ac246
#24
P#X 7
P#C PLSQL."OE"."CREDIT_CARD_PKG":11."CUST_CARD_INFO">#c2ad85321cb9b0ae
#4
P#X 11
P#C SQL."OE"."CREDIT_CARD_PKG":11."_static_sql_exec_line10" #10
P#X 1502
P#R
...
P#C PLSQL."""._plsql_vm"
P#X 3
P#C PLSQL."""._anonymous_block"
P#X 15
P#C PLSQL."SYS"."DBMS_HPROF":11."STOP_PROFILING"#980980e97e42f8ec #53
P#R
P#R
P#! PL/SQL Timer Stopped
```

ORACLE®

12 - 21

Copyright © 2008, Oracle. All rights reserved.

Understanding Raw Profiler Data

You can examine the contents of the generated raw profiler text file. However, it is easier to understand the data after using the analyzer component of hierarchical profiling. The text shown in the slide displays the contents of the `pd_cc_pkg.txt` file. Each line starts with a file indicator. These are the meanings:

- P#V – PLSHPROF banner with version number
- P#C – Call to a subprogram (call event)
- P#R – Return from a subprogram (call event)
- P#X – Elapsed time between preceding and following events
- P#! – Comment

As you will see next, the `DBMS_HPROF.ANALYZE` function generates easier-to-decipher data and saves the data in tables.

Using the Hierarchical Profiler Tables

- Upload the raw profiler data into the database tables.
- Run the dbmshptab.sql script that is located in the /home/rdbms/admin folder to set up the profiler tables.

```
CONNECT OE
-- run this only once per schema
-- under the schema where you want the profiler tables located
@D:\app\Administrator\product\11.1.0\db_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



The Hierarchical Profiler Tables

Before uploading the profiler data into the database tables, you need to create the hierarchical profiler database tables. Run the script named dbmshptab.sql under the schema where you want the profiling tables to create the hierarchical profiling tables. This script is located in your /home/rdbms/admin/ folder.

The script creates three tables and other data structures that are required for persistently storing the profiler data.

Note: Running the script a second time drops any previously created hierarchical profiler tables.

Using DBMS_HPROF.ANALYZE

DBMS_HPROF.ANALYZE:

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

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

ORACLE®

12 - 23

Copyright © 2008, Oracle. All rights reserved.

Steps for Using the PL/SQL Profiler

Use the DBMS_HPROF.ANALYZE function to analyze the raw profiler output and produce the hierarchical profiler information in the database.

This function accepts the following parameters:

- Location: The name of the directory object that identifies the location from which to read.
- Filename: The file name of the raw profiler data to be analyzed.
- summary_mode: A Boolean that identifies whether to generate only top-level summary information into the database tables (TRUE) or to provide detailed analysis (FALSE). The default is false.
- Trace: Identifies whether to analyze only the subtrees rooted at the specified trace entry or perform the analysis for the entire run. This parameter is specified in a special, qualified format within quotation marks. It includes the schema name, module name, and function name. For example "SCOTT" . " "PKG" . "FOO".
- Skip: Analyzes only the subtrees rooted at the specified trace, but ignores the first "skip" invocations to trace. The default value for "skip" is 0. Used only when trace is specified.
- Collect: Analyzes "collect" member of invocations of traces (starting from "skip" + 1). By default, only one invocation is collected. Used only when trace is specified.
- run_comment: A comment for your run.

Using DBMS_HPROF.ANALYZE to Write to Hierarchical Profiler Tables

- Use the DBMS_HPROF.ANALYZE function to upload the raw profiler results into the database tables.

```
DECLARE
  v_runid NUMBER;
BEGIN
  v_runid := DBMS_HPROF.ANALYZE (LOCATION => 'PROFILE_DATA',
                                    FILENAME => 'pd_cc_pkg.txt');
  DBMS_OUTPUT.PUT_LINE('Run ID: ' || v_runid);
END;
```

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

RUN_ID

1

ORACLE®

Steps for Using the PL/SQL Profiler (continued)

In the example shown in the slide, the DBMS_HPROF.ANALYZE function is used to take the raw data from the text file named pd_cc_pkg.txt and place the resulting analyzed data in the profiler tables. The function returns a unique identifier for the run.

Sample Analyzer Output from the DBMSHP_RUNS Table

Query the DBMSHP_RUNS table to find top-level information for each run:

```
SELECT runid, run_timestamp, total_elapsed_time  
FROM dbmshp_runs  
WHERE runid = 1;
```

RUNID	RUN_TIMESTAMP	TOTAL_ELAPSED_TIME
2	12-DEC-07 12.26.24.984000 PM	122650

ORACLE®

12 - 25

Copyright © 2008, Oracle. All rights reserved.

Query the DBMSHP_RUNS Table

The DBMS_HPROF.ANALYZE PL/SQL API is used to analyze the raw profiler output and upload the results into the database tables. This enables you to query the data for custom report generation using tools, such as SQL Developer or other third party tools.

In the example shown in the slide, the DBMSHP_RUNS table is queried after the DBMS_HPROF.ANALYZE is run. This table contains top-level information for each run of the DBMS_HPROF.ANALYZE command. From the output, you can see the run ID, the timestamp of the run, and the total elapsed time in milliseconds. If you provided a comment for the run, you can retrieve that information from the RUN_COMMENT column (not shown).

Sample 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;
```

OWNER	MODULE	TYPE	LINE#	NAMESPACE	CALLS	TIME
		_anonymous_block		PLSQL	2	1650
		_plsql_vm		PLSQL	2	10
OE	CREDIT_CARD PACKAGE BODY	CUST_CARD_INFO		PLSQL	1	66
OE	CREDIT_CARD PACKAGE BODY	UPDATE_CARD_INFO		PLSQL	1	38
SYS	DBMS_HPROF PACKAGE BODY	STOP_PROFILING		PLSQL	1	0
OE	CREDIT_CARD PACKAGE BODY	_static_sql_exec_line10	SQL		1	1502
OE	CREDIT_CARD PACKAGE BODY	_static_sql_exec_line41	SQL		1	8999
		_PKG				
		_PKG				

ORACLE®

Query the DBMSHP_FUNCTION_INFO Table

You can query the DBMSHP_FUNCTION_INFO table to find summary information for each function profiled in a run of the analyzer.

In the query shown above, the following information is retrieved:

- OWNER: Module owner of the function
- MODULE: The module name
- TYPE: Module type, such as package, package body, procedure, and so on
- LINE#: The line number in the module at which the function is defined. This line# helps identify the source location of the function in the module and can be used by the integrated development environment (IDE) tools to navigate to the appropriate location in the source where the function is defined. The line number can also be used to distinguish between overloaded routines.
- NAMESPACE: The language information. At this point in time, SQL and PL/SQL are supported.
- CALLS: The number of calls to the function
- FUNCTION_ELAPSED_TIME: The time in microseconds, not including the time spent in descendant functions

plshprof: A Simple HTML Report Generator

- `plshprof` is a command-line utility.
- You can use it to generate simple HTML reports directly from the raw profiler data.
- The HTML reports can be browsed in any browser.
- The navigational capabilities combined with the links provide a means for you to analyze the performance of large applications.



ORACLE®

12 - 27

Copyright © 2008, Oracle. All rights reserved.

Introducing `plshprof`

`plshprof` is a command-line utility that you can use to generate HTML reports based on the raw profiler data generated by the data collection component after running the DBMS_HPROF.ANALYZE PL/SQL API.

Using plshprof

After generating the raw profiler output file:

1. Change to the directory where you want the HTML output placed.
2. Run plshprof.
 - Syntax:

```
plshprof [option...] output_filename_1 output_filename_2
```

- Example:

```
D:\labs\labs>plshprof -output pd_cc_pkg pd_cc_pkg.txt
PLSHPROF: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Product
[6 symbols processed]
Report written to 'pd_cc_pkg.html'
D:\labs\labs>
```

ORACLE®

12 - 28

Copyright © 2008, Oracle. All rights reserved.

Using plshprof

After you have the raw profiler output file, you can generate HTML reports for that run.

The plshprof utility has these options:

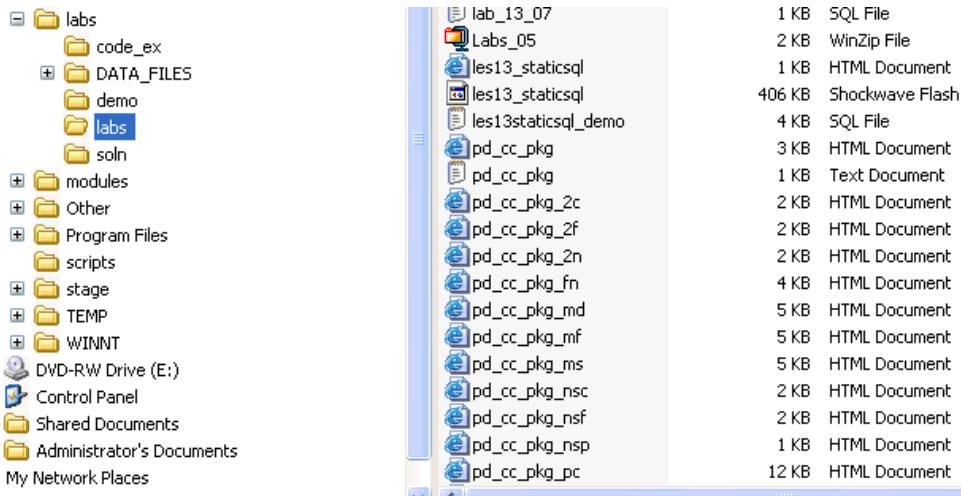
- *-skip count*: Skips the first count calls. Use only with the *-trace symbol*. The default is 0.
- *-collect count*: Collects information for count calls. Use only with *-trace symbol*. The default is 1.
- *-output filename*: Specifies the name of the output file *symbol.html* or *tracefile1.html*
- *-summary*: Prints only elapsed time. No default.
- *-trace symbol*: Specifies the function name of the tree root

Follow these steps to generate the HTML reports:

1. Change to the directory of the raw profiler output file. In the example shown in the slide, the raw profiler file exists in the \Labs\labs folder. Its name is *pd_dd_pkg.txt*.
2. Run the plshprof utility. This utility accepts line arguments. The slide example identifies that the output file should be named *pd_cc_pkg.html* and the original raw profiler file is named *pd_dd_pkg.txt*.

Using plshprof

Resulting generated files:



Using plshprof (continued)

Running the `plshprof` utility generates a set of HTML files. The `filename.html` is the root page where you start browsing. The other files with the same file name prefix are hyperlinks from the base `filename.html` file. In the slide example, the root page is named `pd_cc_pkg.html`.

Using plshprof

3. Open the filename.html in a browser:

The screenshot shows a web browser window with the following details:

- Address Bar:** D:\labs\labs\pd_cc_pkg.html
- Title:** PL/SQL Elapsed Time (microsecs) Analysis
- Text:** 73605 microsecs (elapsed time) & 8 function calls
- Description:** 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 are 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 \(microsecs\)](#)

- 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 \(microsecs\)](#)

ORACLE®

12 - 30

Copyright © 2008, Oracle. All rights reserved.

Using plshprof (continued)

3. You can open the start page file in any browser. In this example, the pd_cc_pkg.html file is opened. This is an example of a single-run report. It contains some overall summary information and hyperlinks to additional information.

Using the HTML Reports

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)

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

12265 microsecs (elapsed time) & 9 function calls

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name
12265	100%		10	0.1%	12255	100%	2	22.2% plsql_vm
12255	100%		1650	13.5%	10605	86.5%	2	22.2% anonymous block
10605	86.5%		38	0.3%	10567	86.2%	1	11.1% OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO (Line 24)
8999	73.4%		8999	73.4%	0	0.0%	1	11.1% OE.CREDIT_CARD_PKG.static_sql_exec_line41 (Line 41)
1568	12.8%		66	0.5%	1502	12.2%	1	11.1% OE.CREDIT_CARD_PKG.CUST_CARD_INFO (Line 4)
1502	12.2%		1502	12.2%	0	0.0%	1	11.1% OE.CREDIT_CARD_PKG.static_sql_exec_line10 (Line 10)
0	0.0%		0	0.0%	0	0.0%	1	11.1% SYS.DBMS_HPROF.STOP_PROFILING (Line 53)

ORACLE®

Viewing the Reports

The function-level summary reports provide you with a flat view of the profile information. The reports detail the total number of calls, self time, subtree time, and descendants for each function. Each report is sorted on a particular attribute.

Note that in the slide example, each function name is hyperlinked to its corresponding parents and children report. The column that is in bold identifies how the report is sorted.

If a subprogram is nested, the profiler reports display the fully qualified name, such as OE.P.A.B; that is, procedure B is nested within procedure A of package OE.P.

Using the HTML Reports

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)

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

12265 microsecs (elapsed time) & 9 function calls

Subtree	Ind%	Function	Ind%	Cum%	Descendants	Ind%	Calls	Ind%	Function Name
8999	73.4%	8999	73.4%	73.4%	0	0.0%	1	11.1%	OE.CREDIT_CARD.PKG.static_sql_exec_line41 (Line 41)
12255	100%	1650	13.5%	86.8%	10605	86.5%	2	22.2%	anonymous block
1502	12.2%	1502	12.2%	99.1%	0	0.0%	1	11.1%	OE.CREDIT_CARD.PKG.static_sql_exec_line10 (Line 10)
1568	12.8%	66	0.5%	99.6%	1502	12.2%	1	11.1%	OE.CREDIT_CARD.PKG.CUST_CARD_INFO (Line 4)
10605	86.5%	38	0.3%	100%	10567	86.2%	1	11.1%	OE.CREDIT_CARD.PKG.UPDATE_CARD_INFO (Line 24)
12265	100%	10	0.1%	100%	12255	100%	2	22.2%	_plsql_vm
0	0.0%	0	0.0%	100%	0	0.0%	1	11.1%	SYS.DBMS_HPROF.STOP_PROFILING (Line 53)

ORACLE®

12 - 32

Copyright © 2008, Oracle. All rights reserved.

Viewing the Reports (continued)

You can use the module level summary report to find information about each PL/SQL module, including the total time spent in the module, the total number of calls to functions in the module, and the total number of calls to functions in the module. The total time spent in a module is calculated by adding the self times for all functions in that module.

Using the HTML Reports

- [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](#)



Namespace Elapsed Time (microsecs) Data sorted by Namespace				
12265 microsecs (elapsed time) & 9 function calls				
Function	Ind%	Calls	Ind%	Namespace
1764	14.4%	7	77.8%	PLSQL
10501	85.6%	2	22.2%	SQL

ORACLE®

Viewing the Reports (continued)

Each function tracked by the profiler is associated with a namespace. PL/SQL functions, procedures, triggers, and packages fall under the category “PLSQL” namespace. Operations corresponding to SQL statement execution from PL/SQL, such as UPDATE, SELECT, FETCH, EXECUTE IMMEDIATE, and so on, fall under the “SQL” namespace. The namespace level summary report provides information about the total time spent in that namespace and the total number of calls to functions in that namespace.

Using the HTML Reports

The screenshot shows the Oracle Profiler's HTML Reports interface. At the top, there is a navigation bar with three items:

- Namespace Elapsed Time (microsecs) Data sorted by Namespace
- Namespace Elapsed Time (microsecs) Data sorted by Total Function Call Count
- Parents and Children Elapsed Time (microsecs) Data

A red box highlights the third item, "Parents and Children Elapsed Time (microsecs) Data". A red arrow points from this box down to the main content area.

The main content area displays two execution profiles:

anonymous_block

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name
12255	100%	1650	13.5%	10605	86.5%	2	22.2%	anonymous block

Parents:

12255	100%	1650	100%	10605	100%	2	100%	plsql vm
-------	------	------	------	-------	------	---	------	----------

Children:

10605	100%	38	100%	10567	100%	1	100%	OE.CREDIT_CARD_PKG.UPDATE_CARD_INFO
0	0.0%	0	N/A	0	N/A	1	100%	SYS.DBMS_HPROF.STOP_PROFILING (Line 53)

plsql_vm

Subtree	Ind%	Function	Ind%	Descendants	Ind%	Calls	Ind%	Function Name
12265	100%	10	0.1%	12255	100%	2	22.2%	plsql vm

Parents:

12265	100%	10	100%	12255	100%	2	100%	ORACLE.root
-------	------	----	------	-------	------	---	------	-------------

Children:

12255	100%	1650	100%	10605	100%	2	100%	anonymous block
-------	------	------	------	-------	------	---	------	-----------------

OE.CREDIT_CARD_PKG.CUST_CARD_INFO (Line 4)

ORACLE

Viewing the Reports (continued)

For each function tracked by the profiler, the parents and children report provides information about what functions call it (parents) and what functions it calls (children). It gives you the function's execution profile on a per-parent basis. It also provides the execution profile for each of the function's children when called from this function.

Summary

In this lesson, you should have learned how to:

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



Summary

In this lesson, you learned how to trace PL/SQL program execution by using the DBMS_TRACE package and you learned to profile your PL/SQL application. With profiling, you learned how to set up the profiler tables, generate raw data, and use DBMS_HPROF to analyze raw data to the profiler tables.

Practice 12: Overview

This practice covers the following topics:

- Hierarchical profiling of PL/SQL code



Practice 12: Overview

Using the OE application that you have created, write code to profile components in your application.

Use the OE schema for this practice.

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”

Practice 12

Profiling

1. Generate profiling data for your CREDIT_CARD_PKG.
 - a. Re-create CREDIT_CARD_PKG by running the D:\labs\labs\lab_12_01.sql script.
 - b. You need to identify the location of the profiler files. Create a DIRECTORY object to identify this information.

```
CREATE DIRECTORY profile_data AS 'D:\Labs\labs';
```
 - c. Use DBMS_HPROF.START_PROFILING to start the profiler for your session.
 - d. Run your CREDIT_CARD_PKG.UPDATE_CARD_INFO with the following data.

```
credit_card_pkg.update_card_info
(154, 'Discover', '123456789');
```
 - e. Use DBMS_HPROF.STOP_PROFILING to stop the profiler.

```
@D:\app\Administrator\product\11.1.0\db_1\RDBMS\ADMIN\dbmshptab.sql
```
2. Run the dbmshptab.sql script that is located in the \home\rdbms\admin folder to set up the profiler tables.
3. Use DBMS_HPROF.ANALYZE to analyze the raw data and write the information to the profiler tables.
 - a. Get the RUN_ID.
 - b. Query the DBMSHP_RUNS table to find top-level information for the RUN_ID that you retrieved.
 - c. Query the DBMSHP_FUNCTION_INFO table to find information about each function profiled.
4. Use the plshprof command-line utility to generate simple HTML reports directly from the raw profiler data.
 - a. Open a command window.
 - b. Change the working directory to D:\labs\labs
 - c. Run the plshprof utility.
5. Open the report in your browser and review the data.

Safeguarding Your Code Against SQL Injection Attacks

13

ORACLE®

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

- Describe SQL injections
- Reduce attack surfaces
- Use DBMS_ASSERT
- Design immune code
- Test code for SQL injection flaws



Objectives

In this lesson, you learn how to use techniques and tools to strengthen your code and applications against SQL injection attacks.

Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT
- Designing code immune to SQL injections
- Testing code for SQL injection flaws



Understanding SQL Injection

SQL injection is a technique for maliciously exploiting applications that use client-supplied data in SQL statements.

- Attackers trick the SQL engine into executing unintended commands.
- SQL injection techniques may differ, but they all exploit a single vulnerability in the application.
- To immunize your code against SQL injection attacks, use bind arguments or validate and sanitize all input concatenated to dynamic SQL.



ORACLE®

13 - 4

Copyright © 2008, Oracle. All rights reserved.

Understanding SQL Injection

SQL injection is a technique for maliciously exploiting applications that use client-supplied data in SQL statements.

Attackers trick the SQL engine into executing unintended commands via supplying specially crafted string input, thereby gaining unauthorized access to a database in order to view or manipulate restricted data.

SQL injection techniques may differ, but they all exploit a single vulnerability in the application.

String literals that are incorrectly validated or not validated are concatenated into a dynamic SQL statement and interpreted as code by the SQL engine.

To immunize your code against SQL injection attacks, you must use bind arguments (either automatically with static SQL, or explicitly with dynamic SQL), or validate and sanitize all input concatenated to dynamic SQL.

Although a program or an application may be vulnerable to SQL injection, Web applications are at a higher risk, because an attacker can perpetrate SQL injection attacks without any database or application authentication.

Identifying Types of SQL Injection Attacks

Category	Description
First Order Attack	The attacker can simply enter a malicious string and cause the modified code to be executed immediately.
Second Order Attack	The attacker injects into persistent storage (such as a table row), which is deemed a trusted source. An attack is subsequently executed by another activity.

ORACLE®

13 - 5

Copyright © 2008, Oracle. All rights reserved.

Types of SQL Injection Attacks

The impact of SQL injection attacks may vary from gathering of sensitive data to manipulating database information, and from executing system-level commands to denial of service of the application. The impact also depends on the database on the target machine, and the roles and privileges that the SQL statement is running with.

Researchers generally divide injection attacks into two categories:

- First Order Attack
- Second Order Attack

SQL injection attacks do not have to return data directly to the user to be useful. “Blind” attacks (for example, that create a database user, but otherwise return no data) can still be very useful to an attacker.

In addition, hackers are known to use timing or other performance indicators, and even error messages in order to deduce the success or results of an attack.

SQL Injection: Example

```
-- First order attack
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;
```

String literals that are incorrectly validated or not validated are concatenated into a dynamic SQL statement, and interpreted as code by the SQL engine.

ORACLE®

13 - 6

Copyright © 2008, Oracle. All rights reserved.

First Order SQL Injection Attack: Example

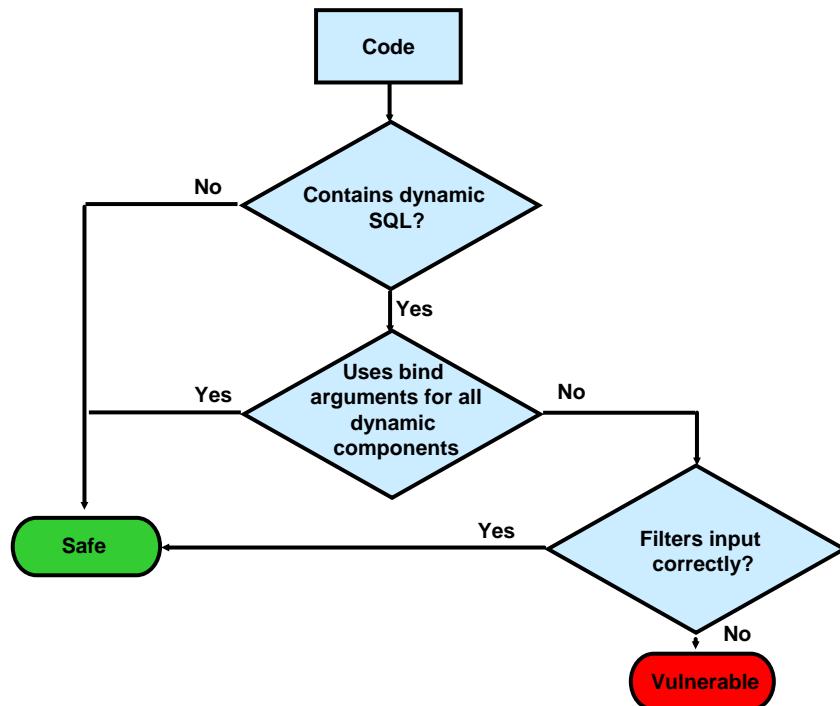
The example shown in the slide demonstrates a procedure with dynamic SQL constructed via concatenation of input value. This is vulnerable to SQL injection.

```
EXECUTE get_email('Andrews')
SQL statement: SELECT cust_email FROM customers WHERE
cust_last_name = 'Andrews'
Email: Ajay.Andrews@YELLOWTHROAT.COM
Email: Dianne.Andrews@TURNSTONE.COM
```

PL/SQL procedure successfully completed.

```
EXECUTE get_email('x' union select username from all_users
where ''x''='x')
SQL statement: SELECT cust_email FROM customers WHERE
cust_last_name = 'x' union
select username from all_users where 'x'='x'
Email: ANONYMOUS
Email: APEX_PUBLIC_USER
Email: BI
Email: CTXSYS
...
```

Assessing Vulnerability



ORACLE®

13 - 7

Copyright © 2008, Oracle. All rights reserved.

Assessing Vulnerability

You need to analyze your code to determine SQL injection vulnerability. The flowchart shows you how to start assessing for vulnerability.

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. In case your filtering requirements cannot be satisfied by the DBMS_ASSERT package, you may need to create your own filter.

ORACLE®

Avoidance Strategies Against SQL Injection

You can use several avoidance strategies to safeguard against, or mitigate the impact of, SQL injection attacks. Listed in the slide are high-level descriptions of each of the strategies that are examined in more detail on subsequent pages.

The available and best method(s) for eliminating SQL injection vulnerability may depend on the vulnerability itself. Not all methods are available for addressing every vulnerability.

Methods

- Use static SQL:
 - If all Oracle identifiers (for example, column, table, view, trigger, program unit, or schema names) are known at code compilation time

Note: Static SQL automatically binds arguments. Data definition language (DDL) statements cannot be executed with static SQL.

- Use dynamic SQL with bind arguments:
 - If any WHERE clause values, VALUES clause values, or SET clause values are unknown, and any Oracle identifiers are unknown at code compilation time
- Validate and sanitize input:
 - If concatenating any strings and if bind arguments require additional filtering

Protecting Against SQL Injection: Example

```
CREATE OR REPLACE PROCEDURE GET_EMAIL
  (p_last_name VARCHAR2 DEFAULT NULL)
AS
BEGIN
  FOR i IN
    (SELECT cust_email
     FROM customers
     WHERE cust_last_name = p_last_name)
  LOOP
    DBMS_OUTPUT.PUT_LINE('Email: '||i.cust_email);
  END LOOP;
END;
```

This example avoids dynamic SQL with concatenated input values.

```
EXECUTE get_email('Andrews')
Email: Ajay.Andrews@YELLOWTHROAT.COM
Email: Dianne.Andrews@TURNSTONE.COM

PL/SQL procedure successfully completed.

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

PL/SQL procedure successfully completed.
```

ORACLE®

13 - 9

Copyright © 2008, Oracle. All rights reserved.

First Order SQL Injection Attack: Example

The example in the slide with static SQL is protected against SQL injection.

Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT
- Designing code immune to SQL injections
- Testing code for SQL injection flaws

ORACLE®

13 - 10

Copyright © 2008, Oracle. All rights reserved.

Reducing the Attack Surface

Using invoker's rights:

- Using invoker's rights helps to limit the privileges
- Helps to minimize the security exposure

```
-- this example does not use Invoker's rights
CONNECT /as sysdba

CREATE OR REPLACE
PROCEDURE change_password(p_username VARCHAR2 DEFAULT NULL,
                           p_new_password VARCHAR2 DEFAULT NULL)
IS
    v_sql_stmt VARCHAR2(500);
BEGIN
    v_sql_stmt := 'ALTER USER '||p_username||' IDENTIFIED BY '
                  || p_new_password;
    EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
```

Note the use of dynamic SQL with concatenated input values.

```
GRANT EXECUTE ON change_password to OE, HR, SH;
```

1

2

ORACLE

13 - 11

Copyright © 2008, Oracle. All rights reserved.

Reducing the Attack Surface

If you do not provide an interface to an attacker, clearly it is not available to be abused. Thus the first, and arguably most important, line of your defense is to reduce the exposed interfaces to only those that are absolutely required. You can reduce the exposed interfaces by:

- Using invoker's rights to reduce SQL injection vulnerability
- Reducing arbitrary inputs

Using Invoker's Rights

Stored program units and SQL methods execute with a set of privileges. By default, the privileges are those of a schema owner, also known as the definer. The definer's rights not only dictate the privileges, they are also used to resolve object references. If a program unit does not need to be executed with the escalated privileges of the definer, you should specify that the program unit executes with the privileges of a caller, also known as the invoker.

1. The example shown in the slide uses definer's rights. Procedure CHANGE_PASSWORD is created under the SYS schema. It accepts two parameters and uses them in the ALTER USER statement.
2. SYS grants OE, HR, and SH the ability to execute the CHANGE_PASSWORD procedure.

Using Invoker's Rights

- OE is successful at changing the SYS password, because, by default, CHANGE_PASSWORD executes with SYS privileges:

```
CONNECT oe

EXECUTE sys.change_password ('SYS', 'mine')

PL/SQL procedure successfully completed.
```

- Add the AUTHID to change the privileges to the invokers:

```
CONNECT /as sysdba
CREATE OR REPLACE
PROCEDURE change_password(p_username VARCHAR2 DEFAULT NULL,
                           p_new_password VARCHAR2 DEFAULT NULL)
AUTHID CURRENT_USER
IS
  v_sql_stmt VARCHAR2(500);
BEGIN
  v_sql_stmt := 'ALTER USER '||p_username||' IDENTIFIED BY '
                || p_new_password;
  EXECUTE IMMEDIATE v_sql_stmt;
END change_password;
```

ORACLE

13 - 12

Copyright © 2008, Oracle. All rights reserved.

Using Invoker's Rights

When OE executes the CHANGE_PASSWORD procedure, it is executed under SYS privileges (the definer of the procedure), and with the code shown, OE can change the SYS password. Obviously, this is an unacceptable outcome.

To disallow another schema from changing a password that does not belong to the schema, redefine the procedure with the invoker's rights. This is done with the AUTHID CURRENT_USER option.

```
CONNECT oe

EXECUTE change_password ('SYS', 'mine')

ERROR at line 1:
ORA-01031: Insufficient privileges
ORA-06512: at "SYS.CHANGE_PASSWORD", at line 1
ORA-06512: at line 1
```

Now OE can no longer change the SYS (or any other account) password.

Notice that the CHANGE_PASSWORD procedure contains dynamic SQL with concatenated input values. This is a SQL injection vulnerability. Although using invoker's rights does not guarantee the elimination of SQL injection risks, it can help mitigate the exposure.

Reducing Arbitrary Inputs

- Reduce the end-user interfaces to only those that are actually needed.
 - In a Web application, restrict users to accessing specified Web pages.
 - In a PL/SQL API, expose only those routines that are intended for customer use.
- Where user input is required, make use of language features to ensure that only data of the intended type can be specified.
 - Do not specify a VARCHAR2 parameter when it will be used as a number.
 - Do not use numbers if you need only positive integers; use natural instead.

ORACLE®

13 - 13

Copyright © 2008, Oracle. All rights reserved.

Reducing the Arbitrary Inputs

Because a SQL injection attack is possible only if user input is allowed, one key measure of prevention is to limit user input.

First, you must reduce the end-user interfaces to only those actually needed. For example:

- In a Web application, restrict users to accessing specified Web pages.
- In a PL/SQL API, expose only those routines that are intended for customer use.

Remove any debug, test, deprecated, or other unnecessary interfaces. They add nothing to product functionality, but do provide an attacker with more ways to target your application.

From Oracle Database 10.2 and later, you can use PL/SQL conditional compilation for managing self-tracing code, asserts, and so on.

Second, where user input is required, make use of language features to ensure that only data of the intended type can be specified. For example:

- Do not specify a VARCHAR2 parameter when it will be used as a number.
- Do not use numbers if you need only positive integers, use natural instead.

Careful selection of parameter types to an API can considerably reduce the scope for attack, and make it a lot easier for customers to use.

Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT
- Designing code immune to SQL injections
- Testing code for SQL injection flaws



Using Static SQL

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

ORACLE®

13 - 15

Copyright © 2008, Oracle. All rights reserved.

Using Static SQL

Because SQL injection is a feature of SQL statements that are dynamically constructed via user inputs, it follows that designing your application to be based on static SQL reduces the scope for attack.

There are two common situations where developers often use dynamic SQL, when static SQL would serve the purpose and be more secure:

- Handling a varying number of IN-list values in the query condition
- Handling the LIKE comparison operator in the query condition

Using Static SQL

```
CREATE OR REPLACE PROCEDURE list_products_dynamic
(p_product_name VARCHAR2 DEFAULT NULL)
AS
    TYPE cv_prodtyp IS REF CURSOR;
    cv_cv_prodtyp;
    v_prodname product_information.product_name%TYPE;
    v_minprice product_information.min_price%TYPE;
    v_listprice product_information.list_price%TYPE;
    v_stmt VARCHAR2(400);
BEGIN
    v_stmt := 'SELECT product_name, min_price, list_price
              FROM product_information WHERE product_name LIKE
              ''%'||p_product_name||'%''';
    OPEN cv FOR v_stmt;
    dbms_output.put_line(v_stmt);
    LOOP
        FETCH cv INTO v_prodname, v_minprice, v_listprice;
        EXIT WHEN cv%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Product Info: '||v_prodname ||', ' ||
                             v_minprice ||', '|| v_listprice);
    END LOOP;
    CLOSE cv;
END;
```

You can convert this statement to static SQL.

ORACLE®

13 - 16

Copyright © 2008, Oracle. All rights reserved.

Converting Dynamic SQL to Static SQL

By converting the dynamic SQL static shown above, you can decrease your attack vulnerability. The code above uses dynamic SQL to handle the LIKE operator in the query condition. Notice that string concatenation is used to build the SQL statement.

Examine the execution of this injection:

```
EXECUTE list_products_dynamic('' and 1=0 union select
                                cast(username as varchar2(100)), null, null
                                from all_users --')

SELECT product_name, min_price, list_price FROM
product_information WHERE
product_name like '%' and 1=0 union select cast(username as
varchar2(100)),
null, null from all_users --%
Product Info: ANONYMOUS,
Product Info: APEX_PUBLIC_USER,
Product Info: BI,
...
```

Notice that the injection succeeded through the concatenation of the UNION set operator to the dynamic SQL statement.

Using Static SQL

- To use static SQL, accept the user input, and then concatenate the necessary string to a local variable.
- Pass the local variable to the static SQL statement.

```
CREATE OR REPLACE PROCEDURE list_products_static
  (p_product_name VARCHAR2 DEFAULT NULL)
AS
  v_bind  VARCHAR2(400);
BEGIN
  v_bind := '%' || p_product_name || '%';
  FOR i IN
    (SELECT product_name, min_price, list_price
     FROM product_information
     WHERE product_name like v_bind)
  LOOP
    DBMS_OUTPUT.PUT_LINE('Product Info: ' || i.product_name || ','
      || i.min_price || ', ' || i.list_price);
  END LOOP;
END list_products_static;
```

ORACLE®

13 - 17

Copyright © 2008, Oracle. All rights reserved.

Converting Dynamic SQL to Static SQL (continued)

In the example shown above, you can avoid SQL injection by concatenating the user input and placing it into a local variable, and then using the local variable within the static SQL.

Examine the results:

```
-- desired results - normal execution
EXECUTE list_products_static('Laptop')
Product Info: Laptop 128/12/56/v90/110, 2606, 3219
Product Info: Laptop 16/8/110, 800, 999
Product Info: Laptop 32/10/56, 1542, 1749
Product Info: Laptop 48/10/56/110, 2073, 2556
Product Info: Laptop 64/10/56/220, 2275, 2768

PL/SQL procedure successfully completed.

-- this example attempts injection
EXECUTE list_products_static(''' and 1=0 union select
cast(username as varchar2(100)), null, null from all_users -- '
)

PL/SQL procedure successfully completed.
```

Using Dynamic SQL

- Dynamic SQL may be unavoidable in the following types of situations:
 - You do not know the full text of the SQL statements that must be executed in a PL/SQL procedure.
 - You want to execute DDL statements and other SQL statements that are not supported in purely static SQL programs.
 - You want to write a program that can handle changes in data definitions without the need to recompile.
- If you must use dynamic SQL, try not to construct it through concatenation of input values. Instead, use bind arguments.

ORACLE®

13 - 18

Copyright © 2008, Oracle. All rights reserved.

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. For example, a SELECT statement that includes an identifier (such as table name) that is unknown at compile time or a WHERE clause in which the number of subclauses is unknown at compile time.
 - You want to execute DDL statements and other SQL statements that are not supported in purely static SQL programs.
 - You want to write a program that can handle changes in data definitions without the need to recompile.
- If you must use dynamic SQL, try not to construct it through concatenation of input values. Instead, use bind arguments.

If you cannot avoid input concatenation, you must validate the input values, and also consider constraining user input to a predefined list of values, preferably numeric values. Input filtering and sanitizing are covered in more detail later in this lesson.

Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- **Using bind arguments**
- Filtering input with DBMS_ASSERT
- Designing code immune to SQL injections
- Testing code for SQL injection flaws

ORACLE®

13 - 19

Copyright © 2008, Oracle. All rights reserved.

Using Bind Arguments with Dynamic SQL

You can rewrite the following statement

```
v_stmt :=  
  'SELECT'||filter(p_column_list)||' FROM customers'||  
  'WHERE account_mgr_id = '''|| p_sales_rep_id ||''';  
  
EXECUTE IMMEDIATE v_stmt;
```

as this 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

You can use bind arguments in the WHERE clause, the VALUES clause, or the SET clause of any SQL statement, as long as the bind arguments are not used as Oracle identifiers (such as column names or table names) or keywords.

Developers often use dynamic SQL to handle a varying number of IN-list values or LIKE comparison operators in the query condition.

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 this 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;
```

ORACLE®

13 - 21

Copyright © 2008, Oracle. All rights reserved.

Using Bind Arguments with Dynamic PL/SQL

As with dynamic SQL, you should avoid constructing dynamic PL/SQL with string concatenation. The impact of SQL injection vulnerabilities in dynamic PL/SQL is more serious than in dynamic SQL, because, with dynamic PL/SQL, multiple statements (such as DELETE or DROP) can be batched together and injected.

Working Around Bind Arguments

Though you should strive to use bind arguments with all dynamic SQL and PL/SQL statements, there are instances where bind arguments cannot be used:

- DDL statements (such as CREATE, DROP, and ALTER)
- Oracle identifiers (such as names of columns, tables, schemas, database links, packages, procedures, and functions)

If bind arguments cannot be used with dynamic SQL or PL/SQL, you must filter and sanitize all input concatenated to the dynamic statement.

Next, you learn how to use the Oracle-supplied DBMS_ASSERT package functions to filter and sanitize input values.

Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT
- Designing code immune to SQL injections
- Testing code for SQL injection flaws

ORACLE®

13 - 22

Copyright © 2008, Oracle. All rights reserved.

Understanding DBMS_ASSERT

DBMS_ASSERT functions:

Function	Description
NOOP	Does not perform any validation and returns the string unchanged. Allows developers to mark some data as trusted, and thereby, disable some SQL injection checks in the source scanning tools. Avoid using this function unless approved by the product security compliance team.
ENQUOTE_LITERAL	Encloses string literal in single quotes
ENQUOTE_NAME	Encloses string literal in double quotes
SIMPLE_SQL_NAME	Verifies that the string is a simple SQL name
QUALIFIED_SQL_NAME	Verifies that the string is a qualified SQL name
SCHEMA_NAME	Verifies that the string is an existing schema name
SQL_OBJECT_NAME	Verifies that the string is a qualified identifier of an existing SQL object

ORACLE®

13 - 23

Copyright © 2008, Oracle. All rights reserved.

Understanding DBMS_ASSERT

To guard against SQL injection in applications that do not use bind arguments with dynamic SQL, you must filter and sanitize concatenated strings. The primary use case for dynamic SQL with string concatenation is when an Oracle identifier (such as a table name) is unknown at code compilation time.

DBMS_ASSERT is an Oracle-supplied PL/SQL package containing seven functions that can be used to filter and sanitize input strings, particularly those that are meant to be used as Oracle identifiers.

While two of these functions can be used to filter and sanitize any input strings, the majority of them are specifically crafted to validate Oracle identifiers. They are the ENQUOTE_LITERAL and the ENQUOTE_NAME functions. The other functions either do nothing (the NOOP function) or return the input string unchanged if the verification algorithm does not raise any exceptions.

When using the DBMS_ASSERT package, always specify the SYS schema rather than relying on a public synonym.

Formatting Oracle Identifiers

- 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';
```

- The object name used as a quoted (normal format) identifier:

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

- This references a different table.
- It is vulnerable to SQL injection.



ORACLE®

13 - 24

Copyright © 2008, Oracle. All rights reserved.

Understanding and Formatting 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.

- An unquoted (or internal format) identifier is not surrounded by punctuation. It must begin with a letter, and may be followed by letters, numbers, or a small set of special characters. This is how identifiers are most often specified, and how object names are stored in data dictionary tables.
- A quoted (or normal format) identifier begins and ends with double quotation marks. The identifier can include almost any character within the double quotes. This format is user-supplied.

SQL injection attacks can use the quoted method to attempt to subvert code that is written to expect only the unquoted, more common, method.

Working with Identifiers in Dynamic SQL

- For your identifiers, determine:
 1. Where will the input come from: user or data dictionary?
 2. What verification is required?
 3. How will the result be used, as an identifier or a literal value?
- These three factors affect:
 - 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

ORACLE®

13 - 25

Copyright © 2008, Oracle. All rights reserved.

Working with Identifiers in Dynamic SQL

When working with identifiers in dynamic SQL statements, you must first determine where the data is coming from, either user data or data dictionary data. For verification, you must determine whether the object needs to exist and the type of identifier. For types of identifiers, you have:

- **Basic:** A basic SQL name is an unquoted identifier. It must start with a letter, and then be followed by letters, numbers, or the special characters. Because of the restricted set of characters that can be used in a basic SQL name, very little input or output formatting is required.
- **Simple:** A simple SQL name is a basic SQL name, but also allows quotes to be used to contain special characters.
- **Qualified SQL name:** A qualified SQL name is one or more simple SQL names that may be followed by a database link.

Lastly, determine how the result will be used, as an identifier or a literal value.

Answering these questions will impact your preprocessing, post-processing, and the use of DBMS_ASSERT.

Choosing a Verification Route

Identifier Type	Verification
Basic	Restrict applications to use only basic SQL names by applying an algorithm similar to the TO_BASIC_SQL_NAME function shown on the next few pages.
Simple	DBMS_ASSERT.SIMPLE_SQL_NAME
Qualified	DBMS_ASSERT.QUALIFIED_SQL_NAME
Database Link	Use DBMS_ASSERT.QUALIFIED_SQL_NAME if you need to filter only the input. However, if you need to confirm that a link name is really a link name, use DBMS.Utility.NAME_TOKENIZE.
Schema	DBMS_ASSERT.SCHEMA_NAME
Object Name	DBMS_ASSERT.SQL_OBJECT_NAME

ORACLE®

13 - 26

Copyright © 2008, Oracle. All rights reserved.

Choosing a Verification Route

After determining the type of identifier you need to verify, follow the table shown in the slide to help you select the appropriate verification routine.

For example, you need to write a procedure that allows a user to change his or her password. To choose the correct DBMS_ASSERT function to use, you need to determine:

1. Where the input will come from, the user or the data dictionary
2. What verification is required—does the object need to exist and if so, what type of identifier is it?
3. How will the identifier be used, as an identifier or as a literal?

Choosing a Verification Route (continued)

TO_WHITESPACE_TRIMMED Code

This is a sample algorithm to trim white space. The code shown on the next page calls it.

```
-- this needs to be declared first:
CREATE OR REPLACE FUNCTION to_whitespace_trimmed
  (i IN VARCHAR2)
RETURN VARCHAR2
AS
  Space CONSTANT CHAR(1) := ' ';
  Newline CONSTANT CHAR(1) := '
';
  Tab CONSTANT CHAR(1) := Chr(9);
  TYPE t IS TABLE OF VARCHAR2(10);
  -- This is just an illustration; there may be others.
  Whitespace_Characters CONSTANT t := t(Space, Newline, Tab);
  v VARCHAR2(32767) := i;
BEGIN
  DECLARE
    Cur_Length PLS_INTEGER := Length(v);
    Prev_Length PLS_INTEGER := Cur_Length + 1;
  BEGIN
    WHILE Cur_Length < Prev_Length LOOP
      Prev_Length := Cur_Length;
      FOR j IN 1..Whitespace_Characters.Count() LOOP
        v := Trim(BOTH Whitespace_Characters(j) FROM v);
      END LOOP;
      Cur_Length := Length(v);
    END LOOP;
  END;
  RETURN v;
END to_whitespace_trimmed;
```

Choosing a Verification Route (continued)

TO_BASIC_SQL_NAME Code

Use this code to convert to a basic SQL name:

```
CREATE OR REPLACE FUNCTION to_basic_sql_name
    (p_user_input VARCHAR2)
RETURN VARCHAR2
AS
    v_trimmed VARCHAR2(30);
    v_temp VARCHAR2(30);
    e_not_a_basic_identifier EXCEPTION;
    PRAGMA Exception_Init(e_not_a_basic_identifier , -20001);
BEGIN
    -- Whitespace is not significant (but note the need to use
    -- the full set of whitespace characters, including tab,
    -- space, newline).
    v_trimmed := To_Whitespace_Trimmed(p_user_input);

    -- If the result is quoted then strip them and
    -- check that the string is upper case. If it
    -- is then it may just be a quoted basic identifier
    IF substr(v_trimmed,1,1) = ''' AND
        substr(v_trimmed,length(v_trimmed),1) = ''' then
        -- remove the quotes
        v_temp := substr(v_trimmed,2,length(v_trimmed)-2);
        IF nls_upper(v_temp) <> v_temp THEN
            -- It isn't upper case so it's not a basic identifier
            RAISE e_not_a_basic_identifier;
        END IF;
    ELSE
        -- Not quoted so safe to convert to upper case
        v_temp := nls_upper(v_trimmed);
    END IF;

    -- Now check the format
    IF regexp_like(v_temp, '\A[[[:alpha:]]([[[:alnum:]#$_]])*$')
    THEN
        -- It is a basic identifier, return the formatted result
        RETURN v_trimmed;
    ELSE
        RAISE e_not_a_basic_identifier;
    END IF;

EXCEPTION WHEN e_not_a_basic_identifier THEN
    DBMS_Output.Put_Line('ORA-20001: Not a basic SQL name.');
    RETURN NULL;
END to_basic_sql_name;
```

DBMS_ASSERT Guidelines

- Do not perform unnecessary uppercase conversions on identifiers.

```
--Bad:  
SAFE_SCHEMA := sys.dbms_assert.SCHEMA_NAME(UPPER(MY_SCHEMA));  
--Good:  
SAFE_SCHEMA := sys.dbms_assert.SCHEMA_NAME(MY_SCHEMA);  
--Best:  
SAFE_SCHEMA := sys.dbms_assert.ENQUOTE_NAME(  
    sys.dbms_assert.SCHEMA_NAME(MY_SCHEMA), FALSE);
```

- When using ENQUOTE_NAME, do not add unnecessary double quotes around identifiers.

```
--Bad:  
tablename := sys.dbms_assert.ENQUOTE_NAME(''  
    ||tab_rec.table_name||''');  
--Good:  
tablename := sys.dbms_assert.ENQUOTE_NAME(tab_rec.table_name, FALSE);  
--Note: If leading quotes are acceptable, then use  
--the following construct:  
safe := sys.dbms_assert.ENQUOTE_NAME(''  
    ||replace('','',''',input_with_leading_quotes)||'''', FALSE);
```

ORACLE®

13 - 29

Copyright © 2008, Oracle. All rights reserved.

DBMS_ASSERT Guidelines

Choosing the correct DBMS_ASSERT verification routines is important, and using them correctly is just as important. The table shown in the slide identifies some usage guidelines.

Limitations

DBMS_ASSERT is not a panacea for all sorts of PL/SQL evils. It is essentially a collection of pattern-matching routines that confirm whether the supplied string matches expected patterns. It can be used to protect against certain kinds of malicious input, but it cannot comprehensively defend against all such inputs.

Here are some instances where DBMS_ASSERT may not help:

- It contains no routines to validate TNS connect strings, for example “((description =...”).
- It is not designed nor is it intended to be a defense against cross-site scripting attacks.
- It does not check for input string lengths, and therefore cannot be used as any kind of defense against a buffer overflow attack.
- It does not guarantee that a SQL name is, in fact, a parseable SQL name.
- It does not protect against parsing as the wrong user or other security risks due to inappropriate privilege management.

DBMS_ASSERT Guidelines

- Avoid using ENQUOTE_NAME with user-supplied input unless it is being used to supply the surrounding double quote for SCHEMA_NAME.
- When using ENQUOTE_LITERAL, remember to escape single quotes in the input.

```
-- Bad
EXECUTE IMMEDIATE 'begin' ||
my_trace_routine('||sys.dbms_assert.ENQUOTE_LITERAL(
my_procedure_name)||');'|||
'end';

--Good
EXECUTE IMMEDIATE 'begin' ||
my_trace_routine('||sys.dbms_assert.ENQUOTE_LITERAL(
replace(my_procedure_name,'''','''')||');'|||
'end';
```

ORACLE®

13 - 30

Copyright © 2008, Oracle. All rights reserved.

DBMS_ASSERT Guidelines (continued)

Another DBMS_ASSERT guideline is:

- If you want to use QUALIFIED_SQL_NAME, but do not want to allow database link specification, you need to use DBMS.Utility.NAME_TOKENIZE to detect the presence of a database link, and then raise an exception.

```
-- Bad
SQL_CMD := 'SELECT COUNT(*) FROM ' ||
sys.dbms_assert.QUALIFIED_SQL_NAME(p_process_order_table)
|| ' WHERE process_order = :1 ';
EXECUTE IMMEDIATE SQL_CMD INTO cnt USING p_process_order;

--Good
sys.dbms_utility.NAME_TOKENIZE(
sys.dbms_assert.QUALIFIED_SQL_NAME(p_process_order_table),
dummy, dummy, dummy, dblink, lastch);
IF dblink IS NOT NULL OR dblink <> '' THEN
    RAISE DBLINK_ERROR; END IF;
-- p_process_order_table has already been verified
SQL_CMD := 'SELECT COUNT(*) FROM '|| p_process_order_table
|| ' WHERE process_order = :1 ';
EXECUTE IMMEDIATE SQL_CMD INTO cnt USING p_process_order;
```

DBMS_ASSERT Guidelines

- Check and reject NULL or empty return results from DBMS_ASSERT (test for NULL, '' , and '""').
- Do not embed verification routines inside the injectable string.

```
-- Bad
CREATE OR REPLACE PROCEDURE change_password7
  (username VARCHAR2, password VARCHAR2)
AS
  v_stmt VARCHAR2(2000);
BEGIN
  v_stmt := 'alter user dbms_assert.ENQUOTE_NAME(
    dbms_assert.SCHEMA_NAME('' '' ||username|| '' ',FALSE)
    identified by
      dbms_assert.SIMPLE_SQL_NAME('' '' ||password|| '' ');
  dbms_output.put_line(v_stmt);
END;
```

ORACLE®

13 - 31

Copyright © 2008, Oracle. All rights reserved.

DBMS_ASSERT Guidelines (continued)

By embedding verification routines inside the injectable string (the V_STMT local PL/SQL variable), the inputs (username and password) are not filtered and sanitized before they are concatenated to the rest of the string. Thus, the code shown on the slide is still vulnerable to injections.

```
EXECUTE change_password7('HR','test') quota unlimited on system--
')
alter user dbms_assert.ENQUOTE_NAME(
  dbms_assert.SCHEMA_NAME('HR',FALSE)
  identified by dbms_assert.SIMPLE_SQL_NAME('test') quota unlimited on
  system--')
PL/SQL procedure successfully completed.
```

By filtering and sanitizing the inputs before they are concatenated into the rest of the string, exceptions would be raised if they are not valid.

```
CREATE OR REPLACE PROCEDURE change_password8
  (username VARCHAR2, password VARCHAR2)
AS
  v_stmt VARCHAR2(2000);
BEGIN
  v_stmt := 'alter user'||sys.dbms_assert.ENQUOTE_NAME
    (sys.dbms_assert.SCHEMA_NAME(username),FALSE)|||
    ' identified by '||dbms_assert.SIMPLE_SQL_NAME(password);
  dbms_output.put_line(v_stmt);
END;
```

DBMS_ASSERT Guidelines

- Protect all injectable parameters and code paths.
- Prefix all calls to DBMS_ASSERT with the owning schema, SYS.
- 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;
```

ORACLE®

13 - 32

Copyright © 2008, Oracle. All rights reserved.

DBMS_ASSERT Guidelines (continued)

You can use PL/SQL conditional compilation from Oracle Database 10.2 and later to manage self-tracing code. For production deployment, the debug messages can be turned off by setting the PLSQL_CCFLAGS parameter for tracing to FALSE. You must ensure that error messages displayed in production deployment do not reveal information that is useful to hackers.

```
ALTER SESSION SET Plsql_CCFlags = 'Tracing:true';

CREATE OR REPLACE PROCEDURE change_password3
  (p_username VARCHAR2, p_password VARCHAR2) AS
BEGIN
  ...
EXCEPTION
  WHEN sys.dbms_assert.INVALID_SCHEMA_NAME THEN
    $if $$Tracing $then dbms_output.put_line('Invalid user.');
    $else dbms_output.put_line('Authentication failed.'); $end
  WHEN sys.dbms_assert.INVALID_SQL_NAME THEN
    $if $$Tracing $then dbms_output.put_line('Invalid pw.');
    $else dbms_output.put_line('Authentication failed.'); $end
  WHEN OTHERS THEN
    dbms_output.put_line('Something else went wrong');
END;
```

Writing Your Own Filters

- Oracle identifiers and keywords require special handling:
 - Consider whether the user input can be constrained to a simply checked identifier. For example, a table name parameter might be constrained to 32 characters. This would allow double quotes surrounding a 30-character string, the maximum length for an Oracle identifier.
 - Perform validation checks using bind arguments. For example, select a table name from the `ALL_TABLES` view to validate that the supplied value truly represents a valid table.
- Valid data may contain special SQL characters such as:
 - Single and double quotes
 - Consecutive dashes (comments notation)
 - Guillems (double angle brackets such as `<>`)
 - Slash and asterisk (comments and hint notation)

ORACLE®

13 - 33

Copyright © 2008, Oracle. All rights reserved.

Writing Your Own Filters

If you have determined that you must write your own input filters, there are a number of issues to consider before you proceed. An incorrectly constructed filter can be just as damaging as, or more damaging than, injectable code.

When crafting your input filters, keep in mind that:

- Oracle identifiers and keywords require special handling.
- Valid data may contain special SQL characters.
- A danger of ‘escaping’ data is that the escape sequence may be truncated by a length limit imposed on a buffer.
- Numeric and logical fields may also be attack vectors and need to be validated.
- Reliable comparison of characters may require negotiating character sets between the system components (for example, the Web browser, the middle tier, and the back-end server).
- Create a list of acceptable characters (known as a white list) and use regular expressions to reject any characters that are not on the list.
- Avoid using a list of unacceptable characters (known as a black list), because it is always possible to overlook an unacceptable character when defining the list. Also, an unacceptable character can be represented in an alternate format to pass validation.

Before deploying your filters, you must obtain approval from the product security compliance team.

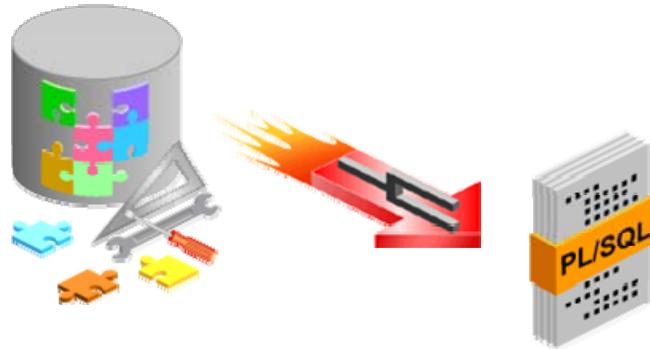
Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT
- Designing code immune to SQL injections
- Testing code for SQL injection flaws



Using Bind Arguments

- Most common vulnerability:
 - Dynamic SQL with string concatenation
- Your code design needs to:
 - Avoid input string concatenation in dynamic SQL
 - Use bind arguments, whether automatically via static SQL or explicitly via dynamic SQL statements



ORACLE®

13 - 35

Copyright © 2008, Oracle. All rights reserved.

Using Bind Arguments

Poor application design can lead to “designed in” vulnerabilities where there are no coding problems per se and everything works as intended.

However, you must design your code such that it is (ideally) entirely free of SQL injection vulnerabilities, or contains measures that mitigate the impact of a successful attack.

As you have seen with the examples presented thus far, the common flaw of all code vulnerable to SQL injection is the construction of dynamic SQL using string concatenation. Complete immunity from SQL injection attack can be achieved only through the elimination of input string concatenation in dynamic SQL.

- Avoid input string concatenation.
- Use bind arguments, whether automatically via static SQL or explicitly via dynamic SQL statements.

Design your code to use bind arguments wherever possible. The only exceptions should be when you need to concatenate identifiers or keywords, because you have no other choice.

Handling Oracle Identifiers Carefully

```
CREATE OR REPLACE PROCEDURE add_trigger
(p_schema VARCHAR2,p_table_name VARCHAR2) AS
  v_stmt VARCHAR2(4000);
BEGIN
  FOR o IN (SELECT object_id FROM all_objects
            WHERE owner=p_schema AND
            object_type='TABLE' AND
            object_name=p_table_name) LOOP
    v_stmt := 'CREATE OR REPLACE TRIGGER '
    || sys.dbms_assert.enquote_name(
      sys.dbms_assert.schema_name(p_schema),FALSE)
    || '.' || sys.dbms_assert.enquote_name('XX$'||o.object_id)
    || ' AFTER UPDATE ON '||sys.dbms_assert.enquote_name(
      sys.dbms_assert.schema_name(p_schema),FALSE)
    || '.' || sys.dbms_assert.enquote_name(
      sys.dbms_assert.simple_sql_name(p_table_name),FALSE)
    || ' FOR EACH ROW Begin NULL; End;';
    DBMS_OutPut.Put_Line('SQL stmt: ' || v_stmt);
    EXECUTE IMMEDIATE v_stmt;
  END LOOP;
EXCEPTION WHEN OTHERS THEN
  RAISE;
END;
```

Note that by looking up the object ID from the data dictionary using the user-supplied input in this static SQL statement, you are essentially validating input.

Use the DBMS_ASSERT functions to ensure that correct identifier formatting is preserved.

ORACLE®

13 - 36

Copyright © 2008, Oracle. All rights reserved.

Handling Oracle Identifiers Carefully

As you have seen, Oracle identifier formats and usage contexts can pose some interesting challenges in input validation.

In addition to following the guidelines for filtering input with DBMS_ASSERT routines, here are a few other issues that you need to pay particular attention to:

- Explicitly validate and enforce identifier length limits. Where the length limits are being enforced, these limits should be clearly marked so that they can be changed easily if the length limit does change in future Database releases.
- Take care to filter out control characters in user-supplied identifier names. Control characters such as chr(10) (linefeed character) can generate spurious audit records.
- If an Oracle identifier needs to be generated based on a user-supplied identifier, avoid concatenating the supplied identifier or using sequences to construct the generated Oracle identifier name. Instead, consider looking up the object ID for the user-supplied identifier in the data dictionary and using the object ID in the generated Oracle identifier name.

For example, you need to write a procedure that generates a trigger based on a table name. The procedure accepts two parameters: the schema name and the table name.

The code shown on the slide can be used to generate Oracle identifiers based on user-supplied identifiers. This example is continued on the next page.

Handling Oracle Identifiers Carefully

```
CONN / AS sysdba

CREATE TABLE oe.test99 (col1 number, col2 varchar2(200));

SELECT object_id, object_name FROM all_objects
WHERE object_name = 'TEST99'
AND object_type = 'TABLE' AND owner = 'OE';

OBJECT_ID OBJECT_NAME
-----
76126 TEST99
```

A test table is created and its object ID is retrieved from the data dictionary.

```
EXEC sys.add_trigger('OE','TEST99')
SQL stmt: CREATE OR REPLACE TRIGGER "OE"."XX$76126" AFTER UPDATE ON
"OE"."TEST99" FOR EACH ROW Begin NULL; End;

SELECT object_id, object_name FROM all_objects
WHERE object_type = 'TRIGGER' AND object_name like 'XX%'
AND owner = 'OE';

OBJECT_ID OBJECT_NAME
-----
76128 XX$76126
```

The ADD_TRIGGER procedure generates Oracle identifiers based on user-supplied identifiers.

ORACLE®

13 - 37

Copyright © 2008, Oracle. All rights reserved.

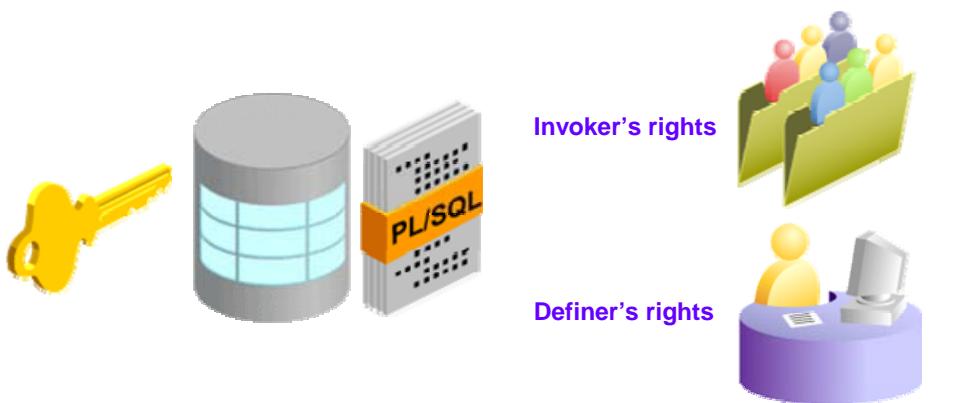
Handling Oracle Identifiers Carefully (continued)

In addition to being SQL injection-proof, the example meets these requirements:

- The generated name is predictable and unique.
- The generated name is immune to object renaming.

Avoiding Privilege Escalation

- Give out privileges appropriately.
- Run code with invoker's rights when possible.
- Ensure that the database privilege model is upheld when using definer's rights.



13 - 38

Copyright © 2008, Oracle. All rights reserved.

ORACLE®

Avoiding Privilege Escalation

Unless carefully designed, routines may effectively grant users more privileges than intended. Wherever possible, run code with invoker's rights to minimize the scope for privilege escalation attacks and to mitigate the impact of a successful SQL injection attack.

Where this is not possible, the routines that are run with definer's rights should be carefully reviewed to ensure that the database privilege model is upheld.

If none of the methods of execution (definer's rights, invoker's rights) appears suitable, consider implementing specific bypass checks for the duration of the call.

Beware of Filter Parameters

- Filter parameter:
 - P_WHERE_CLAUSE is a filter.
 - It is difficult to protect against SQL injection.

```
stmt := 'SELECT session_id FROM sessions
WHERE' || p_where_clause;
```

- Prevention methods:
 - Do not specify APIs that allow arbitrary query parameters to be exposed.
 - Any existing APIs with this type of functionality must be deprecated and replaced with safe alternatives.



Beware of Filter Parameters

Occasionally, routines may need to take a parameter that is used to form an expression in a query or other PL/SQL statements. Typically such parameters are referred to as “filters.”

How do you test that the provided expression is free from SQL injection? You cannot. In these cases, the only sure solution is prevention.

- Do not specify APIs that allow arbitrary query parameters to be exposed.
- Any existing APIs with this type of functionality must be deprecated and replaced with safe alternatives.

If you must make use of filter parameters, contact your security compliance team for approval.

Any such routines must be registered as possible sinks of dynamic SQL and recorded in static code analysis rulesets. This helps ensure that any users of these APIs are well behaved. Contact your security compliance team to register these routines.

Trapping and Handling Exceptions

- Design your code to trap and handle exceptions appropriately.
- Before deploying your application:
 - Remove all code tracing
 - Remove all debug messages



ORACLE®

13 - 40

Copyright © 2008, Oracle. All rights reserved.

Trapping and Handling Exceptions

Normal SQL injection attacks depend, in large measure, on an attacker's reverse-engineering portions of the original SQL query by using information gained from error messages. Therefore, keep application error messages succinct and do not divulge metadata information (such as column names and table names).

Note: From Oracle Database 10.2 and later, you can use PL/SQL conditional compilation for managing self-tracing code, asserts, and so on.

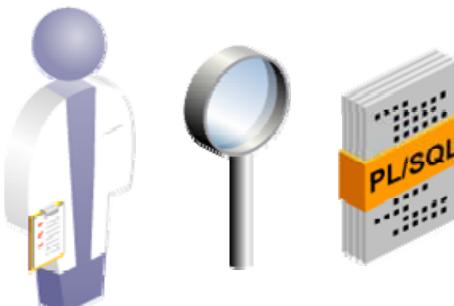
Lesson Agenda

- Understanding SQL injection
- Reducing the attack surface
- Avoiding dynamic SQL
- Using bind arguments
- Filtering input with DBMS_ASSERT
- Designing code immune to SQL injections
- Testing code for SQL injection flaws



Coding Review and Testing Strategy

- Test:
 - Dynamic testing
 - Static testing
- Review:
 - Peer and self reviews
 - Analysis tools



ORACLE®

13 - 42

Copyright © 2008, Oracle. All rights reserved.

Coding Review and Testing Strategy

You can use a number of strategies to test for SQL injection vulnerability. Using a combination of these strategies should be regarded as a sensible minimum to get some degree of confidence in freedom from vulnerabilities.

Effective testing of complex products is essentially a process of investigation, and not merely a matter of creating and following routine procedure. Code reviews or walk-throughs are referred to as “static testing,” whereas actually running the program with a given set of test cases in a given development stage is often referred to as “dynamic testing.”

Testing for SQL injection flaws requires both static and dynamic testing. For static testing, you can begin with a peer (or self) code review and/or make use of a static code analysis tool. After finding and fixing the semantical SQL injection bugs, you must perform dynamic testing by using tools that generate random input (fuzzing), and also run through test cases that you define specifically for SQL injection detection within your code.

Reviewing Code

Language	Look for...
PL/SQL	<code>EXECUTE IMMEDIATE OPEN cursor_var FOR ... DBMS_SQL DBMS_SYS_SQL</code>
C	<p>String substitutions such as:</p> <pre>static const oratext createsq[] = "CREATE SEQUENCE \"%.*s\".\"%.*s\" start with %.*s increment by %.*s";</pre> <p>Followed by usage such as:</p> <pre>DISCARD lstrprintf(sql_txt, createsq, ownerl, owner, seqnaml, seqnam, sizeof(start), start, sinc_byl, sincrement_by);</pre>
Java	<p>String concatenations such as:</p> <pre>sqltext = "DROP VIEW " + this.username + "." + this.viewName;</pre>

ORACLE®

13 - 43

Copyright © 2008, Oracle. All rights reserved.

Reviewing Code

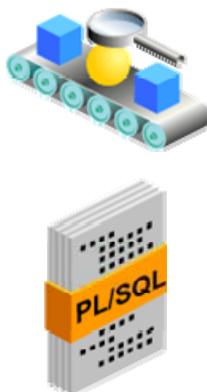
When reviewing code, first identify all dynamic SQL statements. Depending on your programming language, some key indicators that dynamic SQL is in use are shown in the slide.

Next, check to make sure that bind arguments are used in all possible and practical instances.

Where bind arguments cannot be used, make sure that the correct routines are used to filter and sanitize inputs.

Running Static Code Analysis

- Generally performed by an automated tool
- Can be performed on some versions of the source code
- Can be performed on some forms of the object code
- Should be used as one of the initial steps of testing code



ORACLE®

13 - 44

Copyright © 2008, Oracle. All rights reserved.

Running Static Code Analysis

Static code analysis is the analysis of computer software that is performed without executing programs built from that software. In most cases, the analysis is performed on some version of the source code and in other cases, some form of the object code. The term is usually applied to the analysis performed by an automated tool.

Because SQL injections arise from dynamically generated SQL, static code analysis tools may have a hard job identifying all categories of SQL injection with some certainty, because this may require knowledge of object names and contents not available at analysis time. For example, a PL/SQL function might construct a string containing parts of a SQL statement that might then be combined with a malicious table name, thereby leading to an injection.

Static code analysis tools should not be used for any kind of security sign-off. Instead, it should be one of the initial steps in the code testing process.

Testing with Fuzzing Tools

- Is a software testing technique that provides random data (“fuzz”) to the inputs of a program.
- Can enhance software security and software safety, because it often finds odd oversights and defects that human testers would fail to find.
- Must not be used as a substitute for exhaustive testing or formal methods.
- While tools can be used to automate fuzz testing, it may be necessary to customize the test data and application context to get the best results from such tools.

ORACLE®

13 - 45

Copyright © 2008, Oracle. All rights reserved.

Testing with Fuzzing Tools

Fuzz testing or fuzzing is a software testing technique that provides random data (“fuzz”) to the inputs of a program. If the program fails (for example, by crashing or by failing built-in code assertions), the defects can be noted. The great advantage of fuzz testing is that the test design is extremely simple and free of preconceptions about system behavior.

Fuzz testing is thought to enhance software security and software safety, because it often finds odd oversights and defects that human testers would fail to find, and even careful human test designers would fail to create tests for.

However, fuzz testing is not a substitute for exhaustive testing or formal methods; it can only provide a random sample of the system’s behavior, and in many cases passing a fuzz test may only demonstrate that a piece of software handles exceptions without crashing, rather than behaving correctly. Thus, fuzz testing can be regarded only as a bug-finding tool rather than an assurance of quality.

While tools can be used to automate fuzz testing, it may be necessary to customize the test data and application context to get the best results from such tools.

Generating Test Cases

- Test each input parameter individually.
- When testing each parameter, leave all the other parameters unchanged with valid data as their arguments.
- Omitting parameters or supplying bad arguments to other parameters while you are testing another for SQL injection can break the application in ways that prevent you from determining whether SQL injection is possible.
- Always use the full parameter line, supplying every parameter, except the one that you are testing, with a legitimate value.
- Certain object and schema names help you uncover SQL injection vulnerabilities.

ORACLE®

13 - 46

Copyright © 2008, Oracle. All rights reserved.

Generating Test Cases

When designing SQL injection test cases, keep in mind that each input parameter needs to be tested individually. When testing each parameter, leave all the other parameters unchanged with valid data as their arguments.

It can be tempting to delete everything that you are not working with to make things appear simpler, particularly with applications that have parameter lines going into many thousands of characters. Omitting parameters or supplying bad arguments to other parameters while you are testing another for SQL injection can break the application in ways that prevent you from determining whether SQL injection is possible.

So, when testing for SQL injection, always use the full parameter line, supplying every parameter, except the one that you are testing, with a legitimate value.

Certain object and schema names help you uncover SQL injection vulnerabilities. On the next page is a list of names as input values in your tests.

Generating Test Cases (continued)

Name	Test
AAAAAAAAAAABBBB BBBBBBCCCCCCCC	Maximum size identifiers should be tested as both schema and object names. Checks that temporary object names can be created.
AAAAAAAAAAAAAAA AAAAAAAAAAAAAAA AA & AAAAAAAAAAAAAAA AAAAAAAAAAAAAAA A	Maximum and “nearly” maximum object names should be handled as separate objects. Checks that any name truncation operation does not result in non-unique names.
schema1.a schema2.a	Ensures that objects with the same name in different schemas are properly differentiated.
AAA "aaa"	Ensures that objects with different case names are clearly differentiated.
object@dblink	Ensures that link names (both valid and invalid) do not defeat security tests. Ensure that links are rejected unless specifically allowable.
Rename "ABC" to "XYZ"	Renames objects during tests to ensure that applications continue to function correctly.
"quoted"	Ensures that quoted schema and object names can be handled.
"AAA" " AAA" "AAA "	Ensures that quoted objects do not lead to second order injections.
"\$IF"	Refers to a variable and not a preprocessor directive.
"BEGIN"	Refers to an object and not a reserved word.
"A%TYPE"	Refers to an object and not an object attribute.
"a/*X*/ <<A>>, %s-- %00\0AAA ", "a' '''' /*/ .@\?:x&y}{}(><)[]!=;A"	Ensures that special characters in quoted variables do not have other side effects.
"xxx" chr(8) "yyy", "xxx" chr(9) "yyy", "xxx" chr(10) "yyy", "xxx" chr(13) "yyy", "xxx" chr(4) "yyy", "xxx" chr(26) "yyy", "xxx" chr(0) "yyy"	Ensures that object names with embedded control characters are handled correctly. Checks that all output correctly distinguishes these objects, especially any output written to files. Note that some object names may be (correctly) rejected as invalid syntax.
NULL	Null and empty strings may behave differently on different Oracle Database ports or versions.

Summary

In this lesson, you should have learned how to:

- Detect SQL injection vulnerabilities
- Reduce attack surfaces
- Use DBMS_ASSERT
- Design immune code
- Test code for SQL injection flaws



Summary

In this lesson, you learned how to use techniques and tools to strengthen your code and applications against SQL injection attacks.

Practice 13: Overview

This practice covers the following topics:

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



Practice 13: Overview

Using the OE, you examine PL/SQL code, test it for SQL injection, and rewrite it to protect against SQL injection vulnerabilities.

Use the OE schema for this practice.

For detailed instructions about performing this practice, see Appendix A, “Practice Solutions.”

Practice 13

Understanding SQL Injection

Answer the following questions to test your knowledge about how to reduce attack surface.

- Only code that is used in Web applications is vulnerable to SQL injection attack.

True/False

- Code that is most vulnerable to SQL injection attack contains (check all that apply):

- Input parameters
- Dynamic SQL with bind arguments
- Dynamic SQL with concatenated input values
- Calls to exterior functions

- By default, a stored procedure or SQL method executes with the privileges of the owner (definer's rights).

True/False

- By using AUTHID CURRENT_USER in your code, you are (check all that apply):

- Specifying that the code executes with invoker's rights
- Specifying that the code executes with the highest privilege level
- Eliminating any possible SQL injection vulnerability
- Not eliminating all possible SQL injection vulnerabilities

- Match each attack surface reduction technique with an example of the technique.

Technique	Example
Executes code with minimal privileges	Specify appropriate parameter types
Lock down the database	Revoke privileges from PUBLIC
Reduce arbitrary input	Use invoker's rights

Practice 13 (continued)

Rewriting Code to Protect Against SQL Injection

6. Examine this code. Run the lab_13_06.sql script to create the procedure.

```
CREATE OR REPLACE PROCEDURE get_income_level (p_email
    VARCHAR2 DEFAULT NULL)
IS
    TYPE      cv_custtyp IS REF CURSOR;
    cv        cv_custtyp;
    v_income  customers.income_level%TYPE;
    v_stmt    VARCHAR2(400);
BEGIN
    v_stmt := 'SELECT income_level FROM customers WHERE
    cust_email = '''
    || p_email || '''';

    DBMS_OUTPUT.PUT_LINE('SQL statement: ' || v_stmt);
    OPEN cv FOR v_stmt;
    LOOP
        FETCH cv INTO v_income;
        EXIT WHEN cv%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Income level is: ' || v_income);
    END LOOP;
    CLOSE cv;

EXCEPTION WHEN OTHERS THEN
    dbms_output.PUT_LINE(sqlerrm);
    dbms_output.PUT_LINE('SQL statement: ' || v_stmt);
END get_income_level;
/
```

- a. Execute the following statements and note the results.

```
exec get_income_level('Sean.Olin@SCAUP.COM')

exec get_income_level('x'' union select username from
all_users where ''x'''='x')
```

- b. Has SQL injection occurred?

Practice 13 (continued)

Rewriting Code to Protect Against SQL Injection (continued)

7. Rewrite the code to protect against SQL injection. You can run the lab_13_07.sql script to re-create the procedure.

- a. Execute the following statements and note the results:

```
exec get_income_level('Sean.Olin@SCAUP.COM')
```

```
exec get_income_level('x' union select username from all_users where ''x''='x')
```

- b. Has SQL injection occurred?

8. Play the les13_staticsql viewlet located in the D:\labs\viewlets folder. This is an example of using static SQL to handle a varying number of IN-list values in a query condition.

To play a viewlet, double-click the les13_staticsql.htm file that is located in the D:\labs\viewlets folder. When prompted, click Start.

Appendix A

Practices and Solutions

Table of Contents

Practices for Lesson 1	3
Practice 1-1: Introduction	3
Practices for Lesson 2	16
Practice 2-1: PL/SQL Review.....	16
Practices for Lesson 3	21
Practice 3-1: Designing PL/SQL Code	21
Practices for Lesson 4	24
Practice 4-1: Working with Collections.....	24
Practices for Lesson 5	32
Practice 5-1: Using Advanced Interface Methods	32
Practices for Lesson 6.....	36
Practice 6-1: Implementing Fine-Grained Access Control for VPD	36
Practices for Lesson 7	41
Practice 7-1: Manipulating Large Objects	41
Practices for Lesson 8	47
Practice 8-1: Using SecureFile Format LOBs	47
Practices for Lesson 9	56
Practice 9-1: Performance and Tuning	56
Practices for Lesson 10	66
Practice 10-1: Improving Performance with Caching	66
Practices for Lesson 11	71
Practice 11-1: Analyzing Your Code	71
Practices for Lesson 12	79
Practice 12-1: Profiling and Tracing.....	79
Practices for Lesson 13	84
Practice 13-1: Safeguarding Your Code Against SQL Injection Attacks.....	84

Practices for Lesson 1

Practice 1-1: Introduction

In this practice, you review the available Oracle SQL Developer resources. You also learn about the user account that you use in this course. You then start Oracle SQL Developer, create a new database connection, and browse your SH and HR tables. You also set some Oracle SQL Developer preferences, execute SQL statements, and access and bookmark the Oracle Database 11g documentation and other useful Web sites that you can use in this course.

Identifying the Available Oracle SQL Developer Resources

- 1) Familiarize yourself with **Oracle SQL Developer**, as needed, by referring to Appendix C, “Using Oracle SQL Developer.”
- 2) Access the online Oracle SQL Developer Home page, which is available online at: http://www.oracle.com/technology/products/database/sql_developer/index.html

The Oracle SQL Developer Home page appears as follows:

Practice 1-1: Introduction (continued)

The screenshot shows a Microsoft Internet Explorer window displaying the Oracle Technology Network website. The address bar shows the URL: http://www.oracle.com/technology/products/database/sql_developer/index.html. The page content includes a sidebar with links for PRODUCTS, TECHNOLOGIES, and COMMUNITY. The main content area features a section about Oracle SQL Developer, which is described as a free graphical tool for database development. It highlights features like browsing database objects, running SQL statements, and editing PL/SQL. A 'FREE DOWNLOAD' button is present, along with a testimonial from Karl Doering and a list of related technologies.

- 3) Bookmark the page for easier future access.

On the Windows Internet Explorer Address toolbar, click and drag the Explorer icon onto the Links toolbar. The link is added to your Links toolbar as follows:

Practice 1-1: Introduction (continued)

The screenshot shows a Microsoft Internet Explorer window displaying the Oracle Technology Network website. The address bar shows the URL http://www.oracle.com/technology/products/database/sql_developer/index.html. A red box highlights the 'Oracle SQL Developer' link in the 'Links' menu. The main content area displays information about Oracle SQL Developer, including a 'FREE DOWNLOAD' button.

- 4) Access the Oracle SQL Developer tutorial, which is available online at:
<http://st-curriculum.oracle.com/tutorial/SQLDeveloper/index.htm>

Access the Oracle SQL Developer tutorial by using the preceding URL. The following page appears:

The screenshot shows a Microsoft Internet Explorer window displaying the Oracle SQL Developer Tutorial. The address bar shows the URL <http://st-curriculum.oracle.com/tutorial/SQLDeveloper/index.htm>. A red box highlights the left sidebar containing the table of contents for the tutorial. The main content area is titled 'Welcome to the Oracle SQL Developer T' and provides an overview of the tutorial's objectives and last update date.

Practice 1-1: Introduction (continued)

- 5) Preview and experiment with the available links and demonstrations in the tutorial as needed, especially the Creating a Database Connection and Accessing Data links.

To review the section on creating a database connection, click the plus sign (+) next to the “What to Do First” link to display the “Creating a Database Connection” link. To review the Creating a Database Connection topic, click the topic’s link. To review the section on accessing data, click the plus sign (+) next to the “Accessing Data” link to display the list of available topics. To review any of the topics, click the topic’s link.

Creating and Using a New Oracle SQL Developer Database Connection

- 6) Start Oracle SQL Developer.

Click the Oracle SQL Developer icon on your desktop.



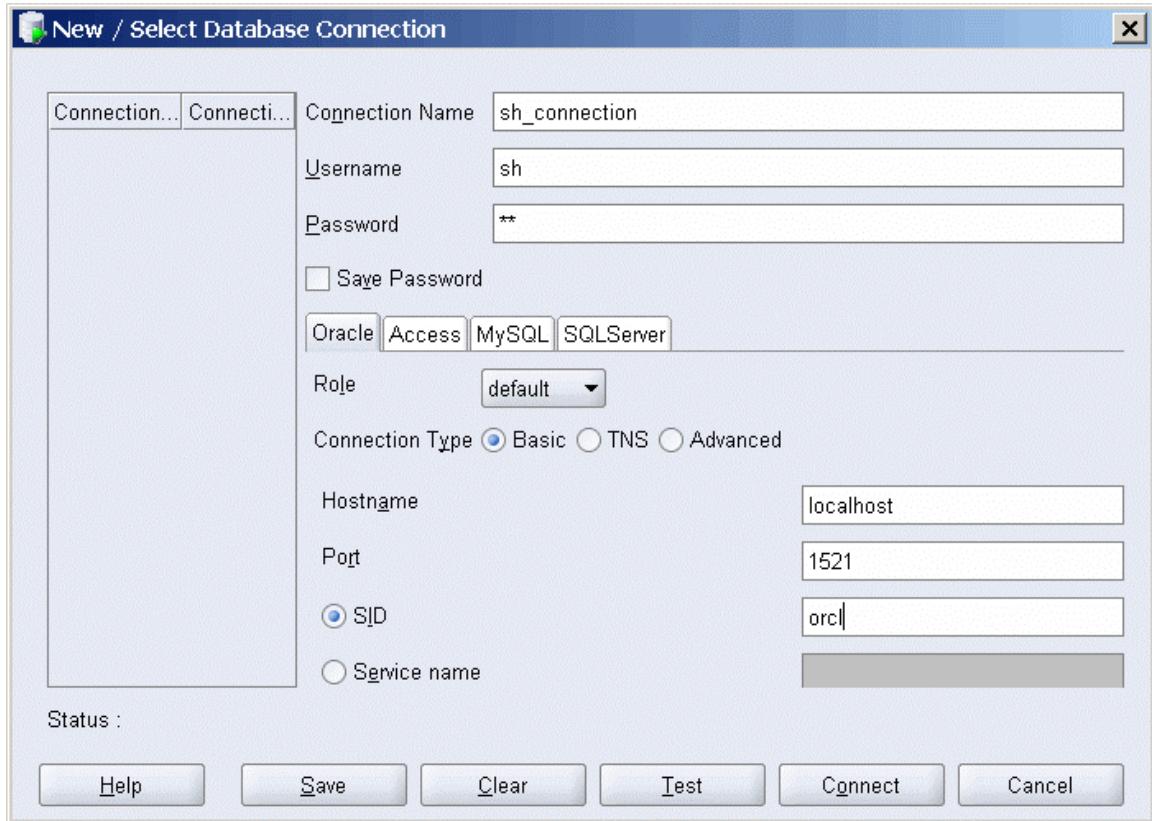
- 7) Create a database connection to SH schema using the following information:

- a) Connection Name: sh_connection
- b) Username: sh
- c) Password: sh
- d) Hostname: localhost
- e) Port: 1521
- f) SID: orcl

Right-click the Connections icon on the Connections tabbed page, and then select the New Connection option from the shortcut menu. The New/Select Database Connection window appears. Use the preceding information to create the new database connection.

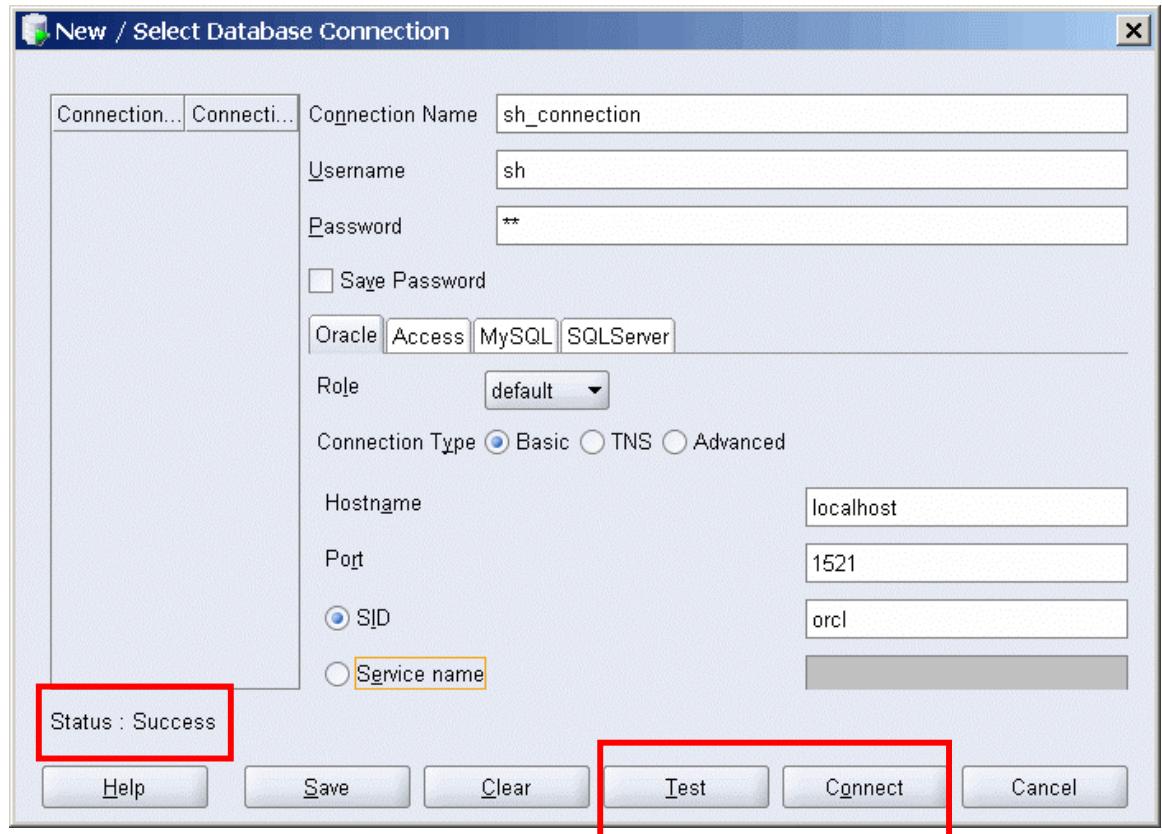
Note: To display the properties of the newly created connection, right-click the connection name, and then select Properties from the shortcut menu. Enter the username, password, host name, and service name with the appropriate information, as provided above. The following is a sample of the newly created database connection for the SH schema using a local connection:

Practice 1-1: Introduction (continued)



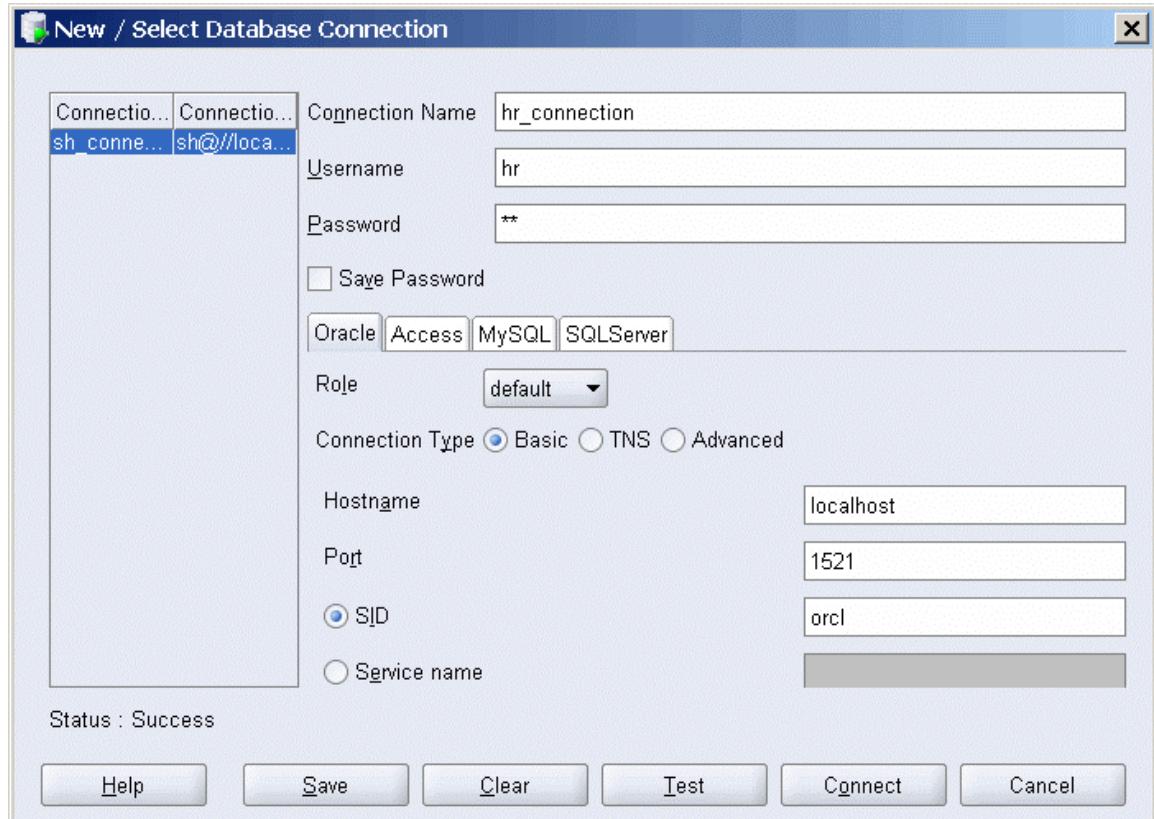
- 8) Test the new connection. If the Status is Success, connect to the database using this new connection.
 - a) Double-click the sh_connection icon on the Connections tabbed page.
 - b) Click the Test button in the New/Select Database Connection window. If the status is Success, click the Connect button.

Practice 1-1: Introduction (continued)



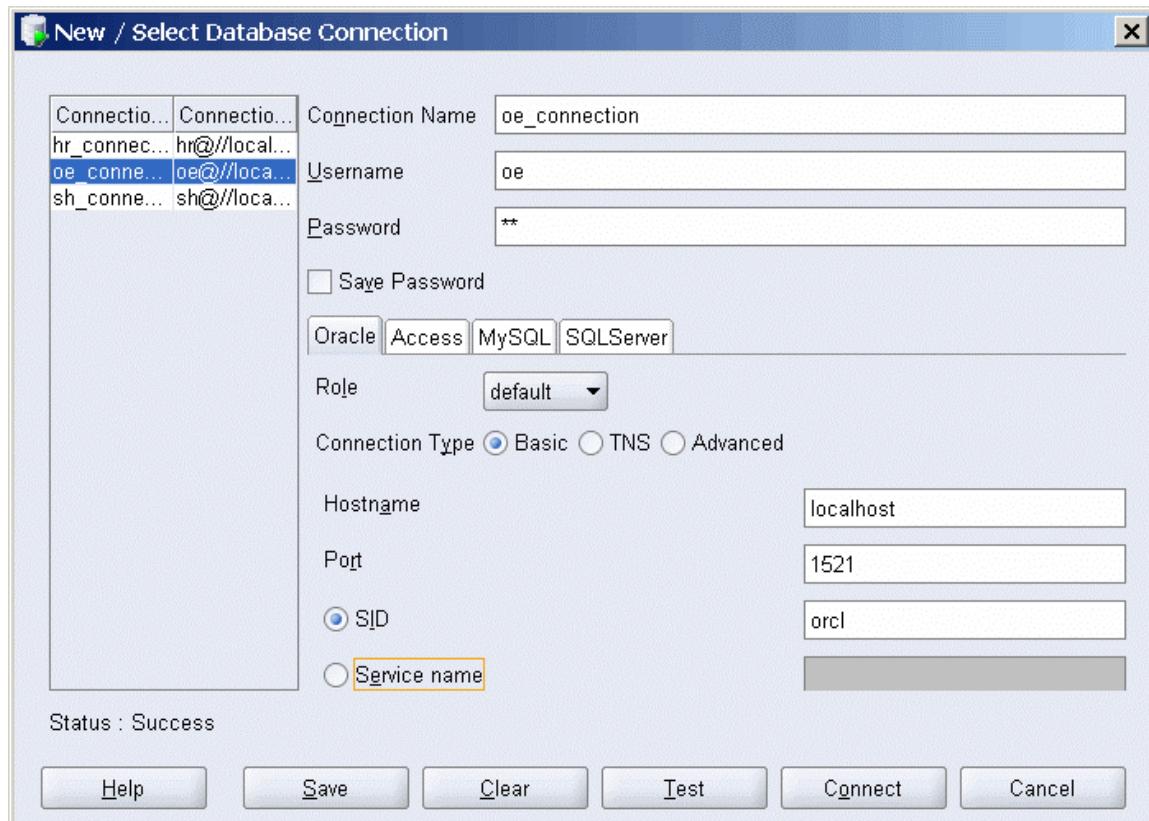
- 9) Create a new database connection named hr_connection.
 - a) Right-click the sh_connection connection in the Object Navigation tree, and select the Properties menu option.
 - b) Enter hr_connection as the connection name and hr as the username and password, and then click Save to create the new connection.
 - c) Repeat step 8 to test the new hr_connection connection.

Practice 1-1: Introduction (continued)



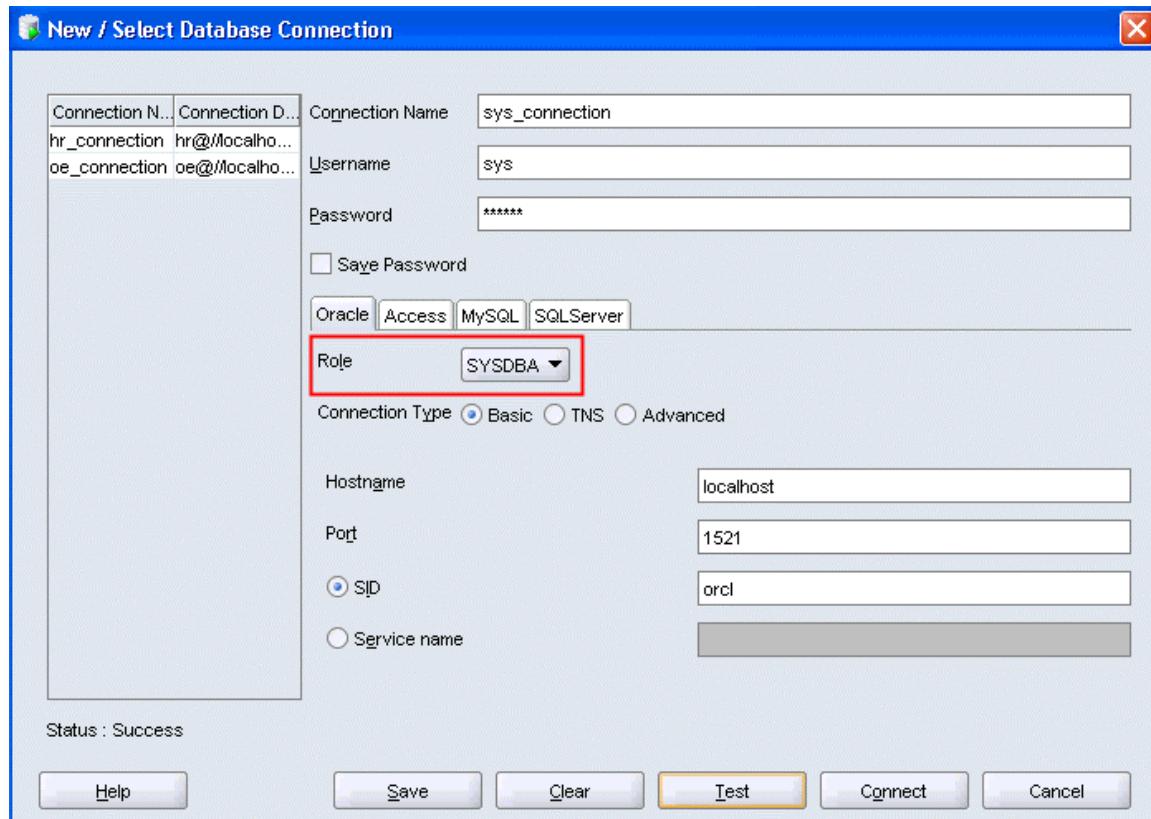
- 10) Repeat step 9 to create and test a new database connection named oe_connection. Enter oe as the database connection username and password.

Practice 1-1: Introduction (continued)



- 11) Repeat step 9 to create and test a new database connection named sys_connection. Enter sys in the Username field, oracle in the Password field, and SYSDBA as the role.

Practice 1-1: Introduction (continued)



From the Role drop-down menu, select SYSDBA as the role.

Browsing the HR, SH, and OE Schema Tables

12) Browse the structure of the EMPLOYEES table in the HR schema.

- a) Expand the hr_connection connection by clicking the plus sign.
- b) Expand the Tables icon by clicking the plus sign.
- c) Display the structure of the EMPLOYEES table.

Double-click the EMPLOYEES table. The Columns tab displays the columns in the EMPLOYEES table as follows:

Practice 1-1: Introduction (continued)

The screenshot shows the Oracle SQL Developer interface. The left pane displays a tree view of database objects under the 'hr_connection' connection, including COUNTRIES, DEPARTMENTS, and EMPLOYEES. The EMPLOYEES node is selected. The right pane shows the 'EMPLOYEES' table structure with 11 columns: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, and DEPARTMENT_ID. Each column has its data type, whether it's nullable, its default value, its column ID, if it's a primary key, and a detailed comment describing its purpose.

Column Name	Data Type	Nullable	Data Default	COLUMN ID	Primary Key	COMMENTS
EMPLOYEE_ID	NUMBER(6,0)	No	(null)	1	1	Primary key of employees table
FIRST_NAME	VARCHAR2(20 BYTE)	Yes	(null)	2	(null)	First name of the employee. A
LAST_NAME	VARCHAR2(25 BYTE)	No	(null)	3	(null)	Last name of the employee. A
EMAIL	VARCHAR2(25 BYTE)	No	(null)	4	(null)	Email id of the employee
PHONE_NUMBER	VARCHAR2(20 BYTE)	Yes	(null)	5	(null)	Phone number of the employee
HIRE_DATE	DATE	No	(null)	6	(null)	Date when the employee starte
JOB_ID	VARCHAR2(10 BYTE)	No	(null)	7	(null)	Current job of the employee; for
SALARY	NUMBER(8,2)	Yes	(null)	8	(null)	Monthly salary of the employee
COMMISSION_PCT	NUMBER(2,2)	Yes	(null)	9	(null)	Commission percentage of the
MANAGER_ID	NUMBER(6,0)	Yes	(null)	10	(null)	Manager id of the employee; ha
DEPARTMENT_ID	NUMBER(4,0)	Yes	(null)	11	(null)	Department id where employee

13) Browse the EMPLOYEES table and display its data.

To display the employee data, click the Data tab. The EMPLOYEES table data is displayed as follows:

Practice 1-1: Introduction (continued)

The screenshot shows the Oracle SQL Developer interface. The title bar reads "Oracle SQL Developer : TABLE HR.EMPLOYEES@hr_connection". The menu bar includes File, Edit, View, Navigate, Run, Debug, Source, Migration, Tools, and Help. The toolbar has icons for Connections, Reports, and various database operations. The left sidebar shows the connection structure under "hr_connection" for the "Tables" category, specifically the "EMPLOYEES" table with its columns: EMPLOYEE_ID, FIRST_NAME, LAST_NAME, EMAIL, PHONE_NUMBER, HIRE_DATE, JOB_ID, SALARY, COMMISSION_PCT, MANAGER_ID, and DEPARTMENT_ID. The main pane displays the data for the "EMPLOYEES" table. A red box highlights the "Data" tab in the top navigation bar. Another red box highlights the status bar at the bottom which says "Fetched Rows: 55".

- 14) Use the SQL Worksheet to select the last names and salaries of all employees whose annual salary is greater than \$10,000. Use both the Execute Statement (F9) and the Run Script (F5) icons to execute the SELECT statement. Review the results of both methods of executing the SELECT statements on the appropriate tabs.

Note: Take a few minutes to familiarize yourself with the data, or consult Appendix B, “Table Descriptions,” which provides the description and data for all tables in the HR, SH, and OE schemas that you will use in this course.

Display the SQL Worksheet by using any of the following two methods:

Select Tools > SQL Worksheet, or click the Open SQL Worksheet icon. The Select Connection window appears. Enter the following statement in the SQL Worksheet:

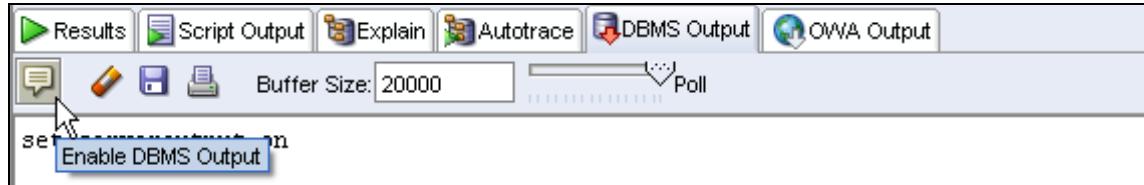
```
SELECT *
FROM employees
WHERE SALARY > 10000;
```

- 15) Create and execute a simple anonymous block that outputs “Hello World.”

Practice 1-1: Introduction (continued)

- a) Enable SET SERVEROUTPUT ON to display the output of the DBMS_OUTPUT package statements.

Click the DBMS_OUTPUT tab, and then click the Enable DBMS Output icon as follows:

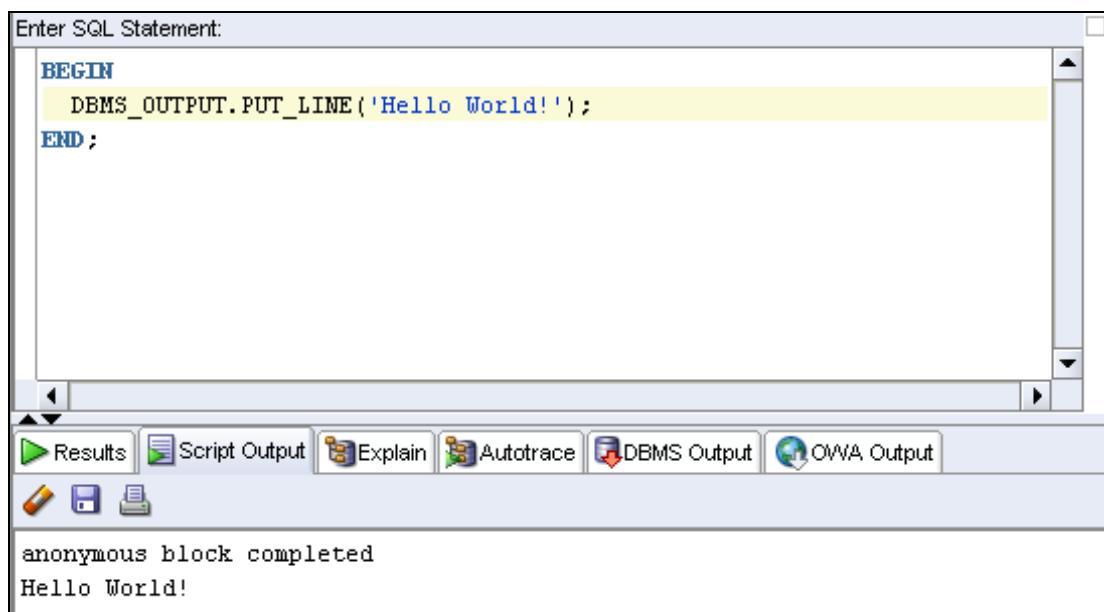


- b) Use the SQL Worksheet area to enter the code for your anonymous block.



- c) Click the Run Script icon (F5) to run the anonymous block.

The Script Output tab displays the output of the anonymous block as follows:



- 16) Browse the structure of the SALES table in the SH schema and display its data.

- a) Double-click the sh_connection database connection.
- b) Expand the Tables icon by clicking the plus sign.

Practice 1-1: Introduction (continued)

- c) Display the structure of the SALES table.
- d) Browse the SALES table and display its data.

Double-click the SALES table. The Columns tab displays the columns in the SALES table. To display the sales data, click the Data tab. The SALES table data is displayed.

- 17) Browse the structure of the ORDERS table in the OE schema and display its data.

- a) Double-click the oe_connection database connection.
- b) Expand the Tables icon by clicking the plus sign.
- c) Display the structure of the ORDERS table.
- d) Browse the ORDERS table and display its data.

Double-click the ORDERS table. The Columns tab displays the columns in the ORDERS table. To display the order data, click the Data tab. The ORDERS table data is displayed.

Accessing the Oracle Database 11g, Release 1 Online Documentation Library

- 18) Access the Oracle Database 11g Release documentation Web page at:

<http://www.oracle.com/pls/db111/homepage>

- 19) Bookmark the page for easier future access.

- 20) Display the complete list of books available for Oracle Database 11g, Release 1.

- 21) Make a note of the following documentation references that you will use in this course as needed:

- a) *Oracle Database 2 Day + Data Warehousing Guide 11g Release 1 (11.1)*
- b) *Oracle Database Data Warehousing Guide 11g Release 1 (11.1)*
- c) *Oracle Database SQL Developer User's Guide Release 1.2*
- d) *Oracle Database Reference 11g Release 1 (11.1)*
- e) *Oracle Database New Features Guide 11g Release 1 (11.1)*
- f) *Oracle Database SQL Language Reference 11g Release 1 (11.1)*
- g) *SQL*Plus User's Guide and Reference Release 11.1*
- h) *Oracle Database SQLJ Developer's Guide and Reference 11g Release 1 (11.1)*
- i) *Oracle Database Concepts 11g Release 1 (11.1)*
- j) *Oracle Database Sample Schemas 11g Release 1 (11.1)*

Practices for Lesson 2

Practice 2-1: PL/SQL Review

In this practice, you test and review your PL/SQL knowledge.

PL/SQL Basics

- 1) What are the four key areas of the basic PL/SQL block? What happens in each area?

Header section: Names the program unit and identifies it as a procedure, function, or package; also identifies any parameters that the code may use.

Declarative section: Area used to define variables, constants, cursors, and exceptions; starts with the keyword IS or AS.

Executable section: Main processing area of the PL/SQL program; starts with the keyword BEGIN.

Exception handler section: Optional error handling section; starts with the keyword EXCEPTION.

- 2) What is a variable and where is it declared?

Variables are used to store data during PL/SQL block execution. You can declare variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its data type, and name the storage location so that you can reference it. Declarations can also assign an initial value and impose the NOT NULL constraint on the variable.

Syntax: `variable_name datatype[(size)] [:= initial_value];`

- 3) What is a constant and where is it declared?

Constants are variables that never change. Constants are declared and assigned a value in the declarative section, before the executable section.

Syntax: `constant_name CONSTANT datatype[(size)] := initial_value;`

- 4) What are the different modes for parameters and what does each mode do?

There are three parameter modes: IN, OUT, and IN OUT. IN is the default and it means a value is passed into the subprogram. The OUT mode indicates that the subprogram is passing a value generated in the subprogram out to the calling environment. The IN OUT mode means that a value is passed into the subprogram. The subprogram may change the value and pass the value out to the calling environment.

Practice 2-1: PL/SQL Review (continued)

5) How does a function differ from a procedure?

A function must execute a RETURN statement that returns a value. Functions are called differently than procedures. They are called as an expression embedded within another command. Procedures are called as statements.

6) What are the two main components of a PL/SQL package?

The package body and the package specification

a) In what order are they defined?

First the package specification, and then the package body

b) Are both required?

No, only a package specification is required. A specification can exist without a body, but a body cannot exist as valid without the specification.

7) How does the syntax of a SELECT statement used within a PL/SQL block differ from a SELECT statement issued in SQL*Plus?

The INTO clause is required with a SELECT statement that is in a PL/SQL subprogram.

8) What is a record?

A record is a composite type that has internal components, which can be manipulated individually. Use the RECORD data type to treat related but dissimilar data as a logical unit.

9) What is an index-by table?

Index-by tables are a data structure declared in a PL/SQL block. It is similar to an array and comprises two components—the index and the data field. The data field is a column of a scalar or record data type, which stores the INDEX BY table elements.

10) How are loops implemented in PL/SQL?

Looping constructs are used to repeat a statement or sequence of statements multiple times. PL/SQL has three looping constructs:

- Basic loops that perform repetitive actions without overall conditions
- FOR loops that perform iterative control of actions based on a count
- WHILE loops that perform iterative control of actions based on a condition

11) How is branching logic implemented in PL/SQL?

You can change the logical flow of statements within the PL/SQL block with a number of control structures. Branching logic is implemented within PL/SQL by using the conditional IF statement or CASE expressions.

Practice 2-1: PL/SQL Review (continued)

Cursor Basics

12) What is an explicit cursor?

The Oracle server uses work areas, called private SQL areas, to execute SQL statements and to store processing information. You can use PL/SQL cursors to name a private SQL area and access its stored information. Use explicit cursors to individually process each row returned by a multiple-row SELECT statement.

13) Where do you define an explicit cursor?

A cursor is defined in the declarative section.

14) Name the five steps for using an explicit cursor.

Declare, Open, Fetch, Test for existing rows, and Close

15) What is the syntax used to declare a cursor?

CURSOR cursor_name IS SELECT_statement

16) What does the FOR UPDATE clause do within a cursor definition?

The FOR UPDATE clause locks the rows selected in the SELECT statement definition of the cursor.

17) What command opens an explicit cursor?

OPEN cursor_name;

18) What command closes an explicit cursor?

CLOSE cursor_name;

19) Name five implicit actions that a cursor FOR loop provides.

Declares a record structure to match the select list of the cursor; opens the cursor; fetches from the cursor; exits the loop when the fetch returns no row; and closes the cursor

20) Describe what the following cursor attributes do:

%ISOPEN: Returns a Boolean value indicating whether the cursor is open

%FOUND: Returns a Boolean value indicating whether the last fetch returned a value

%NOTFOUND: Returns a Boolean value indicating whether the last fetch did not return a value

%ROWCOUNT: Returns an integer indicating the number of rows fetched so far

Practice 2-1: PL/SQL Review (continued)

Exceptions

- 21) An exception occurs in your PL/SQL block, which is enclosed in another PL/SQL block. What happens to this exception?

Control is passed to the exception handler. If the exception is handled in the inner block, processing continues to the outer block. If the exception is not handled in the inner block, an exception is raised in the outer block and control is passed to the exception handler of the outer block. If neither the inner nor the outer block traps the exception, the program ends unsuccessfully.

- 22) An exception handler is mandatory within a PL/SQL subprogram. (True/False)

False

- 23) What syntax do you use in the exception handler area of a subprogram?

```
EXCEPTION
    WHEN named_exception THEN
        statement[s];
    WHEN others THEN
        statement[s];
END;
```

- 24) How do you code for a NO_DATA_FOUND error?

```
EXCEPTION
    WHEN no_data_found THEN
        statement[s];
END;
```

- 25) Name three types of exceptions.

User-defined, Oracle server predefined, and Oracle server non-predefined

- 26) To associate an exception identifier with an Oracle error code, what pragma do you use and where?

Use the PRAGMA EXCEPTION_INIT and place the PRAGMA EXCEPTION_INIT in the declarative section.

- 27) How do you explicitly raise an exception?

Use the RAISE statement or the raise_application_error procedure.

- 28) What types of exceptions are implicitly raised?

All Oracle server exceptions (predefined and non-predefined) are automatically raised.

- 29) What does the RAISE_APPLICATION_ERROR procedure do?

It enables you to issue user-defined error messages from subprograms.

Practice 2-1: PL/SQL Review (continued)

Dependencies

30) Which objects can a procedure or function directly reference?

Table, view, sequence, procedure, function, package specification, object specification, and collection type

31) What are the two statuses that a schema object can have and where are they recorded?

The user_objects dictionary view contains a column called status. Its values are VALID and INVALID.

32) The Oracle server automatically recompiles invalid procedures when they are called from the same _____. To avoid compile problems with remote database calls, you can use the _____ model instead of the timestamp model.

**database
signature**

33) Which data dictionary contains information on direct dependencies?

user_dependencies

34) Which script do you run to create the views deptree and ideptree?

You use the utldtree.sql script.

35) What does the deptree_fill procedure do and what are the arguments that you need to provide?

The deptree_fill procedure populates the deptree and ideptree views to display a tabular representation of all dependent objects, direct and indirect. You pass the object type, object owner, and object name to the deptree_fill procedure.

36) What does the dbms_output package do?

The dbms_output package enables you to send messages from stored procedures, packages, and triggers.

37) How do you write “This procedure works.” from within a PL/SQL program by using dbms_output?

DBMS_OUTPUT.PUT_LINE('This procedure works.');

38) What does dbms_sql do and how does it compare with Native Dynamic SQL?

dbms_sql enables you to embed dynamic data manipulation language (DML), data definition language (DDL), and data control language (DCL) statements within a PL/SQL program. Native dynamic SQL allows you to place dynamic SQL statements directly into PL/SQL blocks. Native dynamic SQL in PL/SQL is easier to use than dbms_sql, requires much less application code, and performs better.

Practices for Lesson 3

Practice 3-1: Designing PL/SQL Code

In this practice, you determine the output of a PL/SQL code snippet and modify the snippet to improve the performance. Next, you implement subtypes and use cursor variables to pass values to and from a package. Files mentioned in the practice exercises are found in the /labs folder. Connect as OE to perform the steps.

- 1) Determine the output of the following code snippet in the lab_03_01.sql file.
Connect as OE.

```
SET SERVEROUTPUT ON
BEGIN
    UPDATE orders SET order_status = order_status;
    FOR v_rec IN ( SELECT order_id FROM orders )
        LOOP
            IF SQL%ISOPEN THEN
                DBMS_OUTPUT.PUT_LINE('TRUE - ' || SQL%ROWCOUNT);
            ELSE
                DBMS_OUTPUT.PUT_LINE('FALSE - ' || SQL%ROWCOUNT);
            END IF;
        END LOOP;
END;
/
```

Execute the code from the lab_03_01.sql file. It will show FALSE - 105 for each row fetched.

- 2) Modify the following snippet of code in the lab_03_02.sql file to make better use of the FOR UPDATE clause and improve the performance of the program.

```
DECLARE
    CURSOR cur_update
        IS SELECT * FROM customers
        WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
    FOR v_rec IN cur_update
        LOOP
            IF v_rec IS NOT NULL THEN
                UPDATE customers
                    SET credit_limit = credit_limit + 200
                    WHERE customer_id = v_rec.customer_id;
            END IF;
        END LOOP;
END;
/
```

Practice 3-1: Designing PL/SQL Code (continued)

Modify the lab_03_02.sql file as shown below:

```
DECLARE
  CURSOR cur_update
  IS SELECT * FROM customers
  WHERE credit_limit < 5000 FOR UPDATE;
BEGIN
  FOR v_rec IN cur_update
  LOOP
    UPDATE customers
    SET credit_limit = credit_limit + 200
    WHERE CURRENT OF cur_update;
  END LOOP;
END;
/
```

Alternatively, you can execute the code from the sol_03_02.sql file.

- 3) Create a package specification that defines subtypes, which can be used for the warranty_period field of the product_information table. Name this package MY_TYPES. The type needs to hold the month and year for a warranty period.

```
CREATE OR REPLACE PACKAGE mytypes
IS
  TYPE typ_warranty
    IS RECORD (month POSITIVE, year PLS_INTEGER);
  SUBTYPE warranty IS typ_warranty; -- based on RECORD type
END mytypes;
/
```

- 4) Create a package named SHOW_DETAILS that contains two subroutines. The first subroutine should show order details for the given order_id. The second subroutine should show customer details for the given customer_id, including the customer ID, first name, phone numbers, credit limit, and email address. Both subroutines should use the cursor variable to return the necessary details.

Practice 3-1: Designing PL/SQL Code (continued)

```
CREATE OR REPLACE PACKAGE show_details AS  
  
TYPE rt_order IS REF CURSOR RETURN orders%ROWTYPE;  
  
TYPE typ_cust_rec IS RECORD  
(cust_id NUMBER(6), cust_name VARCHAR2(20),  
 custphone customers.phone_numbers%TYPE,  
 credit NUMBER(9,2), cust_email VARCHAR2(30));  
TYPE rt_cust IS REF CURSOR RETURN typ_cust_rec;  
  
PROCEDURE get_order(p_orderid IN NUMBER, p_cv_order IN OUT  
 rt_order);
```

```
PROCEDURE get_cust(p_custid IN NUMBER, p_cv_cust IN OUT  
 rt_cust);  
END show_details;  
/  
  
CREATE OR REPLACE PACKAGE BODY show_details AS  
PROCEDURE get_order  
(p_orderid IN NUMBER, p_cv_order IN OUT rt_order)  
IS  
BEGIN  
    OPEN p_cv_order FOR  
        SELECT * FROM orders  
        WHERE order_id = p_orderid;  
    -- CLOSE p_cv_order  
END get_order;  
  
PROCEDURE get_cust  
(p_custid IN NUMBER, p_cv_cust IN OUT rt_cust)  
IS  
BEGIN  
    OPEN p_cv_cust FOR  
        SELECT customer_id, cust_first_name, phone_numbers,  
        credit_limit,  
        cust_email FROM customers  
        WHERE customer_id = p_custid;  
    -- CLOSE p_cv_cust  
END get_cust;  
END;  
/
```

Alternatively, you can execute the code from the `sol_03_04.sql` file.

Practices for Lesson 4

Practice 4-1: Working with Collections

In this practice, you create a nested table collection and use PL/SQL code to manipulate the collection.

Use your OE connection.

Analyzing Collections

- 1) Examine the following definitions. Run the `lab_04_01.sql` script to create these objects.

```
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
/
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
/
```

`@lab_04_01.sql`

- 2) The following code generates an error. Run the `lab_04_02.sql` script to generate and view the error.

```
BEGIN
  -- Insert an order
  INSERT INTO pOrder
    (ordid, supplier, requester, ordered, items)
    VALUES (1000, 12345, 9876, SYSDATE, NULL);
  -- insert the items for the order created
  INSERT INTO TABLE (SELECT items
                      FROM   pOrder
                     WHERE  ordid = 1000)
    VALUES(typ_item(99, 129.00));
END;
/
```

`@lab_04_02.sql`

Practice 4-1: Working with Collections (continued)

- a) Why does the error occur?

The error “ORA-22908: reference to NULL table value” results from setting the table columns to NULL.

- b) How can you fix the error?

You should always use a nested table’s default constructor to initialize it:

```
TRUNCATE TABLE pOrder;

-- A better approach is to avoid setting the table
-- column to NULL, and instead, use a nested table's
-- default constructor to initialize
BEGIN
    -- Insert an order
    INSERT INTO pOrder
        (ordid, supplier, requester, ordered, items)
        VALUES (1000, 12345, 9876, SYSDATE,
                typ_item_nst(typ_item(99, 129.00)));
END;
/

-- However, if the nested table is set to NULL, you can
-- use an UPDATE statement to set its value.
-- This is another alternative. Run either the block above,
-- or the block below to insert an order.

BEGIN
    -- Insert an order
    INSERT INTO pOrder
        (ordid, supplier, requester, ordered, items)
        VALUES (1000, 12345, 9876, SYSDATE, null);
    -- Once the nested table is set to null, use the update
    -- update statement
    UPDATE pOrder
        SET items = typ_item_nst(typ_item(99, 129.00))
        WHERE ordid = 1000;
END;
/
```

- 3) Examine the following code. This code produces an error. Which line causes the error, and how do you fix it?

(**Note:** You can run the lab_04_03.sql script to view the error output.)

Practice 4-1: Working with Collections (continued)

This causes an ORA-06531: Reference to uninitialized collection. To fix it, initialize the v_am variable by using the same technique as the others:

```
DECLARE
    TYPE credit_card_typ
    IS VARRAY(100) OF VARCHAR2(30);

    v_mc    credit_card_typ := credit_card_typ();
    v_visa  credit_card_typ := credit_card_typ();
    v_am    credit_card_typ := credit_card_typ();
    v_disc  credit_card_typ := credit_card_typ();
    v_dc    credit_card_typ := credit_card_typ();

BEGIN
    v_mc.EXTEND;
    v_visa.EXTEND;
    v_am.EXTEND;
    v_disc.EXTEND;
    v_dc.EXTEND;
END;
/
```

Using Collections

In the following practice exercises, you implement a nested table column in the CUSTOMERS table and write PL/SQL code to manipulate the nested table.

- 4) Create a nested table to hold credit card information.
 - a) Create an object type called typ_cr_card. It should have the following specification:

```
card_type  VARCHAR2(25)
card_num   NUMBER
```

```
CREATE TYPE typ_cr_card AS OBJECT --create object
(card_type  VARCHAR2(25),
 card_num   NUMBER);
/
```

- b) Create a nested table type called typ_cr_card_nst that is a table of typ_cr_card.

```
CREATE TYPE typ_cr_card_nst -- define nested table type
    AS TABLE OF typ_cr_card;
/
```

Practice 4-1: Working with Collections (continued)

- c) Add a column called `credit_cards` to the `CUSTOMERS` table. Make this column a nested table of type `typ_cr_card_nst`. You can use the following syntax:

```
ALTER TABLE customers ADD
(credit_cards typ_cr_card_nst)
    NESTED TABLE credit_cards STORE AD c_c_store_tab;
```

- 5) Create a PL/SQL package that manipulates the `credit_cards` column in the `CUSTOMERS` table.
- a) Open the `lab_04_05.sql` file. It contains the package specification and part of the package body.

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2,
         p_card_no VARCHAR2);
    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN
            -- cards exist, add more

            -- fill in code here
            ELSE -- no cards for this customer, construct one

            -- fill in code here
            END IF;
        END update_card_info;

    PROCEDURE display_card_info
        (p_cust_id NUMBER)
    IS
```

Practice 4-1: Working with Collections (continued)

```
v_card_info typ_cr_card_nst;
i INTEGER;
BEGIN
  SELECT credit_cards
    INTO v_card_info
    FROM customers
   WHERE customer_id = p_cust_id;

  -- fill in code here to display the nested table
  -- contents

  END display_card_info;
END credit_card_pkg;  -- package body
/
```

b) Complete the code so that the package:

- Inserts the credit card information (the credit card name and number for a specific customer)
- Displays the credit card information in an unnested format

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2);

  PROCEDURE display_card_info
    (p_cust_id NUMBER);
END credit_card_pkg;  -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

  PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
VARCHAR2)
  IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
  BEGIN
    SELECT credit_cards
      INTO v_card_info
      FROM customers
     WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN  -- cards exist, add more
      i := v_card_info.LAST;
      v_card_info.EXTEND(1);
      v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
      UPDATE customers
        SET credit_cards = v_card_info
    END IF;
  END;
END;
```

Practice 4-1: Working with Collections (continued)

```
    WHERE customer_id = p_cust_id;
ELSE    -- no cards for this customer yet, construct one
    UPDATE customers
        SET credit_cards = typ_cr_card_nst
            (typ_cr_card(p_card_type, p_card_no))
        WHERE customer_id = p_cust_id;
END IF;
END update_card_info;

PROCEDURE display_card_info
    (p_cust_id NUMBER)
IS
    v_card_info typ_cr_card_nst;
    i INTEGER;
BEGIN
    SELECT credit_cards
        INTO v_card_info
        FROM customers
        WHERE customer_id = p_cust_id;
    IF v_card_info.EXISTS(1) THEN
        FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
            DBMS_OUTPUT.PUT('Card Type: ' ||
v_card_info(idx).card_type || ' ');
            DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
v_card_info(idx).card_num );
            END LOOP;
    ELSE
        DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
END display_card_info;
END credit_card_pkg;  -- package body
/
```

- 6) Test your package with the following statements and compare the output:

```
EXECUTE credit_card_pkg.display_card_info(120)
Customer has no credit cards.
PL/SQL procedure successfully completed.

EXECUTE credit_card_pkg.update_card_info -
    (120, 'Visa', 11111111)
PL/SQL procedure successfully completed.

SELECT credit_cards
FROM   customers
WHERE  customer_id = 120;

CREDIT_CARDS(CARD_TYPE, CARD_NUM)
```

Practice 4-1: Working with Collections (continued)

```
-----  
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111))  
  
EXECUTE credit_card_pkg.display_card_info(120)  
Card Type: Visa / Card No: 11111111  
PL/SQL procedure successfully completed.  
  
EXECUTE credit_card_pkg.update_card_info -  
      (120, 'MC', 2323232323)  
PL/SQL procedure successfully completed.  
  
EXECUTE credit_card_pkg.update_card_info -  
      (120, 'DC', 4444444)  
PL/SQL procedure successfully completed.  
  
EXECUTE credit_card_pkg.display_card_info(120)  
Card Type: Visa / Card No: 11111111  
Card Type: MC / Card No: 2323232323  
Card Type: DC / Card No: 4444444  
PL/SQL procedure successfully completed.
```

```
SET SERVEROUT ON  
  
EXECUTE credit_card_pkg.display_card_info(120)  
  
EXECUTE credit_card_pkg.update_card_info(120, 'Visa',  
11111111)  
  
SELECT credit_cards  
FROM   customers  
WHERE  customer_id = 120;  
  
EXECUTE credit_card_pkg.display_card_info(120)  
  
EXECUTE credit_card_pkg.update_card_info(120, 'MC',  
2323232323)  
  
EXECUTE credit_card_pkg.update_card_info (120, 'DC', 4444444)  
EXECUTE credit_card_pkg.display_card_info(120)
```

- 7) Write a SELECT statement against the credit_cards column to unnest the data.
Use the TABLE expression. Use SQL*Plus.

For example, if the SELECT statement returns:

```
SELECT credit_cards  
FROM   customers
```

Practice 4-1: Working with Collections (continued)

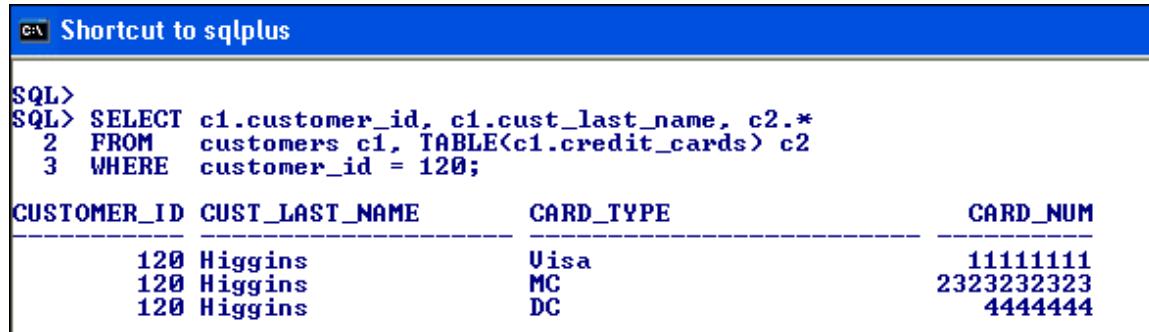
```
WHERE customer_id = 120;

CREDIT_CARDS(CARD_TYPE, CARD_NUM)
-----
TYP_CR_CARD_NST(TYP_CR_CARD('Visa', 11111111),
TYP_CR_CARD('MC', 2323232323), TYP_CR_CARD('DC', 4444444))
```

then rewrite it using the TABLE expression so that the results look like this:

```
-- Use the table expression so that the result is:
CUSTOMER_ID CUST_LAST_NAME CARD_TYPE CARD_NUM
-----
120 Higgins Visa 11111111
120 Higgins MC 2323232323
120 Higgins DC 4444444
```

```
SELECT c1.customer_id, c1.cust_last_name, c2.*
FROM   customers c1, TABLE(c1.credit_cards) c2
WHERE  customer_id = 120;
```



The screenshot shows the SQL*Plus command-line interface. The title bar says "Shortcut to sqlplus". The SQL prompt "SQL>" is followed by the query:

```
SQL> SELECT c1.customer_id, c1.cust_last_name, c2.*
  2  FROM   customers c1, TABLE(c1.credit_cards) c2
  3 WHERE  customer_id = 120;
```

Below the query, the results are displayed in a tabular format:

CUSTOMER_ID	CUST_LAST_NAME	CARD_TYPE	CARD_NUM
120	Higgins	Visa	11111111
120	Higgins	MC	2323232323
120	Higgins	DC	4444444

Practices for Lesson 5

Practice 5-1: Using Advanced Interface Methods Using External C Routines

An external C routine definition is created for you. The .c file is stored in the D:\labs\labs directory. This function returns the tax amount based on the total sales figure that is passed to the function as a parameter. The .c file is named calc_tax.c.

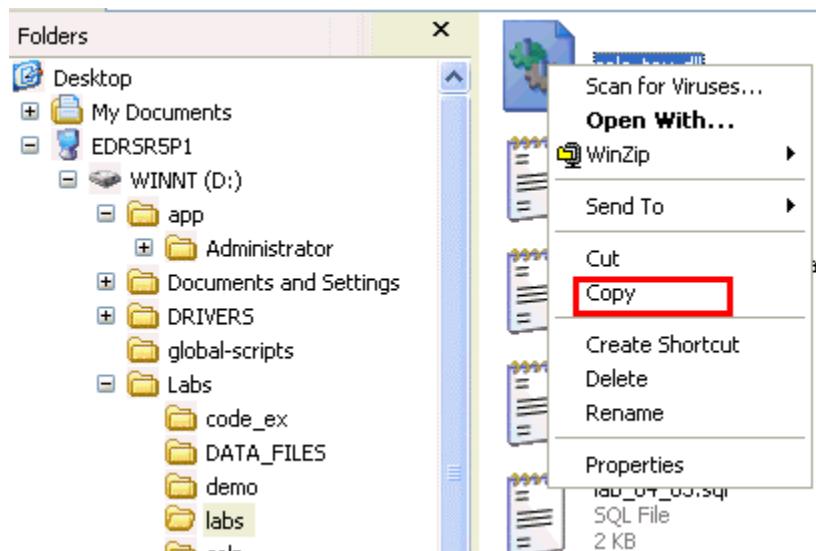
Use your OE connection.

The function is defined as:

```
_declspec(dllexport)
int calc_tax(n)
int n;
{
    int tax;
    tax = (n*8)/100;
    return (tax);
}
```

- 1) A DLL file called calc_tax.dll was created for you. Copy the file from the D:\labs\labs directory into your D:\app\Administrator\product\11.1.0\db_1\BIN directory:

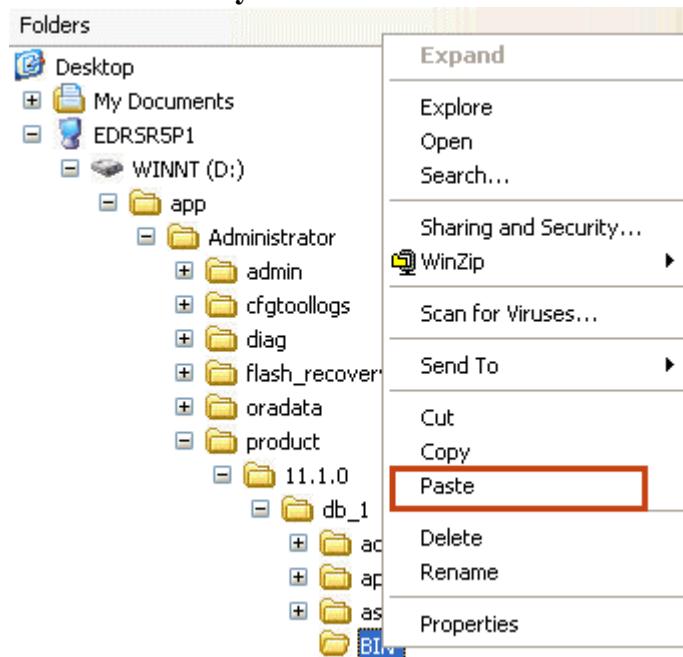
In Explorer, right click the calc_tax.dll file and select Copy from the submenu.



In Explorer, navigate to the D:\app\Administrator\product\11.1.0\db_1\BIN folder. Right-click

Practice 5-1: Using Advanced Interface Methods (continued)

the **BIN** directory and select Paste from the submenu.



- 2) As the SYS user, create the alias library object. Name the library object `c_code` and define its path as:

```
connect / as sysdba
```

```
CREATE OR REPLACE LIBRARY c_code AS  
'd:\app\Administrator\product\11.1.0\db_1\BIN\calc_tax.dll'  
;
```

- 3) Grant execute privilege on the library to the OE user by executing the following command:

```
GRANT EXECUTE ON c_code TO OE;
```

- 4) Publish the external C routine.

As the OE user, create a function named `call_c`. This function has one numeric parameter and it returns a binary integer. Identify the AS LANGUAGE, LIBRARY, and NAME clauses of the function.

```
CONNECT oe

CREATE OR REPLACE FUNCTION call_c
(x BINARY_INTEGER)
RETURN BINARY_INTEGER
AS LANGUAGE C
LIBRARY sys.c_code
NAME "calc_tax";
/
```

Practice 5-1: Using Advanced Interface Methods (continued)

- 5) Create a procedure to call the `call_c` function created in the previous step.

Name this procedure `C_OUTPUT`. It has one numeric parameter. Include a `DBMS_OUTPUT.PUT_LINE` statement so that you can view the results returned from your C function.

```
CREATE OR REPLACE PROCEDURE c_output
  (p_in IN BINARY_INTEGER)
IS
  i BINARY_INTEGER;
BEGIN
  i := call_c(p_in);
  DBMS_OUTPUT.PUT_LINE('The total tax is: ' || i);
END c_output;
/
```

- 6) Set `SERVOROUTPUT ON` and execute the `C_OUTPUT` procedure.

```
SET SERVEROUT ON
EXECUTE c_output(1000000)
```

```
The total tax is: 80000
```

Calling Java from PL/SQL

A Java method definition is created for you. The method accepts a 16-digit credit card number as the argument and returns the formatted credit card number (4 digits followed by a space). The name of the `.class` file is `FormatCreditCardNo.class`. The method is defined as:

```
public class FormatCreditCardNo
{
  public static final void formatCard(String[] cardno)
  {
    int count=0, space=0;
    String oldcc=cardno[0];
    String[] newcc= {" "};
    while (count<16)
    {
      newcc[0]+= oldcc.charAt(count);
      space++;
      if (space ==4)
      { newcc[0]+=" "; space=0; }
      count++;
    }
    cardno[0]=newcc [0];
  }
}
```

Practice 5-1: Using Advanced Interface Methods (continued)

- 7) Load the .java source file.

```
D:\app\Administrator\product\11.1.0\db_1\BIN>cd D:\Labs\labs  
D:\Labs\labs>loadjava -user oe/oe FormatCreditCardNo.java  
D:\Labs\labs>_
```

- 8) Publish the Java class method by defining a PL/SQL procedure named CCFORMAT. This procedure accepts one IN OUT parameter.

Use the following definition for the NAME parameter:

```
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
```

```
CREATE OR REPLACE PROCEDURE ccformat  
(x IN OUT VARCHAR2)  
AS LANGUAGE JAVA  
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';  
/
```

- 9) Execute the Java class method. Define one SQL*Plus or **Oracle SQL Developer** variable, initialize it, and use the EXECUTE command to execute the CCFORMAT procedure. Your output should match the PRINT output shown below.

```
EXECUTE ccformat(:x);
```

```
PRINT x  
-----  
1234 5678 1234 5678
```

```
VARIABLE x VARCHAR2(20)  
EXECUTE :x := '1234567887654321'  
  
EXECUTE ccformat(:x)  
  
PRINT x
```

Practices for Lesson 6

Practice 6-1: Implementing Fine-Grained Access Control for VPD

In this practice, you define an application context and security policy to implement the policy: “Sales representatives can see their own order information only in the ORDERS table.” You create sales representative IDs to test the success of your implementation. Examine the definition of the ORDERS table and the ORDER count for each sales representative:

- 1) Use your OE connection.

Examine and then run the `lab_06_01.sql` script. This script creates the ID accounts of sales representatives with appropriate privileges to access the database:

```
CONNECT /AS sysdba

CREATE USER sr153 IDENTIFIED BY oracle
DEFAULT TABLESPACE USERS
TEMPORARY TABLESPACE TEMP
QUOTA UNLIMITED ON USERS;

CREATE USER sr154 IDENTIFIED BY oracle
DEFAULT TABLESPACE USERS
TEMPORARY TABLESPACE TEMP
QUOTA UNLIMITED ON USERS;

CREATE USER sr155 IDENTIFIED BY oracle
DEFAULT TABLESPACE USERS
TEMPORARY TABLESPACE TEMP
QUOTA UNLIMITED ON USERS;

CREATE USER sr156 IDENTIFIED BY oracle
DEFAULT TABLESPACE USERS
TEMPORARY TABLESPACE TEMP
QUOTA UNLIMITED ON USERS;

CREATE USER sr158 IDENTIFIED BY oracle
DEFAULT TABLESPACE USERS
TEMPORARY TABLESPACE TEMP
QUOTA UNLIMITED ON USERS;

CREATE USER sr159 IDENTIFIED BY oracle
DEFAULT TABLESPACE USERS
TEMPORARY TABLESPACE TEMP
QUOTA UNLIMITED ON USERS;

CREATE USER sr160 IDENTIFIED BY oracle
DEFAULT TABLESPACE USERS
TEMPORARY TABLESPACE TEMP
```

Practice 6-1: Implementing Fine-Grained Access Control for VPD (continued)

```
QUOTA UNLIMITED ON USERS;

CREATE USER sr161 IDENTIFIED BY oracle
DEFAULT TABLESPACE USERS
TEMPORARY TABLESPACE TEMP
QUOTA UNLIMITED ON USERS;

CREATE USER sr163 IDENTIFIED BY oracle
DEFAULT TABLESPACE USERS
TEMPORARY TABLESPACE TEMP
QUOTA UNLIMITED ON USERS;

GRANT create session
    , alter session
TO sr153, sr154, sr155, sr156, sr158, sr159,
    sr160, sr161, sr163;

GRANT SELECT, INSERT, UPDATE, DELETE ON
    oe.orders TO sr153, sr154, sr155, sr156, sr158,
    sr159, sr160, sr161, sr163;

GRANT SELECT, INSERT, UPDATE, DELETE ON
    oe.order_items TO sr153, sr154, sr155, sr156, sr158,
    sr159, sr160, sr161, sr163;

CREATE PUBLIC SYNONYM orders FOR oe.orders;

CREATE PUBLIC SYNONYM order_items FOR oe.order_items;

CONNECT oe/oe
```

```
@lab_06_01.sql
```

- 2) Set up an application context:
 - a) Connect to the database as SYSDBA before creating this context.
 - b) Create an application context named sales_orders_ctx.
 - c) Associate this context with the oe.sales_orders_pkg.

```
CONNECT /AS sysdba

CREATE CONTEXT sales_orders_ctx
USING oe.sales_orders_pkg;
```

Practice 6-1: Implementing Fine-Grained Access Control for VPD (continued)

3) Connect as OE.

a) Examine this package specification:

```
CREATE OR REPLACE PACKAGE sales_orders_pkg
IS
  PROCEDURE set_app_context;
  FUNCTION the_predicate
    (p_schema VARCHAR2, p_name VARCHAR2)
    RETURN VARCHAR2;
END sales_orders_pkg;      -- package spec
/
```

b) Create this package specification, and then the package body in the OE schema.

c) When you create the package body, set up two constants as follows:

```
c_context CONSTANT VARCHAR2(30) := 'SALES_ORDERS_CTX';
c_attrib  CONSTANT VARCHAR2(30) := 'SALES REP';
```

d) Use these constants in the SET_APP_CONTEXT procedure to set the application context to the current user.

```
CREATE OR REPLACE PACKAGE BODY sales_orders_pkg
IS
  c_context CONSTANT VARCHAR2(30) := 'SALES_ORDERS_CTX';
  c_attrib  CONSTANT VARCHAR2(30) := 'SALES REP';

  PROCEDURE set_app_context
  IS
    v_user VARCHAR2(30);
  BEGIN
    SELECT user INTO v_user FROM dual;
    DBMS_SESSION.SET_CONTEXT
      (c_context, c_attrib, v_user);
  END set_app_context;

  FUNCTION the_predicate
  (p_schema VARCHAR2, p_name VARCHAR2)
  RETURN VARCHAR2
  IS
    v_context_value VARCHAR2(100) :=
      SYS_CONTEXT(c_context, c_attrib);
    v_restriction VARCHAR2(2000);
  BEGIN
    IF v_context_value LIKE 'SR%' THEN
      v_restriction := 'SALES REP_ID ='
```

Practice 6-1: Implementing Fine-Grained Access Control for VPD (continued)

```
SUBSTR('' || v_context_value || '', 3, 3)';
ELSE
  v_restriction := null;
END IF;
RETURN v_restriction;
END the_predicate;

END sales_orders_pkg; -- package body
/
```

- 4) Connect as SYSDBA and define the policy.
 - a) Use DBMS_RLS.ADD_POLICY to define the policy.
 - b) Use the following specifications for the parameter values:

```
object_schema    OE
object_name      ORDERS
policy_name      OE_ORDERS_ACCESS_POLICY
function_schema  OE
policy_function  SALES_ORDERS_PKG.THE_PREDICATE
statement_types   SELECT, INSERT, UPDATE, DELETE
update_check     FALSE,
enable           TRUE
```

```
CONNECT /as sysdba

DECLARE
BEGIN
  DBMS_RLS.ADD_POLICY (
    'OE',
    'ORDERS',
    'OE_ORDERS_ACCESS_POLICY',
    'OE',
    'SALES_ORDERS_PKG.THE_PREDICATE',
    'SELECT, INSERT, UPDATE, DELETE',
    FALSE,
    TRUE);
END;
/
```

- 5) Connect as SYSDBA and create a logon trigger to implement the fine-grained access control. You can call the SET_ID_ON_LOGON trigger. This trigger causes the context to be set as each user is logged on.

```
CONNECT /as sysdba

CREATE OR REPLACE TRIGGER set_id_on_logon
```

Practice 6-1: Implementing Fine-Grained Access Control for VPD (continued)

```
AFTER logon on DATABASE
BEGIN
    oe.sales_orders_pkg.set_app_context;
END;
/
```

- 6) Test the fine-grained access implementation. Connect as your SR user and query the ORDERS table. For example, your results should match the following:

```
CONNECT sr153/oracle

SELECT sales_rep_id, COUNT(*)
FROM oe.orders
GROUP BY sales_rep_id;

SALES REP ID      COUNT(*)
----- -----
      153            5

CONNECT sr154/oracle

SELECT sales_rep_id, COUNT(*)
FROM oe.orders
GROUP BY sales_rep_id;

SALES REP ID      COUNT(*)
----- -----
      154            10
```

Notes

During debugging, you may need to disable or remove some of the objects created for this lesson.

If you need to disable the logon trigger, issue the following command:

```
ALTER TRIGGER set_id_on_logon DISABLE;
```

If you need to remove the policy that you created, issue the following command:

```
EXECUTE DBMS_RLS.DROP_POLICY('OE', 'ORDERS', -
'OE_ORDERS_ACCESS_POLICY')
```

Practices for Lesson 7

Practice 7-1: Manipulating Large Objects

Working with LOBs

In this practice, you create a table with both BLOB and CLOB columns. Then, you use the DBMS_LOB package to populate the table and manipulate the data.

Use your OE connection.

- 1) Create a table called PERSONNEL by executing the D:\labs\labs\lab_07_01.sql script file. The table contains the following attributes and data types:

Column Name	Data Type	Length
ID	NUMBER	6
LAST_NAME	VARCHAR2	35
REVIEW	CLOB	N/A
PICTURE	BLOB	N/A

```
CREATE TABLE personnel
(id NUMBER(6) constraint personnel_id_pk PRIMARY KEY,
 last_name VARCHAR2(35),
 review CLOB,
 picture BLOB);
```

```
@D:\labs\labs\lab_07_01.sql
```

- 2) Insert two rows into the PERSONNEL table, one each for employee 2034 (whose last name is Allen) and employee 2035 (whose last name is Bond). Use the empty function for the CLOB, and provide NULL as the value for the BLOB.

```
INSERT INTO personnel
VALUES (2034, 'Allen', empty_clob(), NULL);

INSERT INTO personnel
VALUES (2035, 'Bond', empty_clob(), NULL);
```

Practice 7-1: Manipulating Large Objects (continued)

- 3) Examine and execute the D:\labs\labs\lab_07_03.sql script. The script creates a table named REVIEW_TABLE. This table contains annual review information for each employee. The script also contains two statements to insert review details for two employees.

```
CREATE TABLE review_table (
    employee_id number,
    ann_review  VARCHAR2(2000));

INSERT INTO review_table
VALUES (2034,
        'Very good performance this year. ||
        Recommended to increase salary by $500');

INSERT INTO review_table
VALUES (2035,
        'Excellent performance this year. ||
        Recommended to increase salary by $1000');

COMMIT;
```

```
@D:\labs\labs\lab_07_03.sql
```

- 4) Update the PERSONNEL table.

- a) Populate the CLOB for the first row by using the following subquery in an UPDATE statement:

```
SELECT ann_review
FROM   review_table
WHERE  employee_id = 2034;
```

```
UPDATE personnel
SET review = (SELECT ann_review
              FROM   review_table
              WHERE  employee_id = 2034)
WHERE last_name = 'Allen';
```

- b) Populate the CLOB for the second row by using PL/SQL and the DBMS_LOB package. Use this SELECT statement to provide a value for the LOB locator:

```
SELECT ann_review
FROM   review_table
WHERE  employee_id = 2035;
```

```
UPDATE personnel
SET review = (SELECT ann_review
              FROM   review_table
              WHERE  employee_id = 2035)
WHERE last_name = 'Bond';
```

Practice 7-1: Manipulating Large Objects (continued)

- 5) Create a procedure that adds a locator to a binary file to the PICTURE column of the PRODUCT_INFORMATION table. The binary file is a picture of the Product. The image files are named after the Product IDs. You need to load an image file locator into all rows in the Printers category (CATEGORY_ID = 12) in the PRODUCT_INFORMATION table.
- Create a DIRECTORY object called PRODUCT_PIC that references the location of the binary file. These files are available in the D:\labs\labs\DATA_FILES\PRODUCT_PIC folder.

```
CREATE DIRECTORY product_pic AS 'D:\labs\labs\DEMO_FILES\PRODUCT_PIC';
```

(Alternatively, use the D:\labs\labs\lab_07_05a.sql script.)
 - Add the image column to the PRODUCT_INFORMATION table by using:

```
ALTER TABLE product_information ADD (picture BFILE);
```

(Alternatively, use the D:\labs\labs\lab_07_05_b.sql file.)
 - Create a PL/SQL procedure called load_product_image that uses DBMS_LOB.FILEEXISTS to test whether the product picture file exists. If the file exists, set the BFILE locator for the file in the PICTURE column; otherwise, display a message that the file does not exist. Use the DBMS_OUTPUT package to report file size information for each image associated with the PICTURE column.
(Alternatively, use the D:\labs\labs\lab_07_05_c.sql file)

```
CREATE OR REPLACE PROCEDURE load_product_image
(p_dir IN VARCHAR2)
IS
v_file          BFILE;
v_filename      VARCHAR2(40);
v_rec_number    NUMBER;
v_file_exists   BOOLEAN;
CURSOR product_csr IS
  SELECT product_id
  FROM product_information
  WHERE category_id = 12
  FOR UPDATE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('LOADING LOCATORS TO IMAGES... ');
  FOR rec IN product_csr
  LOOP
    v_filename := rec.product_id || '.gif';
    v_file := BFILENAME(p_dir, v_filename);
    v_file_exists := (DBMS_LOB.FILEEXISTS(v_file) = 1);
    IF v_file_exists THEN
      DBMS_LOB.FILEOPEN(v_file);
      UPDATE product_information
      SET picture = v_file
    END IF;
  END LOOP;
END;
```

Practice 7-1: Manipulating Large Objects (continued)

```
        WHERE CURRENT OF product_csr;
        DBMS_OUTPUT.PUT_LINE('Set Locator to file: ' ||
v_filename
          || ' Size: ' || DBMS_LOB.GETLENGTH(v_file));
        DBMS_LOB.FILECLOSE(v_file);
        v_rec_number := product_csr%ROWCOUNT;
      ELSE
        DBMS_OUTPUT.PUT_LINE('File ' || v_filename ||
          ' does not exist');
      END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('TOTAL FILES UPDATED: ' ||
          v_rec_number);

EXCEPTION
  WHEN OTHERS THEN
    DBMS_LOB.FILECLOSE(v_file);
    DBMS_OUTPUT.PUT_LINE('Error: ' || to_char(SQLCODE) ||
          SQLERRM);
END load_product_image;
/
```

- d) Invoke the procedure by passing the name of the directory object PRODUCT_PICTURE as a string literal parameter value.

```
SET SERVEROUTPUT ON
EXECUTE load_product_image('PRODUCT_PICTURE');
```

```
LOADING LOCATORS TO IMAGES...
Set Locator to file: 1797.gif Size: 7888
Set Locator to file: 2459.gif Size: 9587
Set Locator to file: 3127.gif Size: 9587
Set Locator to file: 1782.gif Size: 7888
Set Locator to file: 2430.gif Size: 7462
Set Locator to file: 1792.gif Size: 7462
Set Locator to file: 1791.gif Size: 7462
Set Locator to file: 2302.gif Size: 7462
Set Locator to file: 2453.gif Size: 9587
TOTAL FILES UPDATED: 9
```

- e) Check the LOB space usage of the PRODUCT_INFORMATION table. Use the D:\labs\labs\lab_07_05_e.sql file to create the procedure and execute it.

```
CREATE OR REPLACE PROCEDURE check_space
IS
  l_fs1_bytes NUMBER;
  l_fs2_bytes NUMBER;
  l_fs3_bytes NUMBER;
  l_fs4_bytes NUMBER;
  l_fs1_blocks NUMBER;
  l_fs2_blocks NUMBER;
  l_fs3_blocks NUMBER;
  l_fs4_blocks NUMBER;
  l_full_bytes NUMBER;
  l_full_blocks NUMBER;
```

Practice 7-1: Manipulating Large Objects (continued)

```
l_unformatted_bytes NUMBER;
l_unformatted_blocks NUMBER;
BEGIN
    DBMS_SPACE.SPACE_USAGE(
        segment_owner      => 'OE',
        segment_name       => 'PRODUCT_INFORMATION',
        segment_type       => 'TABLE',
        fs1_bytes          => l_fs1_bytes,
        fs1_blocks         => l_fs1_blocks,
        fs2_bytes          => l_fs2_bytes,
        fs2_blocks         => l_fs2_blocks,
        fs3_bytes          => l_fs3_bytes,
        fs3_blocks         => l_fs3_blocks,
        fs4_bytes          => l_fs4_bytes,
        fs4_blocks         => l_fs4_blocks,
        full_bytes         => l_full_bytes,
        full_blocks        => l_full_blocks,
        unformatted_blocks => l_unformatted_blocks,
        unformatted_bytes  => l_unformatted_bytes
    );
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = '||l_fs1_blocks||'
                         Bytes = '||l_fs1_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = '||l_fs2_blocks||'
                         Bytes = '||l_fs2_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS3 Blocks = '||l_fs3_blocks||'
                         Bytes = '||l_fs3_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS4 Blocks = '||l_fs4_blocks||'
                         Bytes = '||l_fs4_bytes);
    DBMS_OUTPUT.PUT_LINE('Full Blocks = '||l_full_blocks||'
                         Bytes = '||l_full_bytes);

    DBMS_OUTPUT.PUT_LINE('=====');
    DBMS_OUTPUT.PUT_LINE('Total Blocks =
                        '||to_char(l_fs1_blocks + l_fs2_blocks +
                         l_fs3_blocks + l_fs4_blocks + l_full_blocks)||' ||
                         Total Bytes = '|| to_char(l_fs1_bytes + l_fs2_bytes
                         + l_fs3_bytes + l_fs4_bytes + l_full_bytes));
END;
/
```

Practice 7-1: Manipulating Large Objects (continued)

```
set serveroutput on
execute check_space;
```

```
FS1 Blocks = 0      -
  Bytes = 0
FS2 Blocks = 0
  Bytes = 0
FS3 Blocks = 0
  Bytes = 0
FS4 Blocks = 4
  Bytes = 32768
Full Blocks = 9
  Bytes = 73728
=====
Total Blocks =
  13 !!
Total Bytes = 106496
```

Practices for Lesson 8

Practice 8-1: Using SecureFile Format LOBS

Working with SecureFile LOBS

In this lesson, you practice using the features of the SecureFile format LOBs.

- 1) In this practice, you migrate a BasicFile format LOB to a SecureFile format LOB.
You need to set up several supporting structures:

- a) As the OE user, drop your existing PRODUCT_DESCRIPTIONS table and create a new one:

```
DROP TABLE product_descriptions PURGE;
```

```
CREATE TABLE product_descriptions  
(product_id NUMBER);
```

```
connect oe
```

```
DROP TABLE product_descriptions PURGE;
```

```
CREATE TABLE product_descriptions  
(product_id NUMBER);
```

- b) As the SYS user, create a new tablespace to store the LOB information.

```
CREATE TABLESPACE lob_tbs2  
DATAFILE 'lob_tbs2.dbf' SIZE 1500M REUSE  
AUTOEXTEND ON NEXT 200M  
MAXSIZE 3000M  
SEGMENT SPACE MANAGEMENT AUTO;
```

```
connect /as sysdba
```

```
CREATE TABLESPACE lob_tbs2  
DATAFILE 'lob_tbs2.dbf' SIZE 1500M REUSE  
AUTOEXTEND ON NEXT 200M  
MAXSIZE 3000M  
SEGMENT SPACE MANAGEMENT AUTO;
```

- c) Create a directory object that identifies the location of your LOBs. In the Oracle classroom, the location is in the Oracle D:\labs\labs\DATA_FILES\PRODUCT_PIC folder. Then, grant read privileges on the directory to the OE user.

```
CREATE OR REPLACE DIRECTORY product_files  
AS 'd:\Labs\DATA_FILES\PRODUCT_PIC ';
```

```
GRANT READ ON DIRECTORY product_files TO oe;
```

Practice 8-1: Using SecureFile Format LOBs (continued)

```
connect /as sysdba

CREATE OR REPLACE DIRECTORY product_files
AS 'D:\labs\DATA_FILES\PRODUCT_PIC';

GRANT READ ON DIRECTORY product_files TO oe;
```

- d) As the OE user, alter the table and add a BLOB column of the BASICFILE storage type.

```
ALTER TABLE product_descriptions ADD
(detailed_product_info BLOB )
LOB (detailed_product_info) STORE AS BASICFILE
(tablespace lob_tbs2);
```

```
connect oe

ALTER TABLE product_descriptions ADD
(detailed_product_info BLOB )
LOB (detailed_product_info)
STORE AS BASICFILE (tablespace lob_tbs2);
```

- e) Create the procedure to load the LOB data into the column (You can run the D:\labs\labs\lab_08_01_e.sql script):

```
CREATE OR REPLACE PROCEDURE loadLOBFromBFILE_proc (
    p_dest_loc IN OUT BLOB, p_file_name IN VARCHAR2)
IS
    v_src_loc BFILE := BFILENAME('PRODUCT_FILES',
p_file_name);
    v_amount    INTEGER := 4000;
BEGIN
    DBMS_LOB.OPEN(v_src_loc, DBMS_LOB.LOB_READONLY);
    v_amount := DBMS_LOB.GETLENGTH(v_src_loc);
    DBMS_LOBLOADFROMFILE(p_dest_loc, v_src_loc, v_amount);
    DBMS_LOB CLOSE(v_src_loc);
END loadLOBFromBFILE_proc;
/
```

```
connect oe

CREATE OR REPLACE PROCEDURE loadLOBFromBFILE_proc (
    p_dest_loc IN OUT BLOB, p_file_name IN VARCHAR2)
IS
    v_src_loc BFILE := BFILENAME('PRODUCT_FILES',
p_file_name);
    v_amount    INTEGER := 4000;
BEGIN
    DBMS_LOB.OPEN(v_src_loc, DBMS_LOB.LOB_READONLY);
    v_amount := DBMS_LOB.GETLENGTH(v_src_loc);
```

Practice 8-1: Using SecureFile Format LOBs (continued)

```
DBMS_LOB.LOADFROMFILE(p_dest_loc, v_src_loc, v_amount);
DBMS_LOB CLOSE(v_src_loc);
END loadLOBFromBFILE_proc;
/
```

- f) As the OE user, create the procedure to write the LOB data (You can run the D:\labs\labs\lab_08_01_f.sql script):

```
CREATE OR REPLACE PROCEDURE write_lob (p_file IN
VARCHAR2)
IS
    i      NUMBER;    v_id NUMBER;    v_b    BLOB;
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE('Begin inserting rows...');

    FOR i IN 1 .. 5 LOOP
        v_id:=SUBSTR(p_file, 1, 4);
        INSERT INTO product_descriptions
            VALUES (v_id, EMPTY_BLOB())
            RETURNING detailed_product_info INTO v_b;
        loadLOBFromBFILE_proc(v_b,p_file);
        DBMS_OUTPUT.PUT_LINE('Row '|| i ||' inserted.');
    END LOOP;
    COMMIT;
END write_lob;
/
```

```
connect oe
```

```
CREATE OR REPLACE PROCEDURE write_lob (p_file IN VARCHAR2)
IS
    i      NUMBER;    v_id NUMBER;    v_b    BLOB;
BEGIN
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE('Begin inserting rows...');

    FOR i IN 1 .. 5 LOOP
        v_id:=SUBSTR(p_file, 1, 4);
        INSERT INTO product_descriptions
            VALUES (v_id, EMPTY_BLOB())
            RETURNING detailed_product_info INTO v_b;
        loadLOBFromBFILE_proc(v_b,p_file);
        DBMS_OUTPUT.PUT_LINE('Row '|| i ||' inserted.');
    END LOOP;
    COMMIT;
END write_lob;
/
```

Practice 8-1: Using SecureFile Format LOBs (continued)

- g) As the OE user, execute the procedures to load the data. If you are using SQL*Plus, you can set the timing on to observe the time. (You can run the D:\labs\labs\lab_08_01_g.sql script.)

Note: If you are using **Oracle SQL Developer**, issue only the EXECUTE statements listed as follows. In **Oracle SQL Developer**, some of the SQL*Plus commands are ignored. It is recommended that you use SQL*Plus for this exercise.

```
set serveroutput on
set verify on
set term on
set lines 200
timing start load_data
execute write_lob('1726_LCD.doc');
execute write_lob('1734_RS232.doc');
execute write_lob('1739_SDRAM.doc');
timing stop
```

```
connect oe
set serveroutput on
set verify on
set term on
set lines 200
timing start load_data
execute write_lob('1726_LCD.doc');
execute write_lob('1734_RS232.doc');
execute write_lob('1739_SDRAM.doc');
timing stop
```

Practice 8-1: Using SecureFile Format LOBs (continued)

```
SQL> set serveroutput on
SQL> set verify on
SQL> set term on
SQL> set lines 200
SQL>
SQL> timing start load_data
SQL> execute write_lob('1726_LCD.doc');
Begin inserting rows...
Row 1 inserted.
Row 2 inserted.
Row 3 inserted.
Row 4 inserted.
Row 5 inserted.

PL/SQL procedure successfully completed.

SQL> execute write_lob('1734_RS232.doc');
Begin inserting rows...
Row 1 inserted.
Row 2 inserted.
Row 3 inserted.
Row 4 inserted.
Row 5 inserted.

PL/SQL procedure successfully completed.

SQL> execute write_lob('1739_SDRAM.doc');
Begin inserting rows...
Row 1 inserted.
Row 2 inserted.
Row 3 inserted.
Row 4 inserted.
Row 5 inserted.

PL/SQL procedure successfully completed.

SQL> timing stop
timing for: load_data
Elapsed: 00:00:00.09
SQL>
```

- h) As the SYSTEM user, check the segment type in the data dictionary.

```
SELECT segment_name, segment_type, segment_subtype
FROM dba_segments
WHERE tablespace_name = 'LOB_TBS2'
AND segment_type = 'LOBSEGMENT';
```

```
connect /as sysdba

SELECT segment_name, segment_type, segment_subtype
FROM dba_segments
WHERE tablespace_name = 'LOB_TBS2'
AND segment_type = 'LOBSEGMENT';
```

SEGMENT_NAME	SEGMENT_TYPE	SEGMENT_SU
SYS_LOB0000071571C00002\$\$	LOBSEGMENT	ASSM

Practice 8-1: Using SecureFile Format LOBs (continued)

- i) As the OE user, create an interim table.

```
CREATE TABLE product_descriptions_interim
(product_id NUMBER,
 detailed_product_info BLOB)
LOB(detailed_product_info) STORE AS SECUREFILE
(TABLESPACE lob_tbs2);
```

```
connect oe

CREATE TABLE product_descriptions_interim
(product_id NUMBER,
 detailed_product_info BLOB)
LOB(detailed_product_info) STORE AS SECUREFILE
(TABLESPACE lob_tbs2);
```

- j) Connect as the SYSTEM user and run the redefinition script. (You can run the D:\labs\lab\lab_08_01_j.sql script.)

```
DECLARE
    error_count PLS_INTEGER := 0;
BEGIN
    DBMS_REDEFINITION.START_REDEF_TABLE
    ('OE', 'product_descriptions',
     'product_descriptions_interim',
     'product_id product_id, detailed_product_info
      detailed_product_info',
     OPTIONS_FLAG => DBMS_REDEFINITION.CONS_USE_ROWID);
    DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS
    ('OE', 'product_descriptions',
     'product_descriptions_interim',
     1, true,true,true,false, error_count);
    DBMS_OUTPUT.PUT_LINE('Errors := ' ||
                          TO_CHAR(error_count));
    DBMS_REDEFINITION.FINISH_REDEF_TABLE
    ('OE', 'product_descriptions',
     'product_descriptions_interim');
END;
/
```

```
connect /as sysdba
SET SERVEROUT ON

DECLARE
    error_count PLS_INTEGER := 0;
BEGIN
    DBMS_REDEFINITION.START_REDEF_TABLE
    ('OE', 'product_descriptions',
     'product_descriptions_interim',
     'product_id product_id, detailed_product_info
```

Practice 8-1: Using SecureFile Format LOBs (continued)

```
detailed_product_info',
OPTIONS_FLAG => DBMS_REDEFINITION.CONS_USE_ROWID);
DBMS_REDEFINITION.COPY_TABLE_DEPENDENTS
('OE', 'product_descriptions',
'product_descriptions_interim',
1, true,true,true,false, error_count);
DBMS_OUTPUT.PUT_LINE('Errors := ' ||
TO_CHAR(error_count));
DBMS_REDEFINITION.FINISH_REDEF_TABLE
('OE', 'product_descriptions',
'product_descriptions_interim');
END;
/
```

Errors := 0

- k) As the OE user, remove the interim table.

```
DROP TABLE product_descriptions_interim;
```

```
connect oe
```

```
DROP TABLE product_descriptions_interim;
```

- l) As the SYS user, check the segment type in the data dictionary to make sure it is now set to SECUREFILE.

```
SELECT segment_name, segment_type, segment_subtype
FROM dba_segments
WHERE tablespace_name = 'LOB_TBS2'
AND segment_type = 'LOBSEGMENT';
```

```
connect /as sysdba
```

```
SELECT segment_name, segment_type, segment_subtype
FROM dba_segments
WHERE tablespace_name = 'LOB_TBS2'
AND segment_type = 'LOBSEGMENT';
```

SEGMENT_NAME	SEGMENT_TYPE	SEGMENT_SU
SYS_LOB0000071577C00002\$\$	LOBSEGMENT	SECUREFILE
SYS_LOB0000071571C00002\$\$	LOBSEGMENT	ASSM

- m) As the OE user, check the space of the table by executing the CHECK_SPACE procedure. (You can run the D:\labs\labs\lab_08_01_m.sql script.)

```
CREATE OR REPLACE PROCEDURE check_space
```

Practice 8-1: Using SecureFile Format LOBs (continued)

```
IS
    l_fs1_bytes NUMBER;
    l_fs2_bytes NUMBER;
    l_fs3_bytes NUMBER;
    l_fs4_bytes NUMBER;
    l_fs1_blocks NUMBER;
    l_fs2_blocks NUMBER;
    l_fs3_blocks NUMBER;
    l_fs4_blocks NUMBER;
    l_full_bytes NUMBER;
    l_full_blocks NUMBER;
    l_unformatted_bytes NUMBER;
    l_unformatted_blocks NUMBER;
BEGIN
    DBMS_SPACE.SPACE_USAGE(
        segment_owner      => 'OE',
        segment_name       => 'PRODUCT_DESCRIPTIONS',
        segment_type       => 'TABLE',
        fs1_bytes          => l_fs1_bytes,
        fs1_blocks         => l_fs1_blocks,
        fs2_bytes          => l_fs2_bytes,
        fs2_blocks         => l_fs2_blocks,
        fs3_bytes          => l_fs3_bytes,
        fs3_blocks         => l_fs3_blocks,
        fs4_bytes          => l_fs4_bytes,
        fs4_blocks         => l_fs4_blocks,
        full_bytes         => l_full_bytes,
        full_blocks        => l_full_blocks,
        unformatted_blocks => l_unformatted_blocks,
        unformatted_bytes  => l_unformatted_bytes
    );
    DBMS_OUTPUT.ENABLE;
    DBMS_OUTPUT.PUT_LINE(' FS1 Blocks = '||l_fs1_blocks||'
        Bytes = '||l_fs1_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS2 Blocks = '||l_fs2_blocks||'
        Bytes = '||l_fs2_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS3 Blocks = '||l_fs3_blocks||'
        Bytes = '||l_fs3_bytes);
    DBMS_OUTPUT.PUT_LINE(' FS4 Blocks = '||l_fs4_blocks||'
        Bytes = '||l_fs4_bytes);
    DBMS_OUTPUT.PUT_LINE('Full Blocks = '||l_full_blocks||'
        Bytes = '||l_full_bytes);

    DBMS_OUTPUT.PUT_LINE('=====
    =====');
    DBMS_OUTPUT.PUT_LINE('Total Blocks =
        '||to_char(l_fs1_blocks + l_fs2_blocks +
        l_fs3_blocks + l_fs4_blocks + l_full_blocks)||' ||
        Total Bytes = '|| to_char(l_fs1_bytes + l_fs2_bytes
        + l_fs3_bytes + l_fs4_bytes + l_full_bytes));
END;
/
```

Practice 8-1: Using SecureFile Format LOBs (continued)

```
set serveroutput on
execute check_space;

FS1  Blocks = 0
     Bytes = 0
FS2  Blocks = 0
     Bytes = 0
FS3  Blocks = 0
     Bytes = 0
FS4  Blocks = 0
     Bytes = 0
Full  Blocks = 1
      Bytes = 8192
=====
=====
Total  Blocks =
1 !!
      Total Bytes = 8192
```

Practices for Lesson 9

Practice 9-1: Performance and Tuning

In this practice, you measure and examine performance and tuning.

Use your OE connection.

Writing Better Code

- 1) Open the lab_09_01.sql file and examine the package (the package body is as follows):

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no
        VARCHAR2);
    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/
CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2,
        p_card_no VARCHAR2)
    IS
        v_card_info typ_cr_card_nst;
        i INTEGER;
    BEGIN
        SELECT credit_cards
            INTO v_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF v_card_info.EXISTS(1) THEN -- cards exist, add more
            i := v_card_info.LAST;
            v_card_info.EXTEND(1);
            v_card_info(i+1) := typ_cr_card(p_card_type,
                                            p_card_no);
        UPDATE customers
            SET credit_cards = v_card_info
            WHERE customer_id = p_cust_id;
        ELSE -- no cards for this customer yet, construct one
            UPDATE customers
                SET credit_cards = typ_cr_card_nst
                    (typ_cr_card(p_card_type, p_card_no))
                WHERE customer_id = p_cust_id;
        END IF;
```

Practice 9-1: Performance and Tuning (continued)

```
END update_card_info;
-- continued on next page.
```

```
PROCEDURE display_card_info
  (p_cust_id NUMBER)
IS
  v_card_info typ_cr_card_nst;
  i INTEGER;
BEGIN
  SELECT credit_cards
    INTO v_card_info
    FROM customers
   WHERE customer_id = p_cust_id;
  IF v_card_info.EXISTS(1) THEN
    FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
      DBMS_OUTPUT.PUT('Card Type: ' ||
                      v_card_info(idx).card_type || ' ');
      DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
                      v_card_info(idx).card_num );
    END LOOP;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Customer has no credit
                          cards.');
  END IF;
  END display_card_info;
END credit_card_pkg; -- package body
/
```

However, this code needs to be improved. The following issues exist in the code:

- The local variables use the INTEGER data type.
- The same SELECT statement is run in the two procedures.
- The same IF v_card_info.EXISTS(1) THEN statement is in the two procedures.

Using Efficient Data Types

- 2) To improve the code, make the following modifications:
 - a) Change the local INTEGER variables to use a more efficient data type.
 - b) Move the duplicated code into a function. The package specification for the modification is:

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS

  FUNCTION cust_card_info
    (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
    RETURN BOOLEAN;
```

Practice 9-1: Performance and Tuning (continued)

```
PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2,
     p_card_no VARCHAR2);
PROCEDURE display_card_info
    (p_cust_id NUMBER);

END credit_card_pkg; -- package spec
/
```

- c) Have the function return TRUE if the customer has credit cards. The function should return FALSE if the customer does not have credit cards. Pass into the function an uninitialized nested table. The function places the credit card information into this uninitialized parameter.

```
-- note: If you did not complete lesson 4 practice,
--       you will need to run solution files sol_04_04_a,
--       sol_04_4_b, and sol_04_04_c
--       in order to have the supporting structures in place.

CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    FUNCTION cust_card_info
        (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
        RETURN BOOLEAN;
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2,
         p_card_no VARCHAR2);
    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS

    FUNCTION cust_card_info
        (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
        RETURN BOOLEAN
    IS
        v_card_info_exists BOOLEAN;
    BEGIN
        SELECT credit_cards
          INTO p_card_info
          FROM customers
         WHERE customer_id = p_cust_id;
        IF p_card_info.EXISTS(1) THEN
            v_card_info_exists := TRUE;
        ELSE
            v_card_info_exists := FALSE;
        END IF;
        RETURN v_card_info_exists;
    END cust_card_info;
```

Practice 9-1: Performance and Tuning (continued)

```
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 PLS_INTEGER;
BEGIN
  IF cust_card_info(p_cust_id, v_card_info) THEN
-- cards exist, add more
    i := v_card_info.LAST;
    v_card_info.EXTEND(1);
    v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
    UPDATE customers
      SET credit_cards = v_card_info
      WHERE customer_id = p_cust_id;
  ELSE -- no cards for this customer yet, construct one
    UPDATE customers
      SET credit_cards = typ_cr_card_nst
        (typ_cr_card(p_card_type, p_card_no))
      WHERE customer_id = p_cust_id;
  END IF;
END update_card_info;

PROCEDURE display_card_info
  (p_cust_id NUMBER)
IS
  v_card_info typ_cr_card_nst;
  i PLS_INTEGER;
BEGIN
  IF cust_card_info(p_cust_id, v_card_info) THEN
    FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
      DBMS_OUTPUT.PUT('Card Type: ' ||
                      v_card_info(idx).card_type || ' ');
      DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
                      v_card_info(idx).card_num );
    END LOOP;
  ELSE
    DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
  END IF;
END display_card_info;
END credit_card_pkg; -- package body
/
```

- 3) Test your modified code with the following data:

```
EXECUTE credit_card_pkg.update_card_info -
  (120, 'AM EX', 5555555555)
PL/SQL procedure successfully completed.
```

```
EXECUTE credit_card_pkg.display_card_info(120)
Card Type: Visa / Card No: 11111111
Card Type: MC / Card No: 2323232323
```

Practice 9-1: Performance and Tuning (continued)

Card Type: DC / Card No: 4444444
Card Type: AM EX / Card No: 55555555555

PL/SQL procedure successfully completed.

-- Note: If you did not complete Practice 4, your results
-- will be:

EXECUTE credit_card_pkg.display_card_info(120)
Card Type: AM EX / Card No: 55555555555

PL/SQL procedure successfully completed.

- 4) You need to modify the UPDATE_CARD_INFO procedure to return information (using the RETURNING clause) about the credit cards being updated. Assume that this information will be used by another application developer on your team, who is writing a graphical reporting utility on customer credit cards after a customer's credit card information is changed.
 - a) Open the lab_09_04_a.sql file. It contains the modified code from the previous question #2.
 - b) Modify the code to use the RETURNING clause to find information about the rows that are affected by the UPDATE statements.

```
CREATE OR REPLACE PACKAGE credit_card_pkg
IS
    FUNCTION cust_card_info
        (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
        RETURN BOOLEAN;
    PROCEDURE update_card_info
        (p_cust_id NUMBER, p_card_type VARCHAR2,
         p_card_no VARCHAR2, o_card_info OUT typ_cr_card_nst);
    PROCEDURE display_card_info
        (p_cust_id NUMBER);
END credit_card_pkg; -- package spec
/

CREATE OR REPLACE PACKAGE BODY credit_card_pkg
IS
    FUNCTION cust_card_info
        (p_cust_id NUMBER, p_card_info IN OUT typ_cr_card_nst )
        RETURN BOOLEAN
    IS
        v_card_info_exists BOOLEAN;
    BEGIN
        SELECT credit_cards
            INTO p_card_info
            FROM customers
            WHERE customer_id = p_cust_id;
        IF p_card_info.EXISTS(1) THEN
            v_card_info_exists := TRUE;
```

Practice 9-1: Performance and Tuning (continued)

```
        ELSE
            v_card_info_exists := FALSE;
        END IF;
        RETURN v_card_info_exists;
    END cust_card_info;

PROCEDURE update_card_info
    (p_cust_id NUMBER, p_card_type VARCHAR2,
     p_card_no VARCHAR2, o_card_info OUT typ_cr_card_nst)
IS
    v_card_info typ_cr_card_nst;
    i PLS_INTEGER;
BEGIN

    IF cust_card_info(p_cust_id, v_card_info) THEN
        -- cards exist, add more
        i := v_card_info.LAST;
        v_card_info.EXTEND(1);
        v_card_info(i+1) := typ_cr_card(p_card_type, p_card_no);
        UPDATE customers
            SET credit_cards = v_card_info
            WHERE customer_id = p_cust_id
        RETURNING credit_cards INTO o_card_info;
    ELSE
        -- no cards for this customer yet, construct one
        UPDATE customers
            SET credit_cards = typ_cr_card_nst
                (typ_cr_card(p_card_type, p_card_no))
            WHERE customer_id = p_cust_id
        RETURNING credit_cards INTO o_card_info;
    END IF;
END update_card_info;

PROCEDURE display_card_info
    (p_cust_id NUMBER)
IS
    v_card_info typ_cr_card_nst;
    i PLS_INTEGER;
BEGIN
    IF cust_card_info(p_cust_id, v_card_info) THEN
        FOR idx IN v_card_info.FIRST..v_card_info.LAST LOOP
            DBMS_OUTPUT.PUT('Card Type: ' ||
                v_card_info(idx).card_type || ' ');
            DBMS_OUTPUT.PUT_LINE('/ Card No: ' ||
                v_card_info(idx).card_num );
        END LOOP;
    ELSE
        DBMS_OUTPUT.PUT_LINE('Customer has no credit cards.');
    END IF;
END display_card_info;
END credit_card_pkg; -- package body
/
```

Practice 9-1: Performance and Tuning (continued)

- c) You can test your modified code with the following procedure (contained in lab_09_04_c.sql):

```
CREATE OR REPLACE PROCEDURE test_credit_update_info
(p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no NUMBER)
IS
    v_card_info typ_cr_card_nst;
BEGIN
    credit_card_pkg.update_card_info
        (p_cust_id, p_card_type, p_card_no, v_card_info);
END test_credit_update_info;
/
-- execute this code:
CREATE OR REPLACE PROCEDURE test_credit_update_info
(p_cust_id NUMBER, p_card_type VARCHAR2, p_card_no NUMBER)
IS
    v_card_info typ_cr_card_nst;
BEGIN
    credit_card_pkg.update_card_info
        (p_cust_id, p_card_type, p_card_no, v_card_info);
END test_credit_update_info;
/
```

Procedure created.

- d) Test your code with the following statements set in boldface:

```
EXECUTE test_credit_update_info(125, 'AM EX', 123456789)
PL/SQL procedure successfully completed.
```

```
SELECT credit_cards FROM customers WHERE customer_id = 125;
```

```
CREDIT_CARDS<CARD_TYPE, CARD_NUM>
```

```
TYP_CR_CARD_NST<TYP_CR_CARD<'AM EX', 123456789>>
```

Collecting Exception Information

- 5) In this exercise, you test exception handling with the SAVE EXCEPTIONS clause.

- a) Run the lab_09_05_a.sql file to create a test table:

```
CREATE TABLE card_table
(accepted_cards VARCHAR2(50) NOT NULL);
```

- b) Open the lab_09_05_b.sql file and run the contents:

```
DECLARE
    type typ_cards is table of VARCHAR2(50);
    v_cards typ_cards := typ_cards
    ('Citigroup Visa', 'Nationscard MasterCard',
     'Federal American Express', 'Citizens Visa',
     'International Discoverer', 'United Diners Club');
```

Practice 9-1: Performance and Tuning (continued)

```
BEGIN
    v_cards.Delete(3);
    v_cards.DELETE(6);
    FORALL j IN v_cards.first..v_cards.last
        SAVE EXCEPTIONS
        EXECUTE IMMEDIATE
        'insert into card_table (accepted_cards) values
        (:the_card)'
        USING v_cards(j);
/
END;
/
```

- c) Note the output:

```
ERROR:
ORA-24381: error(s) in array DML
ORA-06512: at line 10
```

Note the output: This returns an “Error in Array DML (at line 11),” which is not very informative. The cause of this error: One or more rows failed in the DML.

- d) Open the lab_09_05_d.sql file and run the contents:

```
DECLARE
    type typ_cards is table of VARCHAR2(50);
    v_cards typ_cards := typ_cards
    ( 'Citigroup Visa', 'Nationscard MasterCard',
      'Federal American Express', 'Citizens Visa',
      'International Discoverer', 'United Diners Club' );
    bulk_errors EXCEPTION;
    PRAGMA exception_init (bulk_errors, -24381 );
BEGIN
    v_cards.Delete(3);
    v_cards.DELETE(6);
    FORALL j IN v_cards.first..v_cards.last
        SAVE EXCEPTIONS
        EXECUTE IMMEDIATE
        'insert into card_table (accepted_cards) values (
        :the_card)'
        USING v_cards(j);
EXCEPTION
    WHEN bulk_errors THEN
        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;
    /

```

Practice 9-1: Performance and Tuning (continued)

- e) Note the output:

```
3:      ORA-22160: element at index [] does not exist
```

- f) Why is the output different?

The PL/SQL block raises the exception 22160 when it encounters an array element that was deleted. The exception is handled and the block is completed successfully.

Timing Performance of SIMPLE_INTEGER and PLS_INTEGER

- 6) In this exercise, you compare the performance between the PLS_INTEGER and SIMPLE_INTEGER data types with native compilation:

- a) Run the lab_09_06_a.sql file to create a testing procedure that contains conditional compilation:

```
CREATE OR REPLACE PROCEDURE p
IS
    t0      NUMBER :=0;
    t1      NUMBER :=0;

    $IF $$Simple $THEN
        SUBTYPE My_Integer_t IS
            SIMPLE_INTEGER;
        My_Integer_t_Name CONSTANT VARCHAR2(30) :=
            'SIMPLE_INTEGER';
    $ELSE
        SUBTYPE My_Integer_t IS
            PLS_INTEGER;
        My_Integer_t_Name CONSTANT VARCHAR2(30) :=
            'PLS_INTEGER';
    $END

    v00  My_Integer_t := 0;      v01  My_Integer_t := 0;
    v02  My_Integer_t := 0;      v03  My_Integer_t := 0;
    v04  My_Integer_t := 0;      v05  My_Integer_t := 0;

    two   CONSTANT My_Integer_t := 2;
    lmt   CONSTANT My_Integer_t := 100000000;

BEGIN
    t0 := DBMS_UTILITY.GET_CPU_TIME();
    WHILE v01 < lmt LOOP
        v00 := v00 + Two;
        v01 := v01 + Two;
        v02 := v02 + Two;
        v03 := v03 + Two;
        v04 := v04 + Two;
        v05 := v05 + Two;
    END LOOP;
```

Practice 9-1: Performance and Tuning (continued)

```
IF v01 <> lmt OR v01 IS NULL THEN
    RAISE Program_Error;
END IF;

t1 := DBMS_UTILITY.GET_CPU_TIME();
DBMS_OUTPUT.PUT_LINE(
    RPAD(LOWER($$PLSQL_Code_Type), 15) ||
    RPAD(LOWER(My_Integer_t_Name), 15) ||
    TO_CHAR((t1-t0), '9999') || ' centiseconds');
END p;
/
```

- b) Open the lab_09_06_b.sql file and run the contents:

```
ALTER PROCEDURE p COMPILE
PLSQL_Code_Type = NATIVE PLSQL_CCFlags = 'simple:true'
REUSE SETTINGS;

EXECUTE p()

ALTER PROCEDURE p COMPILE
PLSQL_Code_Type = native PLSQL_CCFlags = 'simple:false'
REUSE SETTINGS;

EXECUTE p()
```

- c) Note the output:

First run:

native simple_integer 25 centiseconds

Second run:

native pls_integer 537 centiseconds

- d) Explain the output.

The SIMPLE_INTEGER runs much faster in this scenario. If you can use the SIMPLE_INTEGER data type, it can improve performance.

Practices for Lesson 10

Practice 10-1: Improving Performance with Caching

In this practice, you examine the Explain Plan for a query, add the RESULT_CACHE hint to the query, and reexamine the Explain Plan results. You also execute some code and modify the code so that PL/SQL result caching is turned on.

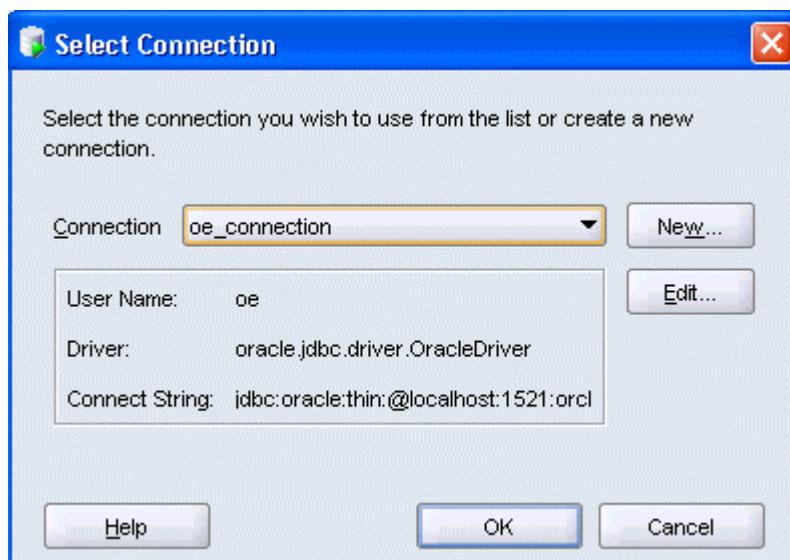
Examining SQL and PL/SQL Result Caching

- 1) Use **Oracle SQL Developer** to connect to the OE schema. Examine the Explain Plan for this query found in the `lab_10_01.sql` file. You can see the Explain Plan by clicking the Execute Explain Plan button on the toolbar in the Code Editor window.

```
SELECT count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
  FROM inventories natural join product_information
 GROUP BY product_id, product_name;
```

Examine the Explain Plan results.

In Oracle SQL Developer, choose the `oe_connection` connection. Select Tools > SQL Worksheet, or click the Open SQL Worksheet icon. The Select Connection window appears. Select `oe_connection` and click OK.

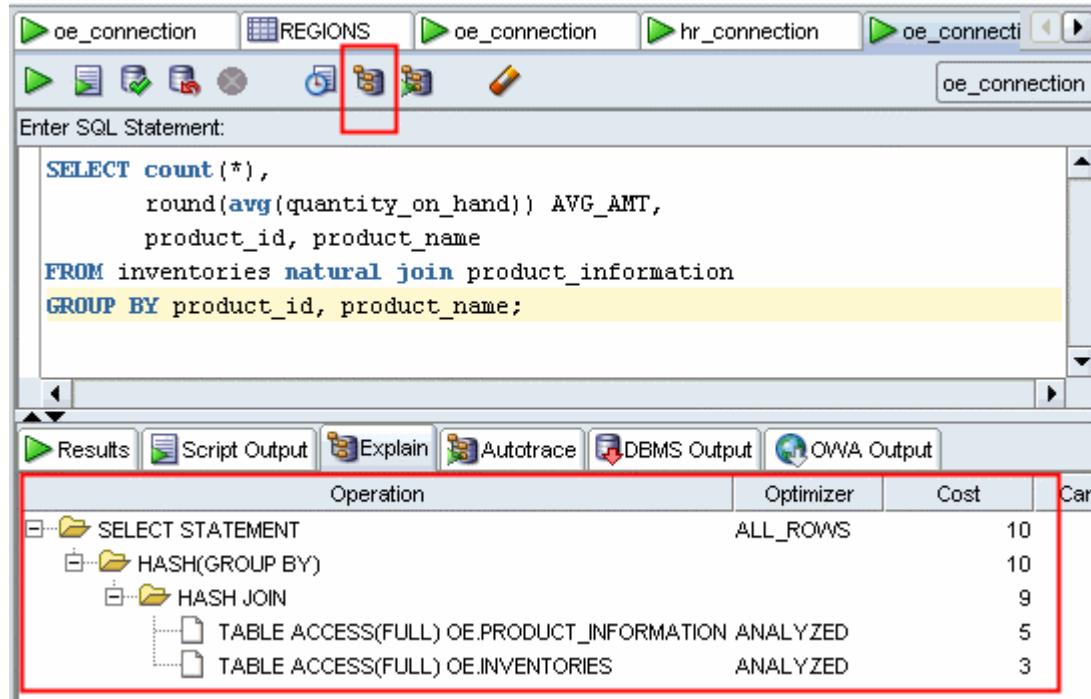


In the SQL Worksheet, paste the code found in the `lab_10_01.sql` file:

```
SELECT count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
  FROM inventories natural join product_information
 GROUP BY product_id, product_name;
```

Practice 10-1: Improving Performance with Caching (continued)

Click the Execute Explain Plan button on the toolbar and observe the results in the lower region:



Results: SQL caching is not enabled and not visible in the Explain Plan.

- 2) Add the RESULT_CACHE hint to the query and reexamine the Explain Plan results.

```
SELECT /*+ result_cache */
       count(*),
       round(avg(quantity_on_hand)) AVG_AMT,
       product_id, product_name
  FROM inventories natural join product_information
 GROUP BY product_id, product_name;
```

Examine the Explain Plan results, compared to the previous Results.

Add the following code immediately after the SELECT command:

```
/*+ result_cache */
```

Click the Execute Explain Plan button on the toolbar again, and compare the results in the lower region with the previous results:

Practice 10-1: Improving Performance with Caching (continued)

The screenshot shows the Oracle SQL Developer interface. In the top navigation bar, there are several connections listed: oe_connection, REGIONS, oe_connection, hr_connection, and oe_connection. Below the bar is a toolbar with various icons. The main area is titled "Enter SQL Statement:" and contains the following SQL code:

```
SELECT /*+ result_cache */  
    count(*),  
    round(avg(quantity_on_hand)) AVG_AMT,  
    product_id, product_name  
FROM inventories natural join product_information  
GROUP BY product_id, product_name;
```

Below the SQL editor is the "Explain" tab of the results panel. The explain plan tree is shown, with the "RESULT CACHE" node highlighted with a red box. The tree structure is as follows:

- SELECT STATEMENT (ALL_ROWS, Cost=10)
 - RESULT CACHE 21utc85su7frh64b9p5az5aw1c (10)
 - HASH(GROUP BY)
 - HASH JOIN
 - TABLE ACCESS(FULL) OE.PRODUCT_INFORMATION ANALYZED (5)
 - TABLE ACCESS(FULL) OE.INVENTORIES ANALYZED (3)

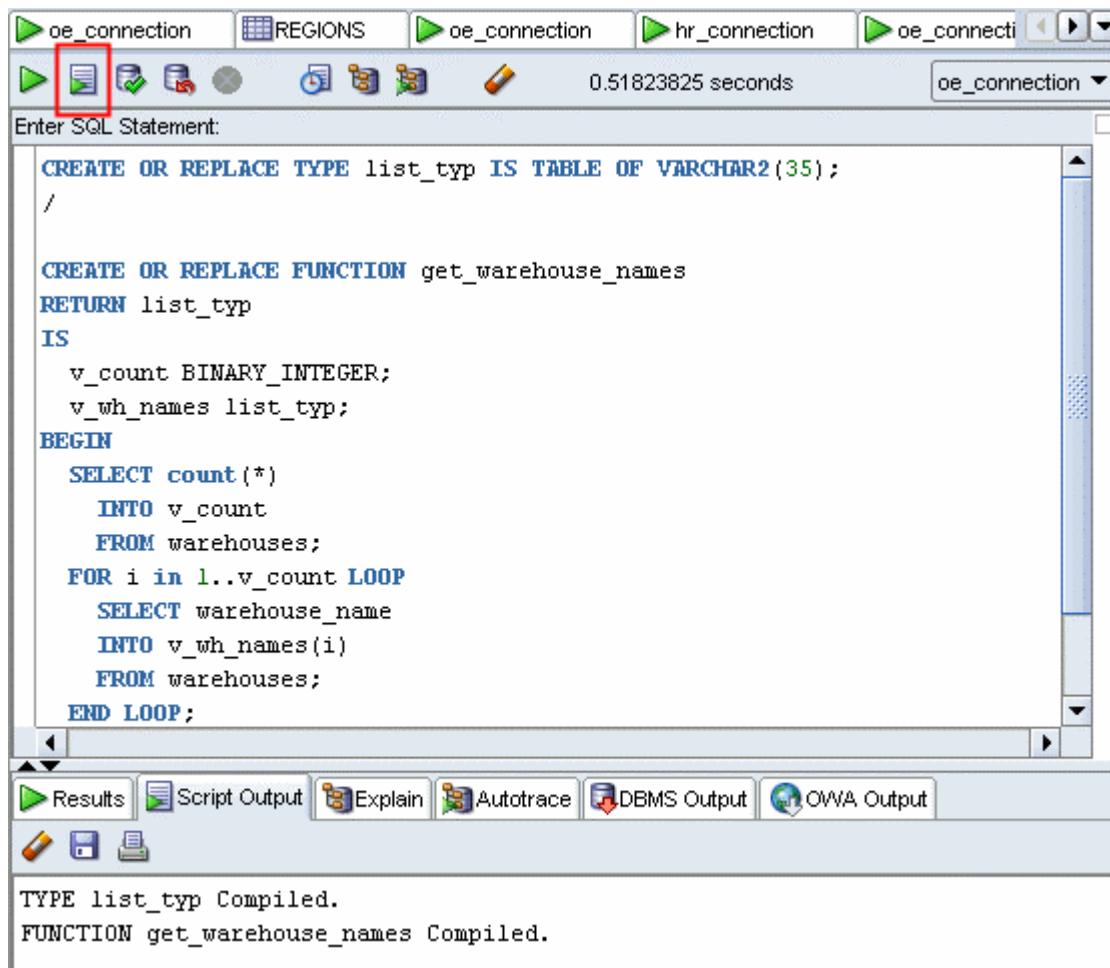
Results: Note that result caching is used in the Explain Plan.

- 3) The following code is used to generate a list of warehouse names for pick lists in applications. The WAREHOUSES table is fairly stable and is not modified often.

Click the Run Script button to compile this code (You can use the lab_10_03.sql file.):

```
CREATE OR REPLACE TYPE list_typ IS TABLE OF VARCHAR2(35);  
/  
  
CREATE OR REPLACE FUNCTION get_warehouse_names  
RETURN list_typ  
IS  
    v_count BINARY_INTEGER;  
    v_wh_names list_typ;  
BEGIN  
    SELECT count(*)  
        INTO v_count  
        FROM warehouses;  
    FOR i in 1..v_count LOOP  
        SELECT warehouse_name  
        INTO v_wh_names(i)  
        FROM warehouses;  
    END LOOP;  
    RETURN v_wh_names;  
END get_warehouse_names;
```

Practice 10-1: Improving Performance with Caching (continued)



The screenshot shows the Oracle SQL Developer interface. The top toolbar has several connection icons and a status bar showing "0.51823825 seconds". Below the toolbar is a menu bar with "File", "Edit", "Tools", "Database", "Help", and "About". The main area is titled "Enter SQL Statement:" and contains the following PL/SQL code:

```
CREATE OR REPLACE TYPE list_typ IS TABLE OF VARCHAR2(35);
/
CREATE OR REPLACE FUNCTION get_warehouse_names
RETURN list_typ
IS
  v_count BINARY_INTEGER;
  v_wh_names list_typ;
BEGIN
  SELECT count(*)
  INTO v_count
  FROM warehouses;
  FOR i in 1..v_count LOOP
    SELECT warehouse_name
    INTO v_wh_names(i)
    FROM warehouses;
  END LOOP;
END;
```

Below the code, the results window displays:

```
TYPE list_typ Compiled.
FUNCTION get_warehouse_names Compiled.
```

Paste the `lab_10_03.sql` script into the “Enter SQL Statement” box. Click the Run Script button. You have compiled the function without PL/SQL result caching.

- 4) Because the function is called frequently, and because the content of the data returned does not frequently change, this code is a good candidate for PL/SQL result caching.

Modify the code so that PL/SQL result caching is turned on. Click the Run Script button to compile this code again.

Insert the following line after `RETURN list_typ`:

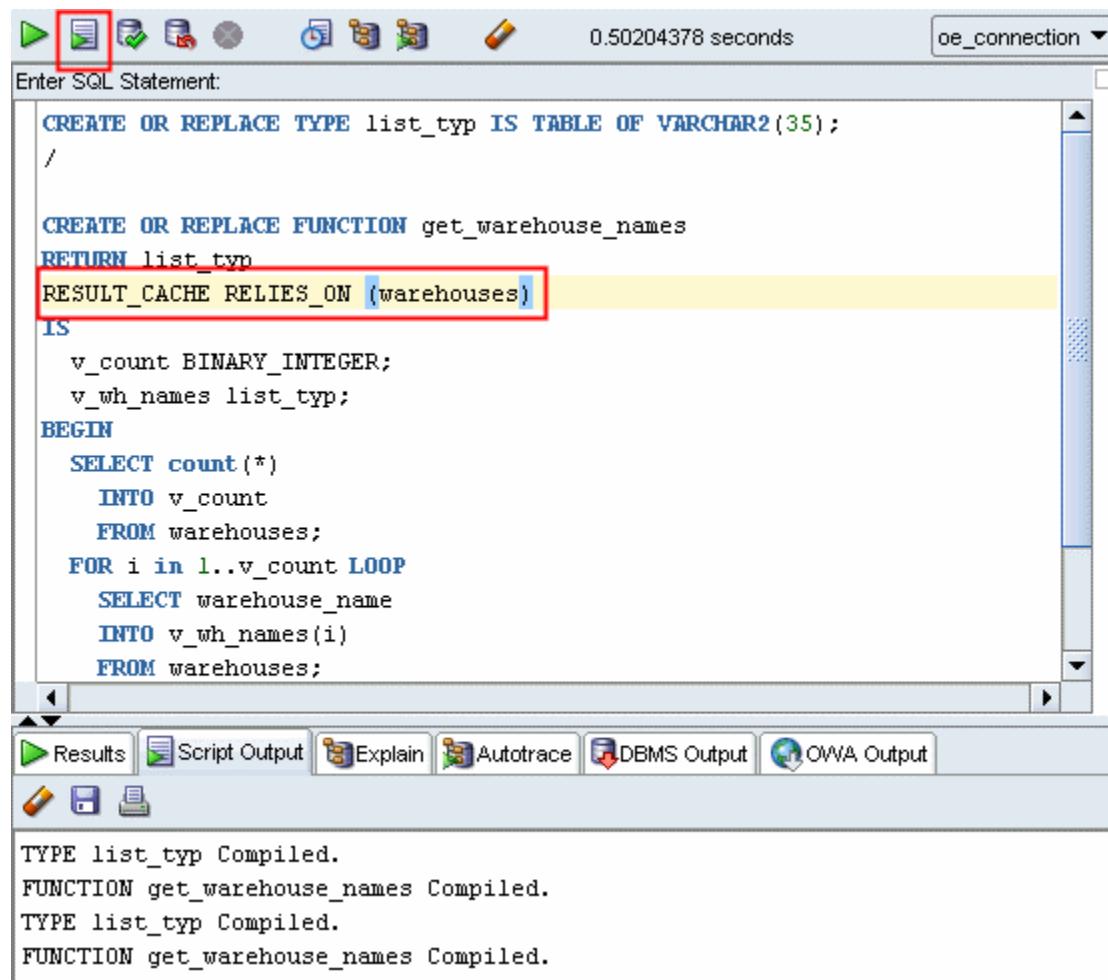
```
RESULT_CACHE RELIES_ON (warehouses)
```

```
CREATE OR REPLACE FUNCTION get_warehouse_names
RETURN list_typ
RESULT_CACHE RELIES_ON (warehouses)
IS
```

Practice 10-1: Improving Performance with Caching (continued)

```
v_count BINARY_INTEGER;
v_wh_names list_typ;
BEGIN
  SELECT count(*)
    INTO v_count
   FROM warehouses;
FOR i in 1..v_count LOOP
  SELECT warehouse_name
    INTO v_wh_names(i)
   FROM warehouses;
END LOOP;
RETURN v_wh_names;
END get_warehouse_names;
```

Click the Run Script button to recompile the code.



Practices for Lesson 11

Practice 11-1: Analyzing Your Code

In this practice, you use PL/SQL and **Oracle SQL Developer** to analyze your code.

Use your OE connection.

Finding Coding Information

- 1) Create the QUERY_CODE_PKG package to search your source code.

- a) Run the lab_11_01_a.sql script to create the QUERY_CODE_PKG package.

```
SQL> @D:\Labs\labs\lab_11_01_a.sql
```

- b) Run the ENCAP_COMPLIANCE procedure to see which of your programs reference tables or views. (**Note:** Your results might differ slightly.)

```
SET SERVEROUT ON
EXECUTE query_code_pkg.encap_compliance
```

```
|Programs that reference tables or views
OE.CREDIT_CARD_PKG,OE.CUSTOMERS
OE.GET_INCOME_LEVEL,OE.CUSTOMERS
OE.LOAD_PRODUCT_IMAGE,OE.PRODUCT_INFORMATION
OE.WRITE_LOB,OE.PRODUCT_DESCRIPTIONS
```

- c) Run the FIND_TEXT_IN_CODE procedure to find all references to 'ORDERS.' (Note: Your results might differ slightly.)

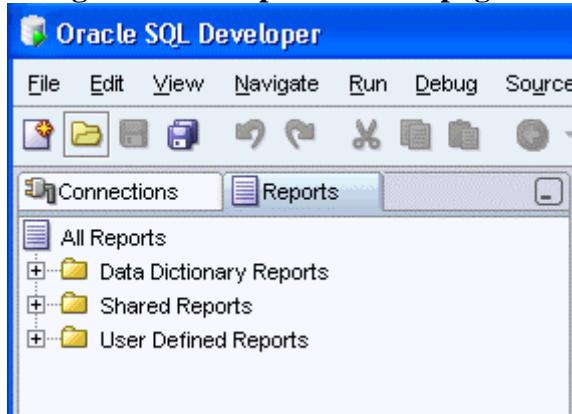
```
SET SERVEROUT ON
EXECUTE query_code_pkg.find_text_in_code('ORDERS')
```

```
|Checking for presence of ORDERS:
CUSTOMER_TYP-12,      , cust_orders      order_list_typ
ORDERS_ITEMS_TRG-1,TRIGGER orders_items_trg INSTEAD OF INSERT ON NESTED
ORDERS_ITEMS_TRG-2, TABLE order_item_list OF oc_orders FOR EACH ROW
ORDERS_TRG-1,TRIGGER orders_trg INSTEAD OF INSERT
ORDERS_TRG-2, ON oc_orders FOR EACH ROW
ORDERS_TRG-4,   INSERT INTO ORDERS (order_id, order_mode, order_total,
```

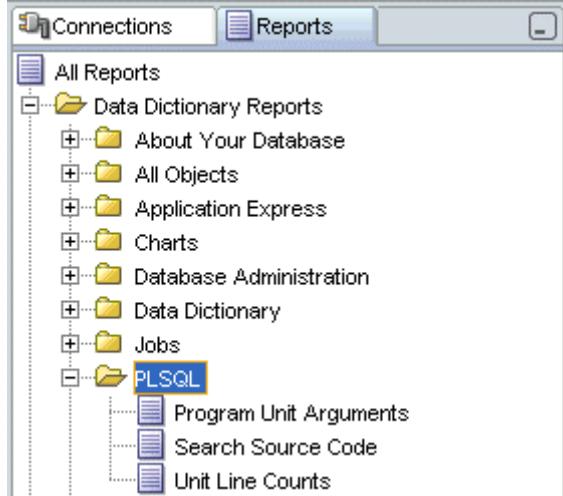
Practice 11-1: Analyzing Your Code (continued)

- d) Use the **Oracle SQL Developer** Reports feature to find the same results for step c shown above.

Navigate to the Reports tabbed page in Oracle SQL Developer.



Expand the Data Dictionary Reports node and expand the PL/SQL node.



Select Search Source Code and then select your OE connection.

Practice 11-1: Analyzing Your Code (continued)

Select Text search and enter ORDERS for the Value: field. Click the Apply button.



The screenshot shows the PL/Scope interface with a toolbar at the top featuring icons for search, refresh, and OE. The main area displays a table of PL/SQL objects:

Owner	PL/SQL Object Name	Type	Line	Text
OE	CUSTOMER_TYP	TYPE	12	, cust_orders order_list_typ
OE	ORDERS_ITEMS_TRG	TRIGGER	1	TRIGGER orders_items_trg INSTEAD OF INSERT
OE	ORDERS_ITEMS_TRG	TRIGGER	2	TABLE order_item_list OF oc_orders FOR EACH
OE	ORDERS_TRG	TRIGGER	1	TRIGGER orders_trg INSTEAD OF INSERT
OE	ORDERS_TRG	TRIGGER	2	ON oc_orders FOR EACH ROW
OE	ORDERS_TRG	TRIGGER	4	INSERT INTO ORDERS (order_id, order_mode,

Using PL/Scope

2) In this practice, you use PL/Scope.

a) Enable your session to collect identifiers.

```
ALTER SESSION SET PLSCOPE_SETTINGS = 'IDENTIFIERS:ALL';
```

b) Recompile your CREDIT_CARD_PKG code.

```
ALTER PACKAGE credit_card_pkg COMPILE;
```

c) Verify that your PLSCOPE_SETTING is set correctly by issuing the following statement:

```
SELECT PLSCOPE_SETTINGS
FROM USER_PLSQL_OBJECT_SETTINGS
WHERE NAME='CREDIT_CARD_PKG' AND TYPE='PACKAGE BODY';
```

Practice 11-1: Analyzing Your Code (continued)

PLSCOPE_SETTINGS

IDENTIFIERS:ALL

- d) Execute the following statement to create a hierarchical report on the identifier information on the CREDIT_CARD_PKG code. You can run the lab_11_02_d.sql script file.

```
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;
```

IDENTIFIER_USAGE_CONTEXTS

Credit_Card_Pkg..... package	definition
Cust_Card_Info.... function	definition
P_Cust_Id..... formal in	declaration
P_Card_Info..... formal in out	declaration
U_Card_Info_Exis variable	declaration
P_Card_Info..... formal in out	assignment
P_Cust_Id..... formal in	reference
Pltblm..... synonym	call
P_Card_Info... formal in out	reference
U_Card_Info_Ex variable	assignment
U_Card_Info_Exis variable	assignment

IDENTIFIER_USAGE_CONTEXTS

U_Card_Info_Exis variable	reference
Update_Card_Info.. procedure	definition
P_Cust_Id..... formal in	declaration
P_Card_Type..... formal in	declaration
P_Card_No..... formal in	declaration
O_Card_Info..... formal out	declaration
U_Card_Info..... variable	declaration
I..... variable	declaration
Cust_Card_Info.. function	call
P_Cust_Id.... formal in	reference
U_Card_Info... variable	reference

Practice 11-1: Analyzing Your Code (continued)

IDENTIFIER_USAGE_CONTEXTS

I.....	variable	assignment
'Plitblm.....	synonym	call
'U_Card_Inf	variable	reference
Plitblm.....	synonym	call
'U_Card_Info.	variable	reference
'U_Card_Info...	variable	assignment
I.....	variable	reference
P_Card_Type.	formal in	reference
P_Card_No...	formal in	reference
U_Card_Info...	variable	reference
P_Cust_Id.....	formal in	reference

IDENTIFIER_USAGE_CONTEXTS

O_Card_Info...	formal out	assignment
P_Card_Type.....	formal in	reference
P_Card_No.....	formal in	reference
P_Cust_Id.....	formal in	reference
O_Card_Info.....	formal out	assignment
Display_Card_Info.	procedure	definition
P_Cust_Id.....	formal in	declaration
U_Card_Info.....	variable	declaration
I.....	variable	declaration
Cust_Card_Info..	function	call
P_Cust_Id.....	formal in	reference

IDENTIFIER_USAGE_CONTEXTS

U_Card_Info....	variable	reference
Idx.....	iterator	declaration
'Plitblm.....	synonym	call
'U_Card_Inf	variable	reference
Plitblm.....	synonym	call
'U_Card_Inf	variable	reference
Dbms_Output.	synonym	reference
'U_Card_Inf	variable	reference
Idx.....	iterator	reference
Dbms_Output.	synonym	reference
'U_Card_Inf	variable	reference

IDENTIFIER_USAGE_CONTEXTS

Idx.....	iterator	reference
Dbms_Output.....	synonym	reference

57 rows selected.

Practice 11-1: Analyzing Your Code (continued)

Using DBMS_METADATA

- 3) Use DBMS_METADATA to find the metadata for the ORDER_ITEMS table.

- a) Create the GET_TABLE_MD function. You can run the lab_11_03_a.sql script.

```
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', 'ORDER_ITEMS');
  -- 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;
/
```

- b) Issue the following statements to view the metadata generated from the GET_TABLE_MD function:

```
set pagesize 0
set long 1000000
```

```
SELECT get_table_md FROM dual;
```

```
CREATE TABLE "OE"."ORDER_ITEMS"
  (
    "ORDER_ID" NUMBER<12,0>,
    "LINE_ITEM_ID" NUMBER<3,0> NOT NULL ENABLE,
    "PRODUCT_ID" NUMBER<6,0> NOT NULL ENABLE,
    "UNIT_PRICE" NUMBER<8,2>,
    "QUANTITY" NUMBER<8,0>
  )
  CONSTRAINT "ORDER_ITEMS_PK" PRIMARY KEY <"ORDER_ID", "LINE_ITEM_ID">
  USING INDEX PCTFREE 10 INITTRANS 2 MAXTRANS 255 NOLOGGING COMPUTE STATISTICS
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "EXAMPLE" ENABLE,
  CONSTRAINT "ORDER_ITEMS_PRODUCT_ID_FK" FOREIGN KEY <"PRODUCT_ID">
  REFERENCES "OE"."PRODUCT_INFORMATION" <"PRODUCT_ID"> ENABLE,
  CONSTRAINT "ORDER_ITEMS_ORDER_ID_FK" FOREIGN KEY <"ORDER_ID">
  REFERENCES "OE"."ORDERS" <"ORDER_ID"> ON DELETE CASCADE ENABLE NOVALID
ATE
  > PCTFREE 10 PCTUSED 40 INITTRANS 1 MAXTRANS 255 NOCOMPRESS NOLOGGING
  STORAGE(INITIAL 65536 NEXT 1048576 MINEXTENTS 1 MAXEXTENTS 2147483645
  PCTINCREASE 0 FREELISTS 1 FREELIST GROUPS 1 BUFFER_POOL DEFAULT)
  TABLESPACE "EXAMPLE"
```

Practice 11-1: Analyzing Your Code (continued)

- c) Generate an XML representation of the ORDER_ITEMS table by using the DBMS_METADATA.GET_XML function. Spool the output to a file named ORDER_ITEMS_XML.txt in the D:\labs folder.

```
SPOOL D:\labs\ORDER_ITEMS_XML.txt

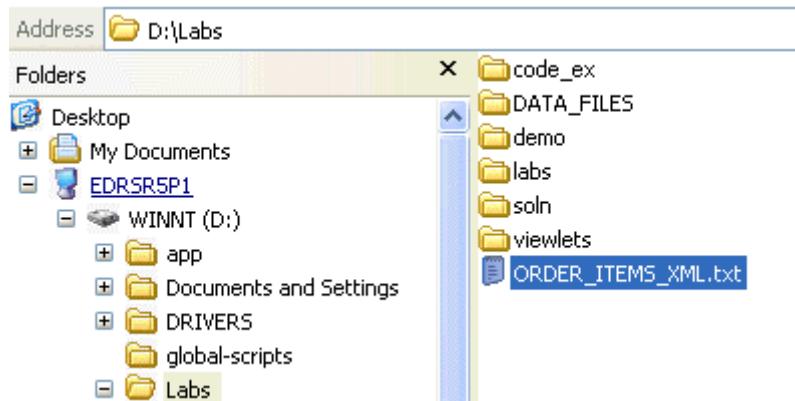
SELECT DBMS_METADATA.GET_XML
      ('TABLE', 'ORDER_ITEMS', 'OE')
FROM   dual;

SPOOL OFF
```

Practice 11-1: Analyzing Your Code (continued)

- d) Verify that the ORDER_ITEMS_XML.txt file in the D:\labs folder is created.

In Windows Explorer, check that the file is created:



Open the file to verify its contents:

```
SQL>
SQL> SELECT DBMS_METADATA.GET_XML
2      ('TABLE', 'ORDER_ITEMS', 'OE')
3     FROM dual;
<?xml version="1.0"?><ROWSET><ROW>
<TABLE_T>
<VERS_MAJOR>1</VERS_MAJOR>
<VERS_MINOR>2 </VERS_MINOR>
<OBJ_NUM>70328</OBJ_NUM>
<SCHEMA_OBJ>
<OBJ_NUM>70328</OBJ_NUM>
<DATAOBJ_NUM>69702</DATAOBJ_NUM>
<OWNER_NUM>83</OWNER_NUM>
<OWNER_NAME>OE</OWNER_NAME>
<NAME>ORDER_ITEMS</NAME>
<NAMESPACE>1</NAMESPACE>
<TYPE_NUM>2</TYPE_NUM>
<TYPE_NAME>TABLE</TYPE_NAME>
<CTIME>2008-01-31 18:59:41</CTIME>
<MTIME>2008-01-31 19:00:40</MTIME>
<STIME>2008-01-31 18:59:41</STIME>
<STATUS>1</STATUS>
<FLAGS>0</FLAGS>
<SPARE1>6</SPARE1>
<SPARE2>1</SPARE2>
<SPARE3>83</SPARE3>
</SCHEMA_OBJ>
<STORAGE>
<FILE_NUM>5</FILE_NUM>
<BLOCK_NUM>323</BLOCK_NUM>
<TYPE_NUM>5</TYPE_NUM>
<TS_NUM>6</TS_NUM>
<BLOCKS>8</BLOCKS>
<EXTENTS>1</EXTENTS>
<INIEXTS>8</INIEXTS>
<MINEXTS>1</MINEXTS>
```

Practices for Lesson 12

Practice 12-1: Profiling and Tracing

Profiling Data

In this practice, you generate profiler data and analyze it.

Use your OE connection.

- 1) Generate profiling data for your CREDIT_CARD_PKG.
 - a) Re-create CREDIT_CARD_PKG by running the D:\labs\labs\lab_12_01.sql script.
 - b) You need to identify the location of the profiler files. Create a DIRECTORY object to identify this information:

```
CREATE DIRECTORY profile_data AS 'D:\labs\labs';
```
 - c) Use DBMS_HPROF.START_PROFILING to start the profiler for your session.

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

- d) Run your CREDIT_CARD_PKG.UPDATE_CARD_INFO with the following data.

```
credit_card_pkg.update_card_info
(154, 'Discover', '123456789');

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

- e) Use DBMS_HPROF.STOP_PROFILING to stop the profiler.

```
BEGIN
    DBMS_HPROF_STOP_PROFILING;
END;
/
```

- 2) Run the dbmhptab.sql script located in the \home\rdbms\admin folder to set up the profiler tables.

· D:\app\Administrator\product\11.1.0\db_1\RDBMS\ADMIN\dbmhptab.sql

Practice 12-1: Profiling and Tracing (continued)

- 3) Use DBMS_HPROF.ANALYZE to analyze the raw data and write the information to the profiler tables.

- a) Get the RUN_ID.

```
SET SERVEROUT ON

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;
/
```

Run ID: 1

- b) Query the DBMSHP_RUNS table to find top-level information for the RUN_ID that you retrieved.

```
SET VERIFY OFF

SELECT runid, run_timestamp, total_elapsed_time
FROM dbmshp_runs
WHERE runid = &your_run_id;
```

Enter value for your_run_id: 1
1
09-FEB-08 02.49.30.031000 PM
72578

- c) Query the DBMSHP_FUNCTION_INFO table to find information on each function profiled.

```
SELECT owner, module, type, function_line#, namespace,
       calls, function_elapsed_time
FROM   dbmshp_function_info
WHERE  runid = &your_run_id;
```

Practice 12-1: Profiling and Tracing (continued)

Enter value for your_run_id: 1

<u>anonymous_block</u>	PLSQL	2	41
<u>plsql_vm</u>	PLSQL	2	10
OE PACKAGE BODY UPDATE_CARD_INFO	CREDIT_CARD_PKG		
PLSQL		1	150
SYS PACKAGE BODY STOP_PROFILING	DBMS_HPROF		
PLSQL		1	0
OE PACKAGE BODY <u>_static_sql_exec_line21</u>	CREDIT_CARD_PKG		
SQL		1	12424
OE PACKAGE BODY <u>_static_sql_exec_line9</u>	CREDIT_CARD_PKG		
SQL		1	59953

6 rows selected.

(Note: Results are formatted better in Oracle SQL Developer.)

OWNER	MODULE	TYPE	LINE#	NAMESPACE	CALLS	FUNCTION_ELAPSED
1 (null)	(null)	(null)	<u>anonymous_block</u>	PLSQL	2	
2 (null)	(null)	(null)	<u>plsql_vm</u>	PLSQL	2	
3 OE	CREDIT_CARD_PKG	PACKAGE BODY	UPDATE_CARD_INFO	PLSQL	1	
4 SYS	DBMS_HPROF	PACKAGE BODY	STOP_PROFILING	PLSQL	1	
5 OE	CREDIT_CARD_PKG	PACKAGE BODY	<u>_static_sql_exec_line21</u>	SQL	1	
6 OE	CREDIT_CARD_PKG	PACKAGE BODY	<u>_static_sql_exec_line9</u>	SQL	1	

- 4) Use the plshprof command-line utility to generate simple HTML reports directly from the raw profiler data.
 - a) Open a command window.
 - b) Change the working directory to D:\labs\labs.
 - c) Run the plshprof utility.

```
--at your command window, change your working directory to
D:\labs\labs
cd D:\labs\labs
plshprof - output pd_cc_pkg pd_cc_pkg.txt
```

Practice 12-1: Profiling and Tracing (continued)

```
D:\labs\labs>plshprof -output pd_cc_pkg pd_cc_pkg.txt
PLSHPROF: Oracle Database 11g Enterprise Edition Release 11.1.0.6.0 - Producti
[6 symbols processed]
[Report written to 'pd_cc_pkg.html']

D:\labs\labs>
```

- 5) Open the report in your browser and review the data.

In Windows Explorer, navigate to the D:\labs\labs folder.

	Name	Size	Type
	lab_09_05_d	1 KB	SQL File
	lab_09_06_a	2 KB	SQL File
	lab_09_06_b	1 KB	SQL File
	lab_11_01_a	2 KB	SQL File
	lab_11_02_d	1 KB	SQL File
	lab_11_03_a	1 KB	SQL File
	lab_12_01	2 KB	SQL File
	lab_13_06	1 KB	SQL File
	lab_13_07	1 KB	SQL File
	Labs_05	2 KB	WinZip File
	les13_staticsql	1 KB	HTML Document
	les13_staticsql	406 KB	Shockwave Flash
	les13staticsql_demo	4 KB	SQL File
	pd_cc_pkg	3 KB	HTML Document
	pd_cc_pkg	1 KB	Text Document
	pd_cc_pkg_2c	2 KB	HTML Document
	pd_cc_pkg_2f	2 KB	HTML Document
	pd_cc_pkg_2n	2 KB	HTML Document
	pd_cc_pkg_fn	4 KB	HTML Document
	pd_cc_pkg_md	5 KB	HTML Document
	pd_cc_pkg_mf	5 KB	HTML Document
	pd_cc_pkg_ms	5 KB	HTML Document
	pd_cc_pkg_nsc	2 KB	HTML Document
	pd_cc_pkg_nsf	2 KB	HTML Document
	pd_cc_pkg_nsp	1 KB	HTML Document
	pd_cc_pkg_pc	12 KB	HTML Document

Practice 12-1: Profiling and Tracing (continued)

Double-click the pd_cc_pkg.html file.

Examine the different reports.

The screenshot shows a web browser window with the address bar containing "D:\Labs\pd_cc_pkg.html". The main content area has a title "PL/SQL Elapsed Time (microsecs) Analysis" and a subtitle "72578 microsecs (elapsed time) & 8 function calls". Below this, a text block states: "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:". A bulleted list follows, with each item being a link:

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

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 \(microsecs\)](#)
- [Function Elapsed Time \(microsecs\) Data sorted by Mean Descendants Elapsed Time \(microsecs\)](#)
- [Module Elapsed Time \(microsecs\) Data sorted by Total Function Elapsed Time \(microsecs\)](#)
- [Module Elapsed Time \(microsecs\) Data sorted by Module Name](#)
- [Module Elapsed Time \(microsecs\) Data sorted by Total Function Call Count](#)
- [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](#)

Practices for Lesson 13

Practice 13-1: Safeguarding Your Code Against SQL Injection Attacks

In this practice, you examine PL/SQL code, test it for SQL injection, and rewrite it to protect against SQL injection vulnerabilities.

Use your OE connection.

Understanding SQL Injection

- 1) Only code used in Web applications is vulnerable to SQL injection attack.

True/False

- 2) Code that is most vulnerable to SQL injection attack contains (check all that apply):

Input parameters

Dynamic SQL with bind arguments

⇒ Dynamic SQL with concatenated input values

Calls to exterior functions

- 3) By default, a stored procedure or SQL method executes with the privileges of the owner (definer's rights).

True/False

- 4) By using AUTHID CURRENT_USER in your code, you are (check all that apply):

⇒ Specifying that the code executes with invoker's rights

Specifying that the code executes with the highest privilege level

Eliminating any possible SQL injection vulnerability

⇒ Not eliminating all possible SQL injection vulnerabilities

Practice 13-1: Safeguarding Your Code Against SQL Injection Attacks (continued)

- 5) Match each attack surface reduction technique to an example of the technique.
Technique Example

Executes code with minimal privileges: Use invoker's rights

Lockdown the database: Revoke privileges from PUBLIC

Reduce arbitrary input: Specify appropriate parameter types

Rewriting Code to Protect Against SQL Injection

- 6) Examine this code. Run the lab_13_06.sql script to create the procedure.

```
CREATE OR REPLACE PROCEDURE get_income_level
    (p_email VARCHAR2 DEFAULT NULL)
IS
    TYPE      cv_custtyp IS REF CURSOR;
    cv        cv_custtyp;
    v_income  customers.income_level%TYPE;
    v_stmt    VARCHAR2(400);
BEGIN
    v_stmt := 'SELECT income_level FROM customers WHERE
              cust_email = ''' || p_email || '''';

    DBMS_OUTPUT.PUT_LINE('SQL statement: ' || v_stmt);
    OPEN cv FOR v_stmt;
    LOOP
        FETCH cv INTO v_income;
        EXIT WHEN cv%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Income level is: ' || v_income);
    END LOOP;
    CLOSE cv;

EXCEPTION WHEN OTHERS THEN
    dbms_output.PUT_LINE(sqlerrm);
    dbms_output.PUT_LINE('SQL statement: ' || v_stmt);
END get_income_level;
/
```

- a) Execute the following statements and note the results.

```
exec get_income_level('Sean.Olin@SCAUP.COM')
```

```
SQL statement: SELECT income_level FROM customers WHERE cust_email =
'Sean.Olin@SCAUP.COM'
Income level is: F: 110,000 - 129,999
```

Practice 13-1: Safeguarding Your Code Against SQL Injection Attacks (continued)

```
exec get_income_level('x' union select username from all_users where ''x''='x')

SQL> exec get_income_level('x' union select username from all_users where ''x''='x')
SQL statement: SELECT income_level FROM customers WHERE cust_email = 'x' union
select username from all_users where 'x'='x'
Income level is: ANONYMOUS
Income level is: APEX_PUBLIC_USER
Income level is: BI
Income level is: CTXSYS
Income level is: DBSNMP
Income level is: DIP
Income level is: EXFSYS
Income level is: FLOWS_030000
Income level is: FLOWS_FILES
Income level is: HR
Income level is: IX
Income level is: MDDATA
Income level is: MDSYS
Income level is: MGMT_VIEW
Income level is: OE
Income level is: OLAPSYS
Income level is: ORACLE_OCM
Income level is: ORDPLUGINS
Income level is: ORDSYS
Income level is: OUTLN
Income level is: OWBSYS
Income level is: PM
Income level is: SCOTT
Income level is: SH
Income level is: SI_INFORMTN_SCHEMA
```

- b) Has SQL injection occurred?

Yes, by using dynamic SQL constructed via concatenation of input values, you see all users in the database.

Rewriting Code to Protect Against SQL Injection (continued)

- 7) Rewrite the code to protect against SQL injection. You can run the lab_13_07.sql script to recreate the procedure.

```
CREATE OR REPLACE
PROCEDURE get_income_level (p_email VARCHAR2 DEFAULT NULL)
AS
BEGIN
FOR i IN
  (SELECT income_level
   FROM customers
   WHERE cust_email = p_email)
LOOP
  DBMS_OUTPUT.PUT_LINE('Income level is:
    '||i.income_level);
END LOOP;
END get_income_level;
/
```

- a) Execute the following statements and note the results.

```
exec get_income_level('Sean.Olin@SCAUP.COM')
```

```
SQL statement: SELECT income_level FROM customers WHERE cust_email =
'Sean.Olin@SCAUP.COM'
Income level is: F: 110,000 - 129,999
```

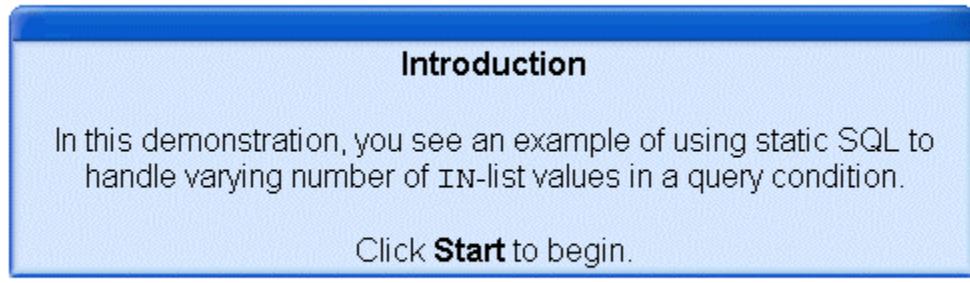
Practice 13-1: Safeguarding Your Code Against SQL Injection Attacks (continued)

```
exec get_income_level('x' union select username from all_users where ''x''='x')
```

```
SQL> exec get_income_level('x' union select username from all_users where ''x''='x')
PL/SQL procedure successfully completed.
```

- b) Has SQL injection occurred?
No
- 8) Play the `les13_staticsql` viewlet located in the `D:\labs\labs\viewlets` folder. This is an example of using static SQL to handle a varying number of IN-list values in a query condition.

To play a viewlet, double-click the `les13_staticsql.htm` file located in the `D:\labs\labs\viewlets` folder. When prompted, click Start.



B

Table Descriptions and Data

ORACLE

Copyright © 2008, Oracle. All rights reserved.

Schema Descriptions

Overall Description

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

- The Human Resources division tracks information about the employees and facilities of the company.
- The Order Entry division tracks product inventories and sales of the products of the company through various channels.
- The Sales History division tracks business statistics to facilitate business decisions.

Each division is represented by a schema. In this course, you have access to the objects in all these schemas. However, the emphasis of the examples, demonstrations, and practices utilizes the Order Entry (OE) schema.

All scripts necessary to create the sample schemas reside in the
\$ORACLE_HOME/demo/schema/ folder.

Schema Descriptions (continued)

Order Entry (OE)

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

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

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

Customer information is tracked in some detail. Each customer is assigned an identification number. Customer records include name, street address, city or province, country, phone numbers (up to five phone numbers for each customer), and postal code. Some customers order through the Internet, so email addresses are also recorded. Because of language differences among customers, the company records the National Language Support (NLS) language and territory of each customer. The company places a credit limit on its customers to limit the amount for which they can purchase at one time. Some customers have account managers, whom the company monitors. It keeps track of a customer's phone number. These days, you never know how many phone numbers a customer might have, but you try to keep track of all of them. Because of the language differences of the customers, you identify the language and territory of each customer.

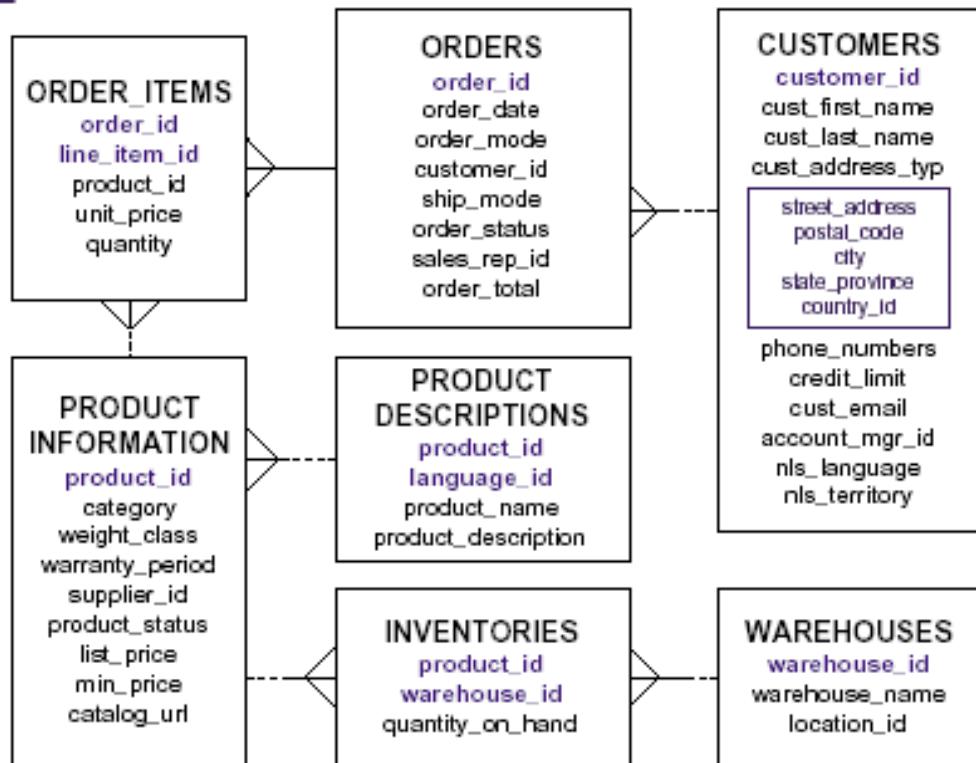
When a customer places an order, the company tracks the date of the order, the mode of the order, status, shipping mode, total amount of the order, and the sales representative who helped place the order. This may be the same individual as the account manager for a customer, it may be a different individual,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

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

Schema Descriptions (continued)

Order Entry (OE)

OE



Schema Descriptions (continued)

Order Entry (OE) Row Counts

```
SELECT COUNT(*) FROM customers;
  COUNT(*)
-----
      319

SELECT COUNT(*) FROM inventories;
  COUNT(*)
-----
      1112

SELECT COUNT(*) FROM orders;
  COUNT(*)
-----
      105

SELECT COUNT(*) FROM order_items;
  COUNT(*)
-----
      665

SELECT COUNT(*) FROM product_descriptions;
  COUNT(*)
-----
      8640

SELECT COUNT(*) FROM product_information;
  COUNT(*)
-----
      288

SELECT COUNT(*) FROM warehouses;
  COUNT(*)
-----
      9
```

Schema Descriptions (continued)

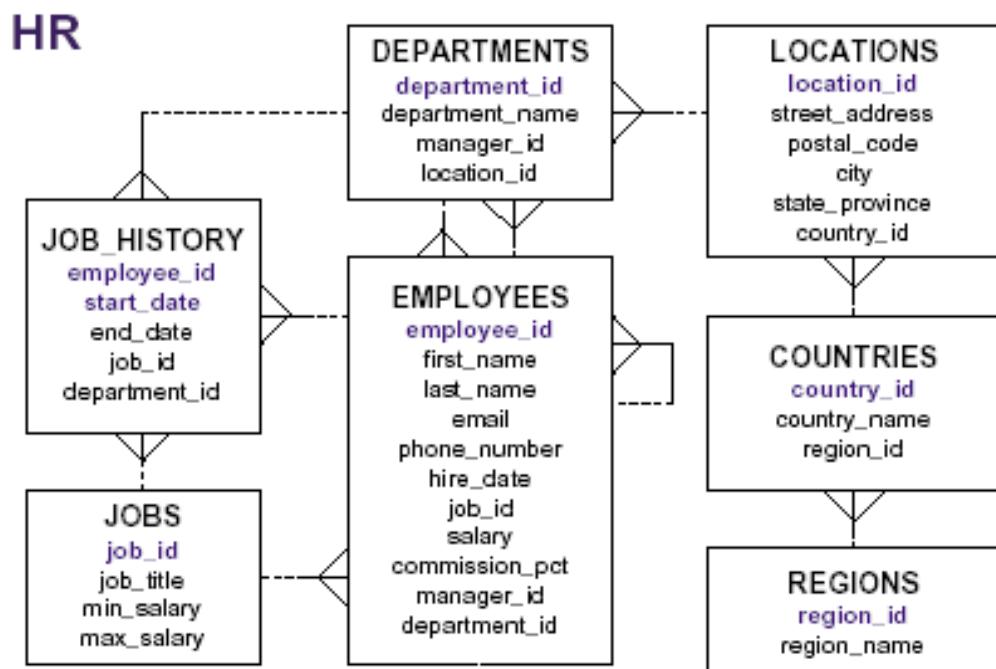
Human Resources (HR)

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

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

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

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



Schema Descriptions (continued)

Human Resources (HR) Row Counts

```
SELECT COUNT(*) FROM employees;
  COUNT(*)
-----
      107

SELECT COUNT(*) FROM departments;
  COUNT(*)
-----
      27

SELECT COUNT(*) FROM locations;
  COUNT(*)
-----
      23

SELECT COUNT(*) FROM countries;
  COUNT(*)
-----
      25

SELECT COUNT(*) FROM regions;
  COUNT(*)
-----
      4

SELECT COUNT(*) FROM jobs;
  COUNT(*)
-----
      19

SELECT COUNT(*) FROM job_history;
  COUNT(*)
-----
      10
```


Using SQL Developer

ORACLE

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

- List the key features of Oracle SQL Developer
- Install Oracle SQL Developer
- Identify 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

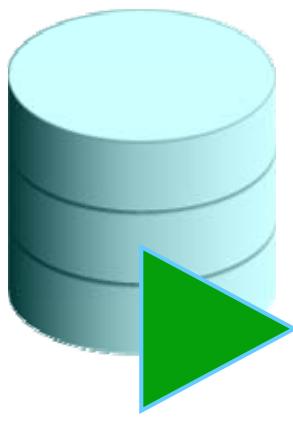


Objectives

In this appendix, you are introduced to the graphical tool called SQL Developer. You learn how to use SQL Developer for your database development tasks. You learn how to use SQL worksheet to execute SQL statements and SQL scripts.

What Is Oracle SQL Developer?

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



SQL Developer

ORACLE

C - 3

Copyright © 2008, Oracle. All rights reserved.

What Is Oracle SQL Developer?

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

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

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

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

Note: The SQL Developer 1.2 release is called the *Migration release*, because it tightly integrates with *Developer Migration Workbench* that provides users with a single point to browse database objects and data in third-party databases, and to migrate from these databases to Oracle. You can also connect to schemas for selected third-party (non-Oracle) databases, such as MySQL, Microsoft SQL Server, and Microsoft Access, and you can view metadata and data in these databases.

Additionally, SQL Developer 1.2 includes support for Oracle Application Express 3.0.1 (Oracle APEX).

Specifications of SQL Developer

- Developed in Java
- Supports Windows, Linux, and Mac OS X platforms
- Default connectivity by using the JDBC Thin driver
- Does not require an installer
 - Unzip the downloaded SQL Developer kit and double-click `sqldeveloper.exe` to start SQL Developer.
- Connects to any Oracle Database version 9.2.0.1 and later
- Freely downloadable from the following link:
 - <http://www.oracle.com/technology/software/products/sql/index.html>
- Needs JDK 1.5 to be installed on your system. It can be downloaded from the following link:
 - http://java.sun.com/javase/downloads/index_jdk5.jsp



Specifications of SQL Developer

Oracle SQL Developer is developed in Java by leveraging the Oracle JDeveloper integrated development environment (IDE). Therefore, it is a cross-platform tool. The tool runs on Windows, Linux, and Mac operating system (OS) X platforms. You can install SQL Developer on the Database Server and connect remotely from your desktop, thus avoiding client/server network traffic.

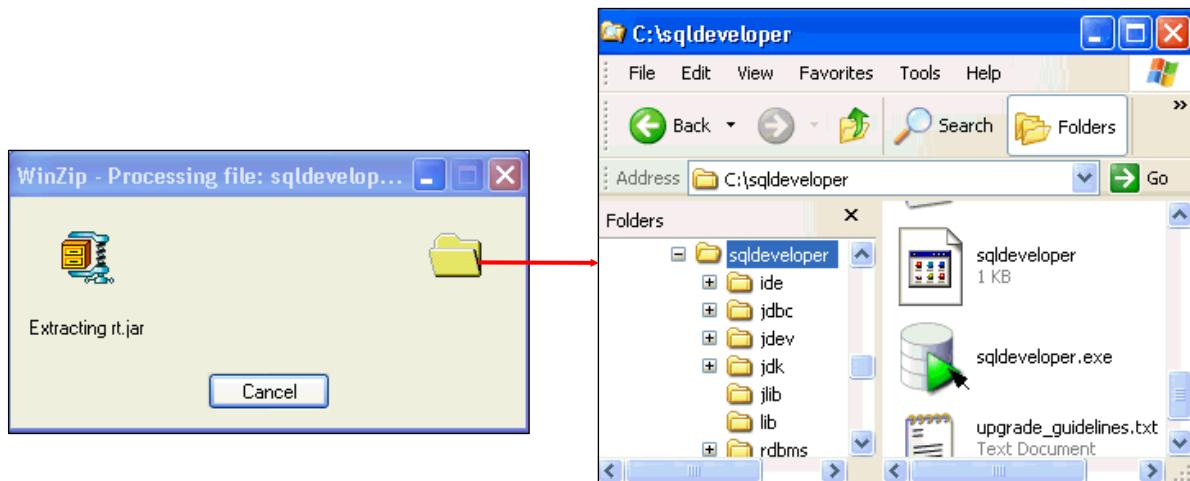
Default connectivity to the database is through the Java Database Connectivity (JDBC) Thin driver, and therefore, no Oracle Home is required. SQL Developer does not require an installer and you need to simply unzip the downloaded file. With SQL Developer, users can connect to Oracle Databases 9.2.0.1 and later, and all Oracle database editions, including Express Edition.

SQL Developer can be downloaded with the following packaging options:

- Oracle SQL Developer for Windows (option to download with or without Java Development Kit [JDK] 1.5)
- Oracle SQL Developer for Multiple Platforms (JDK 1.5 should already be installed.)
- Oracle SQL Developer for Mac OS X platforms (JDK 1.5 should already be installed.)
- Oracle SQL Developer Red Hat Package Manager (RPM) for Linux (JDK 1.5 should already be installed.)

Installing SQL Developer

Download the Oracle SQL Developer kit and unzip into any directory on your machine.



Installing SQL Developer

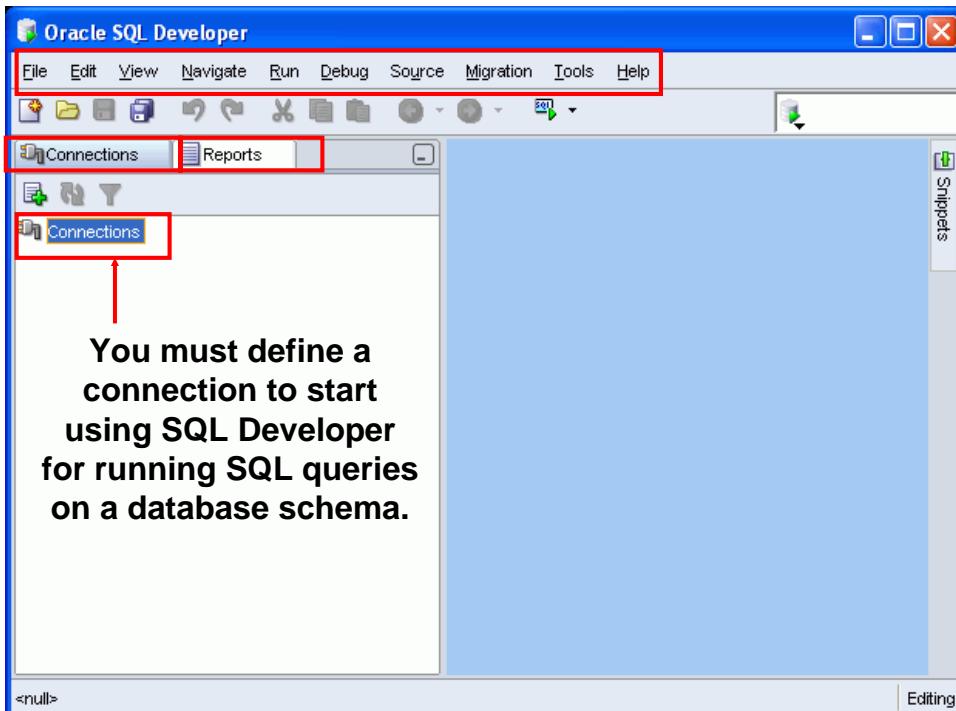
Oracle SQL Developer does not require an installer. To install SQL Developer, you need an unzip tool.

To install SQL Developer, perform the following steps:

1. Create a folder as <local drive>:\SQL Developer.
2. Download the SQL Developer kit from
<http://www.oracle.com/technology/software/products/sql/index.html>.
3. Unzip the downloaded SQL Developer kit into the folder that was created in step 1.

To start SQL Developer, go to <local drive>:\SQL Developer, and double-click `sqldeveloper.exe`.

SQL Developer Interface



ORACLE

C - 6

Copyright © 2008, Oracle. All rights reserved.

SQL Developer Interface

SQL Developer has two main navigation tabs:

- **Connections navigator:** By using this tab, you can browse database objects and users to which you have access.
- **Reports Tab:** By using this tab, you can run predefined reports, or create and add your own reports.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about the selected objects. You can customize many aspects of the appearance and behavior of SQL Developer by setting preferences.

The menus at the top contain standard entries, plus entries for features specific to SQL Developer:

- **View:** Contains options that affect what is displayed in the SQL Developer interface
- **Navigate:** Contains options for navigating to panes and for executing subprograms
- **Run:** Contains the Run File and the Execution Profile options that are relevant when a function or procedure is selected
- **Debug:** Contains options that are relevant when a function or procedure is selected for debugging
- **Source:** Contains options for use when editing functions and procedures
- **Migration:** Contains options related to migrating third-party databases to Oracle
- **Tools:** Invokes SQL Developer tools, such as SQL*Plus, Preferences, and SQL worksheet

Note: You need to define at least one connection to be able to connect to a database schema and issue SQL queries or run procedures or functions.

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
 - For multiple schemas
- SQL Developer automatically imports connections defined in the `tnsnames.ora` file on your system.
- You can export connections to an XML file.
- Each additional database connection that is created is listed in the Connections navigator hierarchy.



Creating a Database Connection

A connection is a SQL Developer object that specifies the necessary information for connecting to a specific database as a specific user of that database. To use SQL Developer, you must have at least one database connection, which may be existing, created, or imported.

You can create and test connections for multiple databases and for multiple schemas.

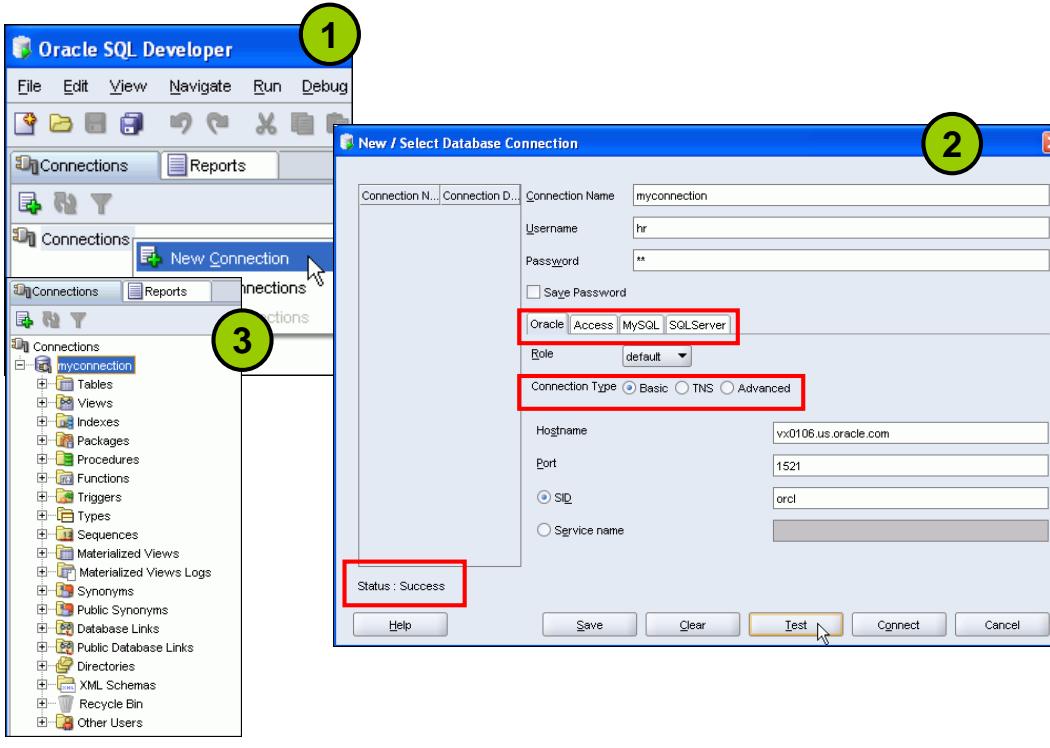
By default, the `tnsnames.ora` file is located in the `$ORACLE_HOME/network/admin` directory. But, it can also be in the directory specified by the `TNS_ADMIN` environment variable or registry value. When you start SQL Developer and display the Database Connections dialog box, SQL Developer automatically imports any connections defined in the `tnsnames.ora` file on your system.

Note: On Windows systems, if the `tnsnames.ora` file exists but its connections are not being used by SQL Developer, define `TNS_ADMIN` as a system environment variable.

You can export connections to an XML file so that you can reuse it later.

You can create additional connections as different users to the same database or to connect to the different databases.

Creating a Database Connection



ORACLE

Creating a Database Connection (continued)

To create a database connection, perform the following steps:

1. On the Connections tabbed page, right-click **Connections** and select **New Connection**.
2. In the New/Select Database Connection window, enter the connection name, username, and password of the schema that you want to connect to.
3. From the Role drop-down list, select either *default* or SYSDBA. (Choose SYSDBA for the sys user or any user with DBA privileges.)
4. Select the connection type:
 - **Basic:** In this type, enter host name and SID for the database that you want to connect to. Port is already set to 1521. Or, you can also choose to enter the Service name directly if you are using a remote database connection.
 - **TNS:** You can select any one of the database aliases imported from the tnsnames.ora file.
 - **Advanced:** You can define a custom JDBC URL to connect to the database.
5. Click **Test** to make sure that the connection was set correctly.
6. Click **Connect**.

If you select the Save Password check box, the password is saved to an XML file. So, after you close the SQL Developer connection and open it again, you are not prompted for the password.

Creating a Database Connection (continued)

7. The connection is added to the Connections navigator. You can expand the connection to view the database objects and view object definitions; for example, dependencies, details, statistics, and so on.

Note: From the same New>Select Database Connection window, you can define connections to non-Oracle data sources by using the Access, MySQL, and SQL Server tabs. However, these connections are read-only connections that enable you to browse objects and data in that data source.

Browsing Database Objects

Use the Connections navigator to:

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

The screenshot shows the Oracle SQL Developer interface. On the left, the Connections navigator is open, showing a tree structure of database objects under 'myconnection'. A red box highlights the 'Tables' node, which is expanded to show 'COUNTRIES', 'DEPARTMENTS', 'EMPLOYEES' (selected), 'JOB_HISTORY', 'JOBS', 'LOCATIONS', and 'REGIONS'. To the right of the navigator, the 'EMPLOYEES' table is selected in a tabbed interface. A second red box highlights the 'Columns' tab, which is active and displays the following table definition:

Column Name	Data Type	Nullable	Data Default
EMPLOYEE_ID	NUMBER(6,0)	No	(null)
FIRST_NAME	VARCHAR2(20 BYTE)	Yes	(null)
LAST_NAME	VARCHAR2(25 BYTE)	No	(null)
EMAIL	VARCHAR2(25 BYTE)	No	(null)
PHONE_NUMBER	VARCHAR2(20 BYTE)	Yes	(null)
HIRE_DATE	DATE	No	(null)
JOB_ID	VARCHAR2(10 BYTE)	No	(null)
SALARY	NUMBER(8,2)	Yes	(null)
COMMISSION_PCT	NUMBER(2,2)	Yes	(null)
MANAGER_ID	NUMBER(6,0)	Yes	(null)
DEPARTMENT_ID	NUMBER(4,0)	Yes	(null)

Browsing Database Objects

After creating a database connection, you can use the Connections navigator to browse through many objects in a database schema, including tables, views, indexes, packages, procedures, triggers, and types.

SQL Developer uses the left side for navigation to find and select objects, and the right side to display information about the selected objects. You can customize many aspects of the appearance of SQL Developer by setting preferences.

You can see the definition of the objects broken into tabs of information that is pulled from the data dictionary. For example, if you select a table in the navigator, the details about columns, constraints, grants, statistics, triggers, and so on are displayed on an easy-to-read tabbed page.

If you want to see the definition of the EMPLOYEES table as shown in the slide, perform the following steps:

1. Expand the Connections node in the Connections navigator.
2. Expand **Tables**.
3. Click **EMPLOYEES**. By default, the Columns tab is selected. It shows the column description of the table. On the Data tab, you can view the table data and also enter new rows, update data, and commit these changes to the database.

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 or one of the many context-sensitive menus.
- View the DDL for adjustments, such as creating a new object or editing an existing schema object.



ORACLE

C - 11

Copyright © 2008, Oracle. All rights reserved.

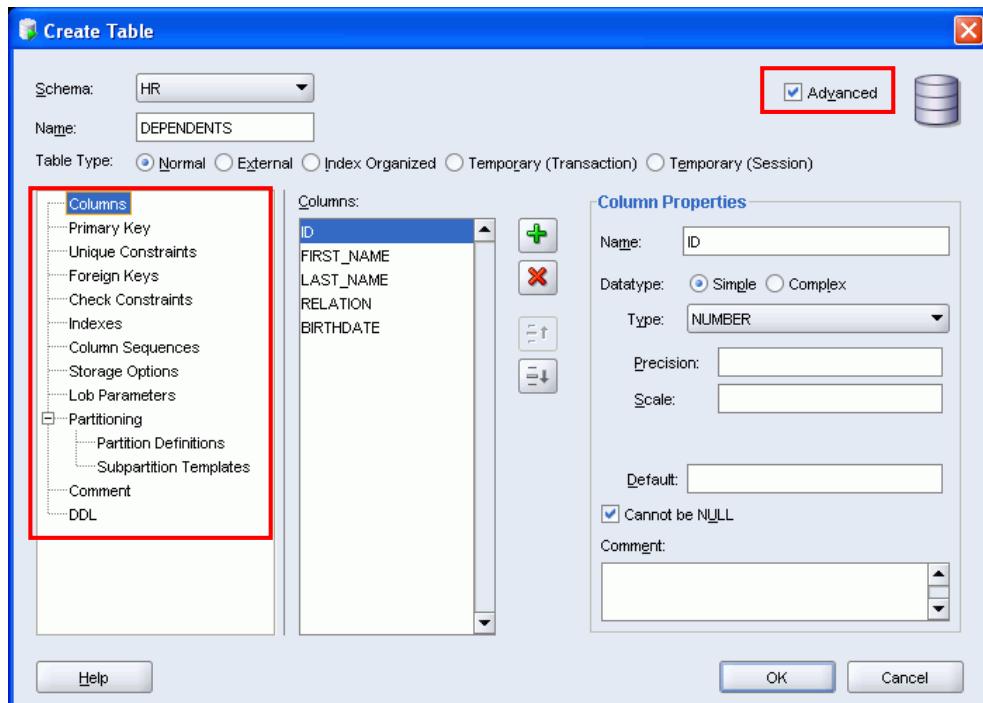
Creating a Schema Object

SQL Developer supports the creation of any schema object by executing a SQL statement in SQL worksheet. Alternatively, you can create objects by using the context menus. After they are created, you can edit the objects by using an edit dialog or one of the many context-sensitive menus.

As new objects are created or the existing objects are edited, the data definition language (DDL) for those adjustments is available for review. An Export DDL option is available if you want to create the full DDL for one or more objects in the schema.

The slide shows how to create a table by using the context menu. To open a dialog box for creating a new table, right-click **Tables** and select **New Table**. The dialog boxes for creating and editing database objects have multiple tabs, each reflecting a logical grouping of properties for that type of object.

Creating a Table: Example



ORACLE

C - 12

Copyright © 2008, Oracle. All rights reserved.

Creating a Table: Example

In the Create Table dialog box, if you do not select the **Advanced** check box, you can create a table quickly by specifying columns and some frequently used features.

If you select the **Advanced** check box, the Create Table dialog box changes to one with multiple options, in which you can specify an extended set of features while creating the table.

The example in the slide shows how to create the DEPENDENTS table by selecting the **Advanced** check box.

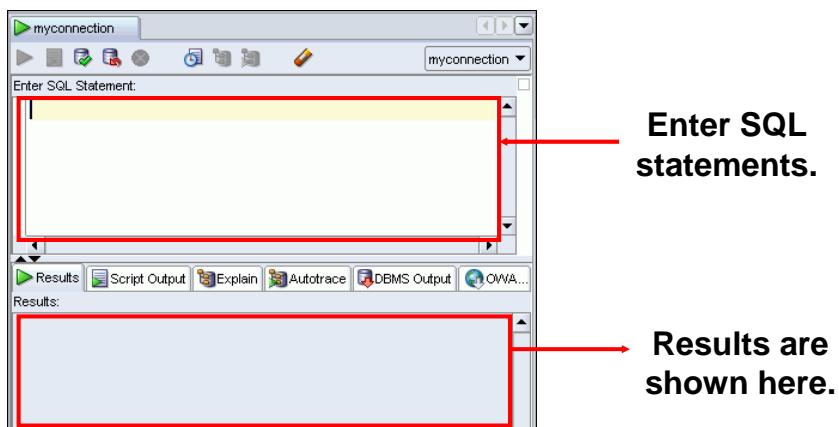
To create a table, perform the following steps:

1. In the Connections navigator, right-click **Tables**.
2. Select **New Table**.
3. In the Create Table dialog box, select **Advanced**.
4. Specify column information.
5. Click **OK**.

Although it is not required, you should also specify a primary key by using the Primary Key tab in the dialog box. Sometimes, you may want to edit the table that you created. To edit a table, right-click the table in the Connections navigator and select **Edit**.

Using the SQL Worksheet

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



ORACLE

Using the SQL Worksheet

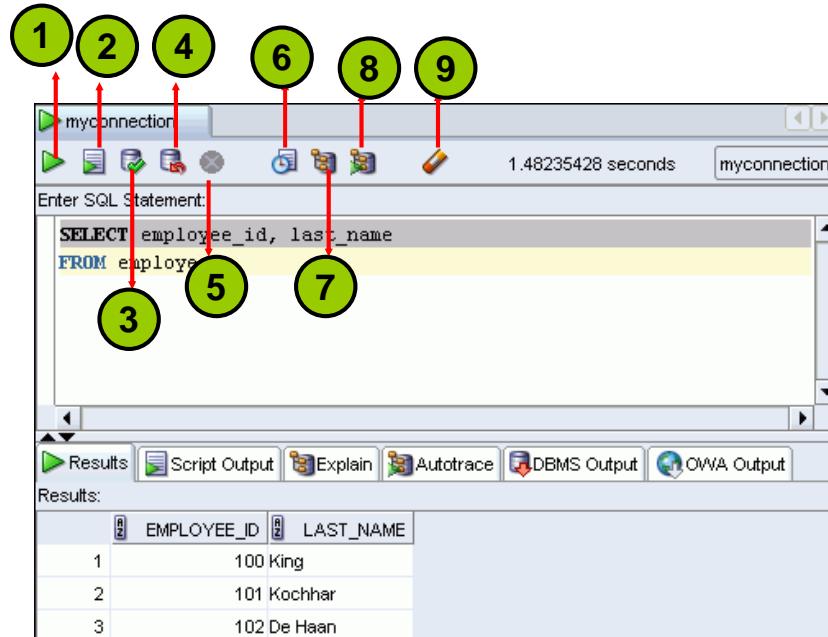
When you connect to a database, a SQL worksheet window for that connection automatically opens. You can use the SQL worksheet to enter and execute SQL, PL/SQL, and SQL*Plus statements. All SQL and PL/SQL commands are supported as they are passed directly from the SQL worksheet to the Oracle database. The SQL*Plus commands used in SQL Developer must be interpreted by the SQL worksheet before being passed to the database.

The SQL worksheet currently supports a number of SQL*Plus commands. Commands not supported by the SQL worksheet are ignored and are not sent to the Oracle database. Through the SQL worksheet, you can execute SQL statements and some of the SQL*Plus commands.

You can display a SQL worksheet by using any of the following options:

- Select **Tools > SQL Worksheet**.
- Click the **Open SQL Worksheet** icon.

Using the SQL Worksheet



ORACLE

C - 14

Copyright © 2008, Oracle. All rights reserved.

Using the SQL Worksheet (continued)

You may want to use shortcut keys or icons to perform certain tasks, such as executing a SQL statement, running a script, and viewing the history of the SQL statements that you executed.

You can use the SQL worksheet toolbar that contains icons to perform the following tasks:

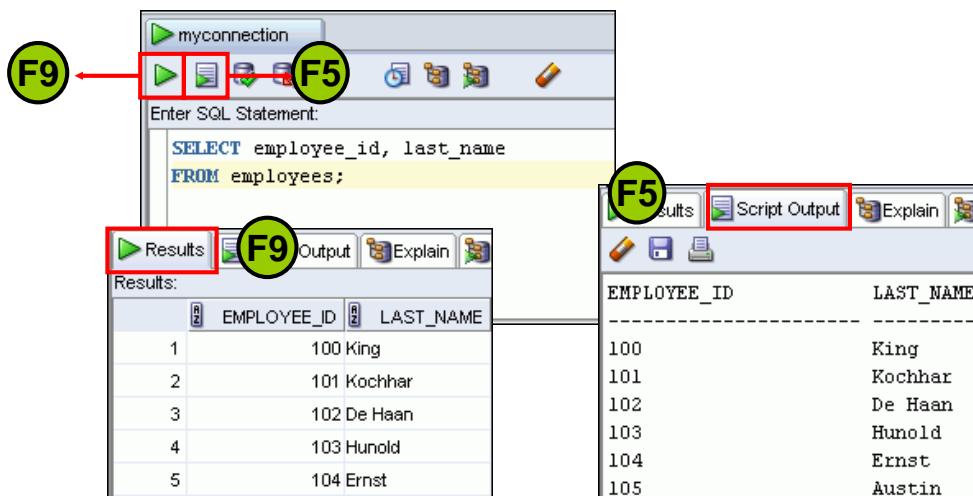
1. **Execute Statement:** Executes the statement at the cursor in the Enter SQL Statement box. Alternatively, you can press F9. The output is generally shown in a formatted manner on the Results tabbed page.
2. **Run Script:** Executes all statements in the Enter SQL Statement box by using the Script Runner. The output is generally shown in the conventional script format on the Scripts tab page.
3. **Commit:** Writes changes to the database, and ends the transaction.
4. **Rollback:** Discards changes to the database, without writing them to the database, and ends the transaction.
5. **Cancel:** Stops the execution of statements that are being executed.

Using the SQL Worksheet (continued)

6. **SQL History:** Displays a dialog box with information about the SQL statements that you executed.
7. **Execute Explain Plan:** Generates the execution plan, which you can see by clicking the Explain tab.
8. **Autotrace:** Displays trace-related information when you execute the SQL statement by clicking the Autotrace icon. This information can help you identify SQL statements that will benefit from tuning.
9. **Clear:** Erases the statements in the Enter SQL Statement box. Alternatively, press Ctrl + D to erase the statements.

Executing SQL Statements

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



ORACLE

C - 16

Copyright © 2008, Oracle. All rights reserved.

Executing SQL Statements

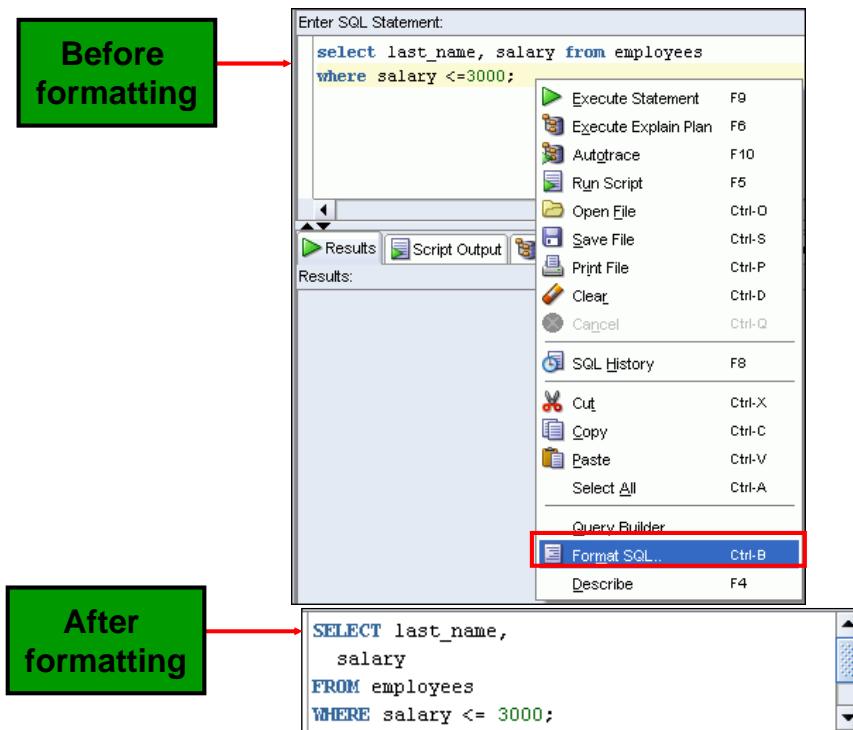
In the SQL worksheet, you can use the Enter SQL Statement box to enter a single or multiple SQL statements. For a single statement, the semicolon at the end is optional.

When you enter the statement, the SQL keywords are automatically highlighted. To execute a SQL statement, ensure that your cursor is within the statement and click the **Execute Statement** icon. Alternatively, you can press the **F9** key.

To execute multiple SQL statements and see the results, click the **Run Script** icon. Alternatively, you can press the **F5** key.

The example in the slide shows the difference in output for the same query when the F9 key or Execute Statement is used versus the output when F5 or Run Script is used.

Formatting the SQL Code



ORACLE

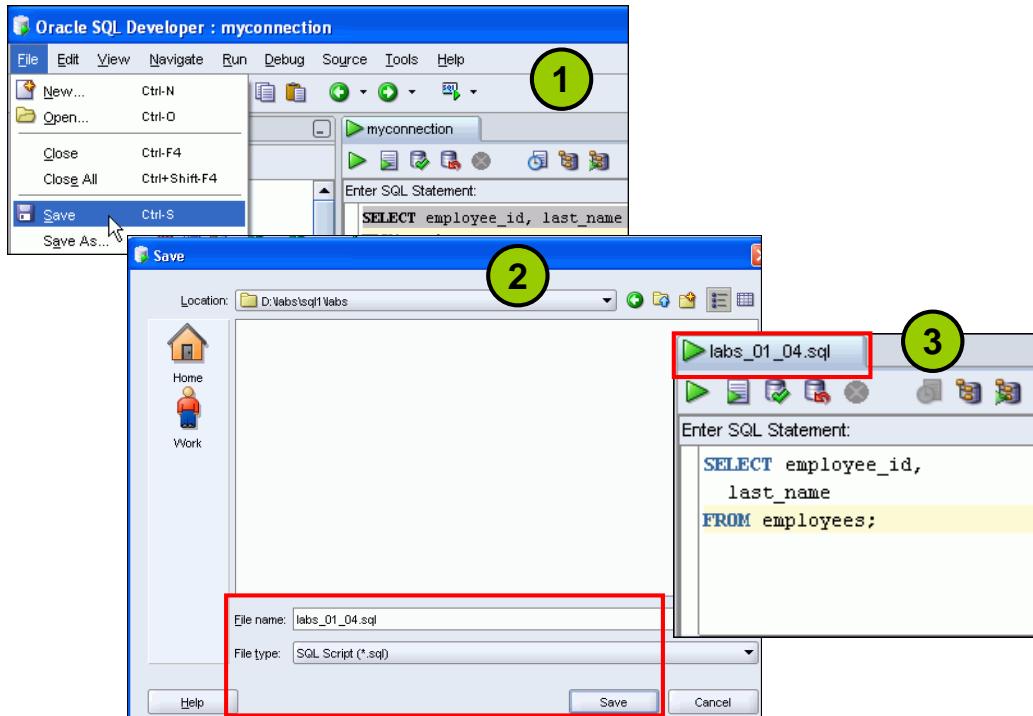
Formatting the SQL Code

You may want to format the indentation, spacing, capitalization, and line separation of the SQL code. SQL Developer has the feature of formatting the SQL code.

To format the SQL code, right-click in the statement area, and select **Format SQL**.

In the example in the slide, before formatting, the SQL code keywords are not capitalized and the statement is not properly indented. After formatting, the SQL code is formatted with the keywords capitalized and the statement properly indented.

Saving SQL Statements

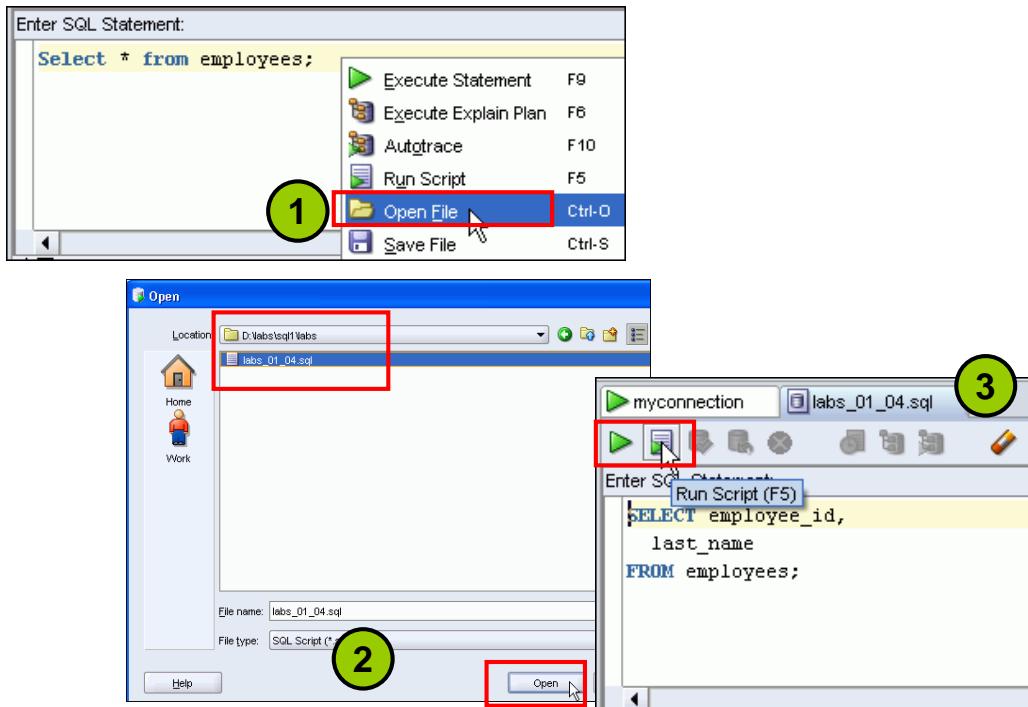


Saving SQL Statements

In most of the practices, you must save a particular query in a SQL worksheet as a .sql file. To do so, perform the following:

1. From the File menu, select **Save** or **Save As** (if you are renaming a current .sql script), or you can right-click in the SQL worksheet and select **Save File**. Alternatively, you can press Ctrl + S.
2. In the Save dialog box, enter the file name. Make sure that the extension is .sql or the File type is selected as SQL Script (*.sql). Click **Save**. **Note:** For this course, you need to save your SQL scripts in the D:\labs\sql1\labs folder.
3. The SQL worksheet is renamed as the file name that you saved the script as. Make sure that you do not enter any other SQL statements in the same worksheet. To continue with other SQL queries, open a new worksheet.

Running Script Files



ORACLE

C - 19

Copyright © 2008, Oracle. All rights reserved.

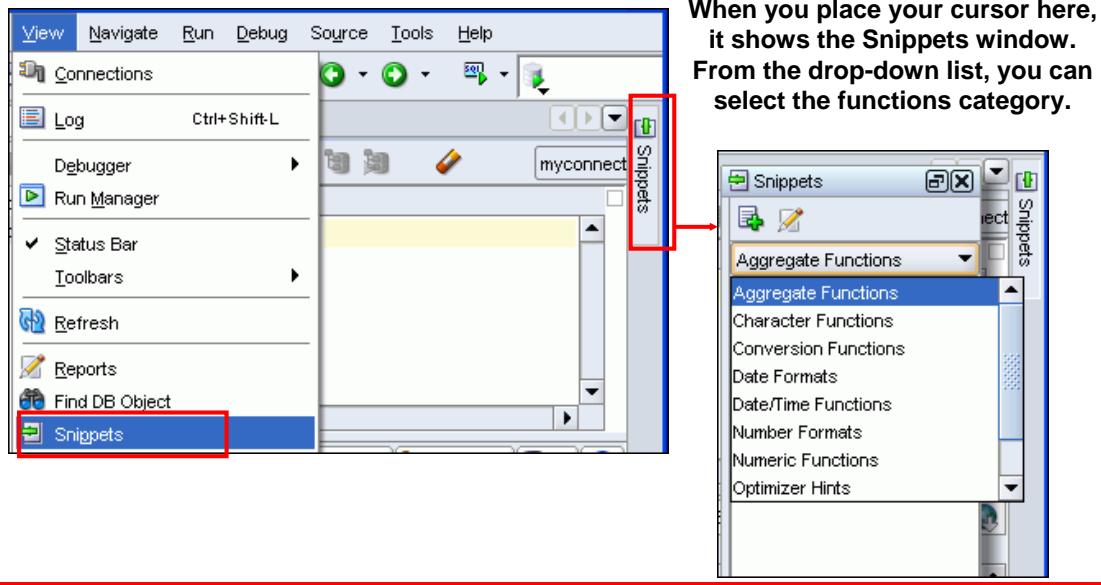
Running Script Files

To run the saved .sql script files, perform the following:

1. Right-click in the SQL worksheet and select **Open File**, or select **Open** from the File menu. Alternatively, you can press Ctrl + O.
2. In the Open dialog box, move to the D:\labs\sql\labs folder or to the location where you saved the script file, select the file, and click **Open**.
3. The script file opens in a new worksheet. Now, you can run the script by either clicking the Execute Statement icon or the Run Script icon. Again, make sure that you do not enter any other SQL statements in the same worksheet. To continue with other SQL queries, open a new worksheet.

Using Snippets

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



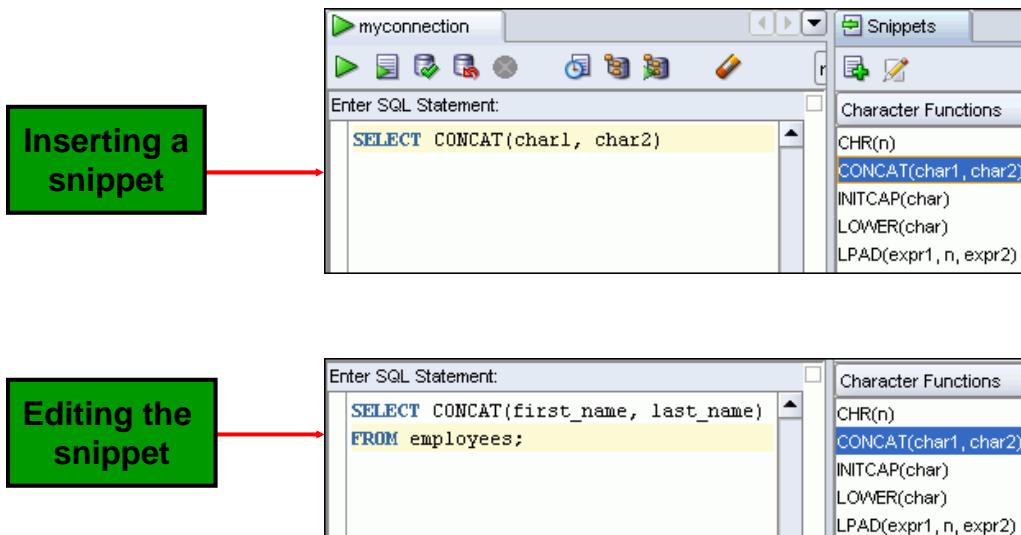
Using Snippets

You may want to use certain code fragments when you are using the SQL worksheet, or creating or editing a PL/SQL function or procedure. SQL Developer has a feature called Snippets. Snippets are code fragments, such as SQL functions, Optimizer hints, and miscellaneous PL/SQL programming techniques. You can drag snippets to the Editor window.

To display Snippets, select **View > Snippets**.

The Snippets window appears on the right side. You can use the drop-down list to select a group. A Snippets button is placed in the right window margin, so that you can display the Snippets window if it becomes hidden.

Using Snippets: Example



ORACLE

Using Snippets: Example

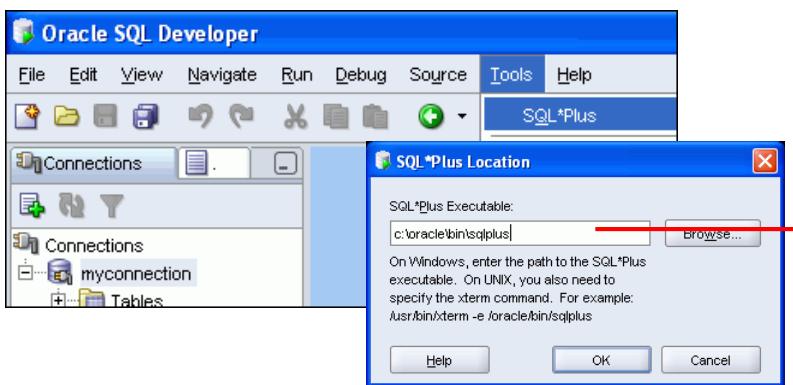
To insert a Snippet into your code in a SQL worksheet or in a PL/SQL function or procedure, drag the snippet from the Snippets window to your code. Then, you can edit the syntax so that the SQL function is valid in the current context. To see a brief description of a SQL function in a tool tip, place the cursor over the function name.

The example in the slide shows that `CONCAT(char1, char2)` is dragged from the Character Functions group in the Snippets window. Then, the `CONCAT` function syntax is edited and the rest of the statement is added such as in the following:

```
SELECT CONCAT(first_name, last_name)  
FROM employees;
```

Using SQL*Plus

- You can invoke the SQL*Plus command-line interface from SQL Developer.
- Close all SQL worksheets to enable the SQL*Plus menu option.



Provide the location of the `sqlplus.exe` file only the first time you invoke SQL*Plus.

ORACLE

Using SQL*Plus

The SQL worksheet supports most of the SQL*Plus statements. SQL*Plus statements must be interpreted by the SQL worksheet before being passed to the database; any SQL*Plus statements that are not supported by the SQL worksheet are ignored and not passed to the database. To display the SQL*Plus command window, from the Tools menu, select **SQL*Plus**. To use this feature, the system on which you are using SQL Developer must have an Oracle home directory or folder, with a SQL*Plus executable under that location. If the location of the SQL*Plus executable is not already stored in your SQL Developer preferences, you are asked to specify its location.

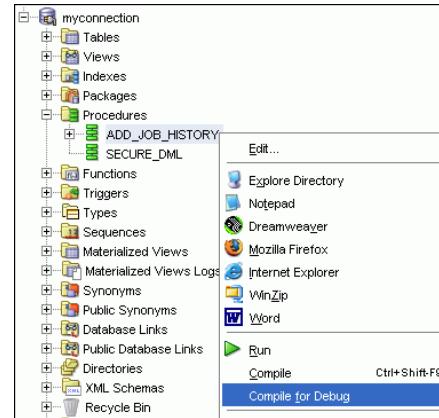
For example, some of the SQL*Plus statements that are not supported by SQL worksheet are:

- append
- archive
- attribute
- break

For a complete list of the SQL*Plus statements that are either supported or not supported by SQL worksheet, refer to the “SQL*Plus Statements Supported and Not Supported in SQL Worksheet” topic in SQL Developer online Help.

Debugging Procedures and Functions

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



ORACLE

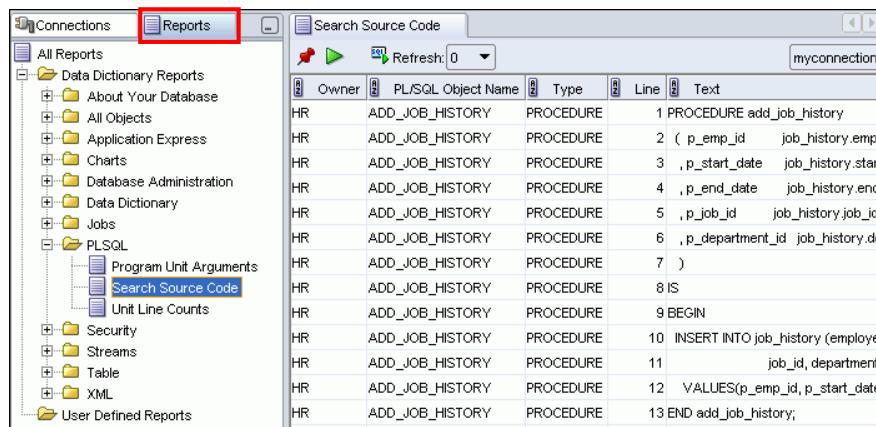
Debugging Procedures and Functions

In SQL Developer, you can debug PL/SQL procedures and functions. Using the Debug menu options, you can perform the following debugging tasks:

- **Find Execution Point** goes to the next execution point.
- **Resume** continues execution.
- **Step Over** bypasses the next method and goes to the next statement after the method.
- **Step Into** goes to the first statement in the next method.
- **Step Out** leaves the current method and goes to the next statement.
- **Step to End of Method** goes to the last statement of the current method.
- **Pause** halts execution but does not exit, thus allowing you to resume execution.
- **Terminate** halts and exits the execution. You cannot resume execution from this point; instead, to start running or debugging from the beginning of the function or procedure, click the Run or Debug icon on the Source tab toolbar.
- **Garbage Collection** removes invalid objects from the cache in favor of more frequently accessed and more valid objects.

Database Reporting

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



ORACLE

Database Reporting

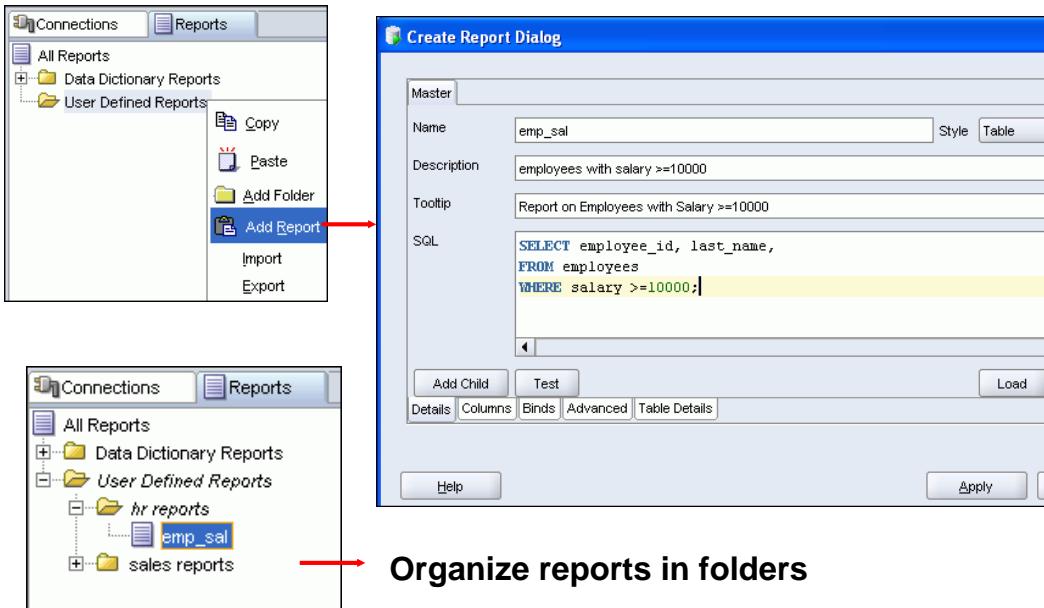
SQL Developer provides many reports about the database and its objects. These reports can be grouped into the following categories:

- About Your Database reports
- Database Administration reports
- Table reports
- PL/SQL reports
- Security reports
- XML reports
- Jobs reports
- Streams reports
- All Objects reports
- Data Dictionary reports
- User-Defined reports

To display reports, click the Reports tab on the left side of the window. Individual reports are displayed in tabbed panes on the right side of the window; and for each report, you can select (using a drop-down list) the database connection for which to display the report. For reports about objects, the objects shown are only those visible to the database user associated with the selected database connection, and the rows are usually ordered by Owner. You can also create your own user-defined reports.

Creating a User-Defined Report

Create and save user-defined reports for repeated use.



ORACLE

Creating a User-Defined Report

User-defined reports are reports created by SQL Developer users.

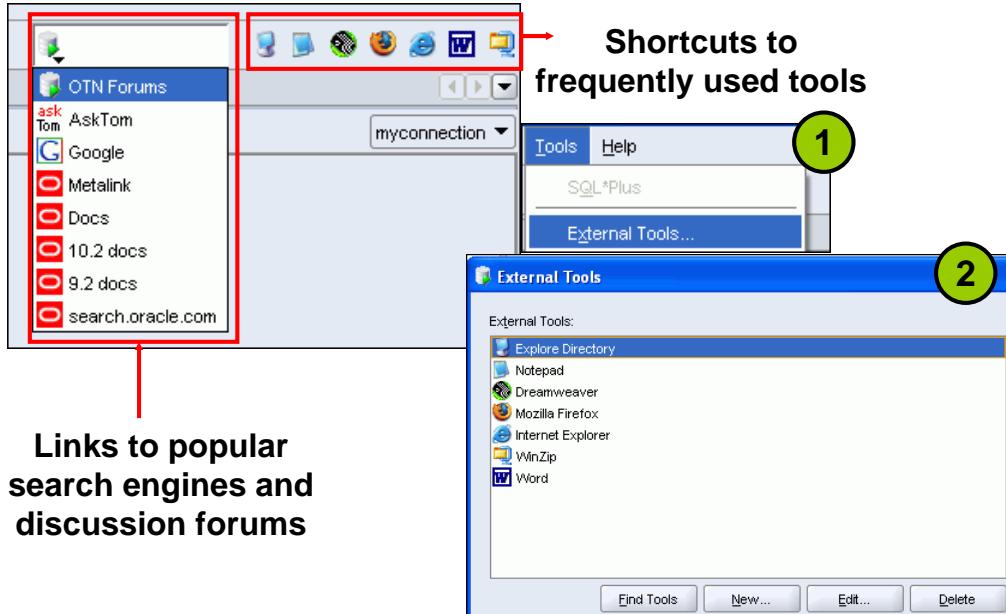
To create a user-defined report, perform the following steps:

- Right-click the User Defined Reports node under Reports, and select Add Report.
- In the Create Report Dialog box, specify the report name and the SQL query to retrieve the information for the report, and click Apply.

In the example in the slide, the report name is specified as `emp_sal`. An optional description indicates that the report contains details of employees with `salary >= 10000`. The complete SQL statement for retrieving the information to be displayed in the user-defined report is specified in the SQL box. You can also include an optional tool tip to be displayed when the cursor hovers over the report name in the Reports navigator display.

You can organize user-defined reports in folders, and you can create a hierarchy of folders and subfolders. To create a folder for user-defined reports, right-click the User Defined Reports node or any folder name under that node and select Add Folder. Information about user-defined reports, including any folders for these reports, is stored in a file named `UserReports.xml` under the directory for user-specific information.

Search Engines and External Tools



Search Engines and External Tools

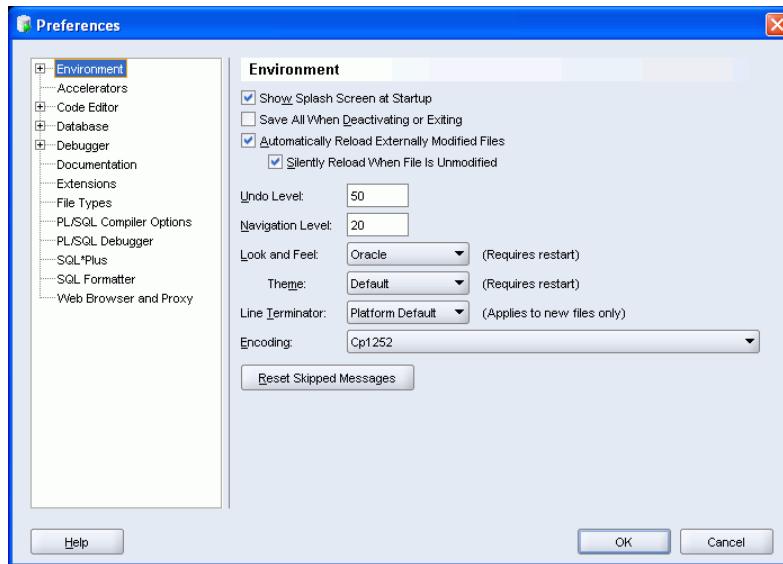
To enhance the productivity of SQL developers, SQL Developer has added quick links to popular search engines and discussion forums such as AskTom and Google. Also, shortcut icons to some of the frequently used tools, such as Notepad, Microsoft Word, and Dreamweaver, are available.

You can add external tools to the existing list or even delete shortcuts to tools that you do not use frequently. To do so, perform the following:

1. From the Tools menu, select **External Tools**.
2. In the External Tools dialog box, select **New** to add new tools. Select **Delete** to remove any tool from the list.

Setting Preferences

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



ORACLE

C - 27

Copyright © 2008, Oracle. All rights reserved.

Setting Preferences

You can customize many aspects of the SQL Developer interface and environment by modifying the SQL Developer preferences according to your preferences and needs. To modify the SQL Developer preferences, select Tools, and then Preferences.

The preferences are grouped in the following categories:

- Environment
- Accelerators (Keyboard shortcuts)
- Code Editors
- Database
- Debugger
- Documentation
- Extensions
- File Types
- Migration
- PL/SQL Compilers
- PL/SQL Debugger, and so on

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



Summary

SQL Developer is a free graphical tool to simplify the database development tasks. Using SQL Developer, you can browse, create, and edit database objects. You can use SQL worksheet to run SQL statements and scripts. SQL Developer enables you to create and save your own special set of reports for repeated use.

Using SQL*Plus

ORACLE

Copyright © 2008, Oracle. All rights reserved.

Objectives

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

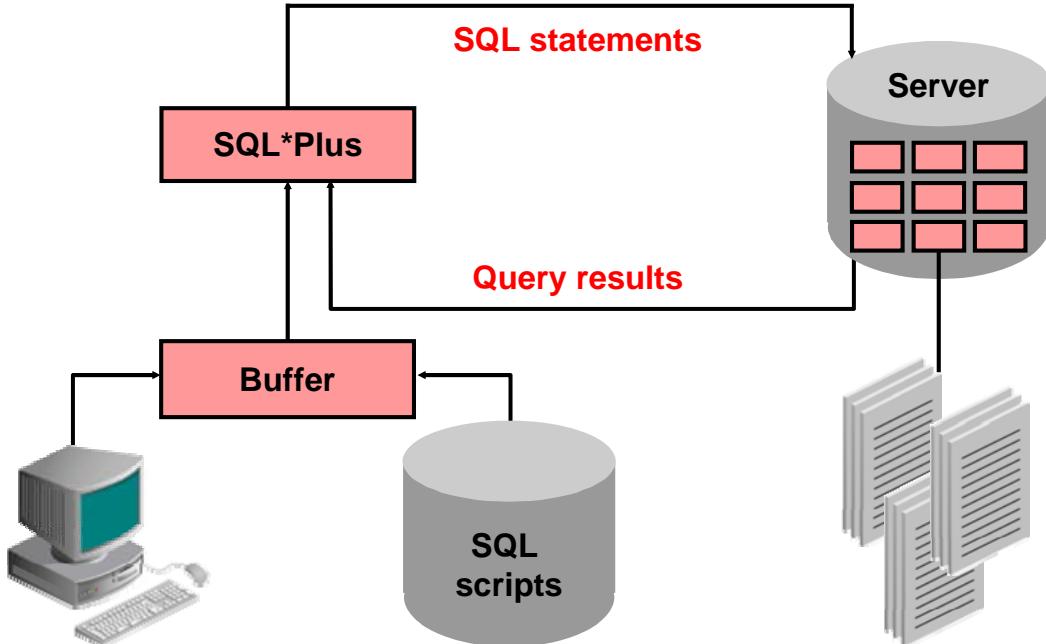
- Log in to SQL*Plus
- Edit SQL commands
- Format output using SQL*Plus commands
- Interact with script files



Objectives

You might want to create SELECT statements that can be used again and again. This appendix also covers the use of SQL*Plus commands to execute SQL statements. You learn how to format output using SQL*Plus commands, edit SQL commands, and save scripts in SQL*Plus.

SQL and SQL*Plus Interaction



D - 3

Copyright © 2008, Oracle. All rights reserved.

ORACLE

SQL and SQL*Plus Interaction

SQL is a command language for communication with the Oracle server from any tool or application. Oracle SQL contains many extensions. When you enter a SQL statement, it is stored in a part of memory called the *SQL buffer* and remains there until you enter a new SQL statement. SQL*Plus is an Oracle tool that recognizes and submits SQL statements to the Oracle database server for execution. It contains its own command language.

Features of SQL

- Can be used by a range of users, including those with little or no programming experience
- Is a nonprocedural language
- Reduces the amount of time required for creating and maintaining systems
- Is an English-like language

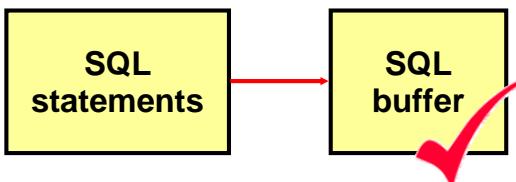
Features of SQL*Plus

- Accepts ad hoc entry of statements
- Accepts SQL input from files
- Provides a line editor for modifying SQL statements
- Controls environmental settings
- Formats query results into basic reports
- Accesses local and remote databases

SQL Statements Versus SQL*Plus Commands

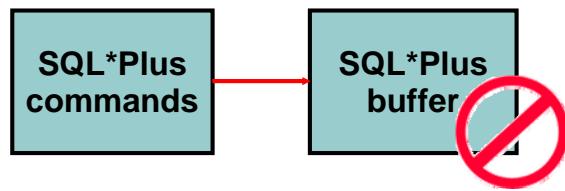
SQL

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



SQL*Plus

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



ORACLE

D - 4

Copyright © 2008, Oracle. All rights reserved.

SQL Statements Versus SQL*Plus Commands

The following table compares SQL and SQL*Plus:

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

SQL*Plus: Overview

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

ORACLE

D - 5

Copyright © 2008, Oracle. All rights reserved.

SQL*Plus: Overview

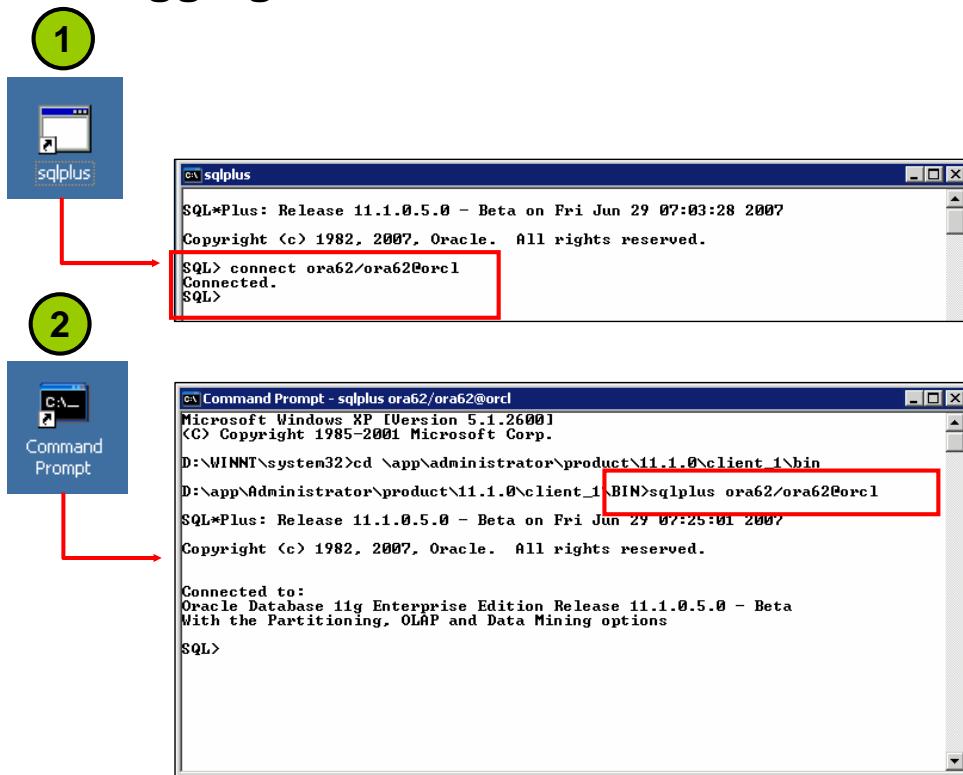
SQL*Plus is an environment in which you can perform the following:

- Execute SQL statements to retrieve, modify, add, and remove data from the database.
- Format, perform calculations on, store, and print query results in the form of reports.
- Create script files to store SQL statements for repeated use in the future.

SQL*Plus commands can be divided into the following main categories:

Category	Purpose
Environment	Affect the general behavior of SQL statements for the session
Format	Format query results
File manipulation	Save, load, and run script files
Entry	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: Available Methods



ORACLE

D - 6

Copyright © 2008, Oracle. All rights reserved.

Logging In to SQL*Plus: Available Methods

How you invoke SQL*Plus depends on which type of operating system or Windows environment you are running.

To log in from a Windows environment:

1. Select Start > Programs > Oracle > Application Development > SQL*Plus.
2. Enter the username, password, and database name.

To log in from a command-line environment:

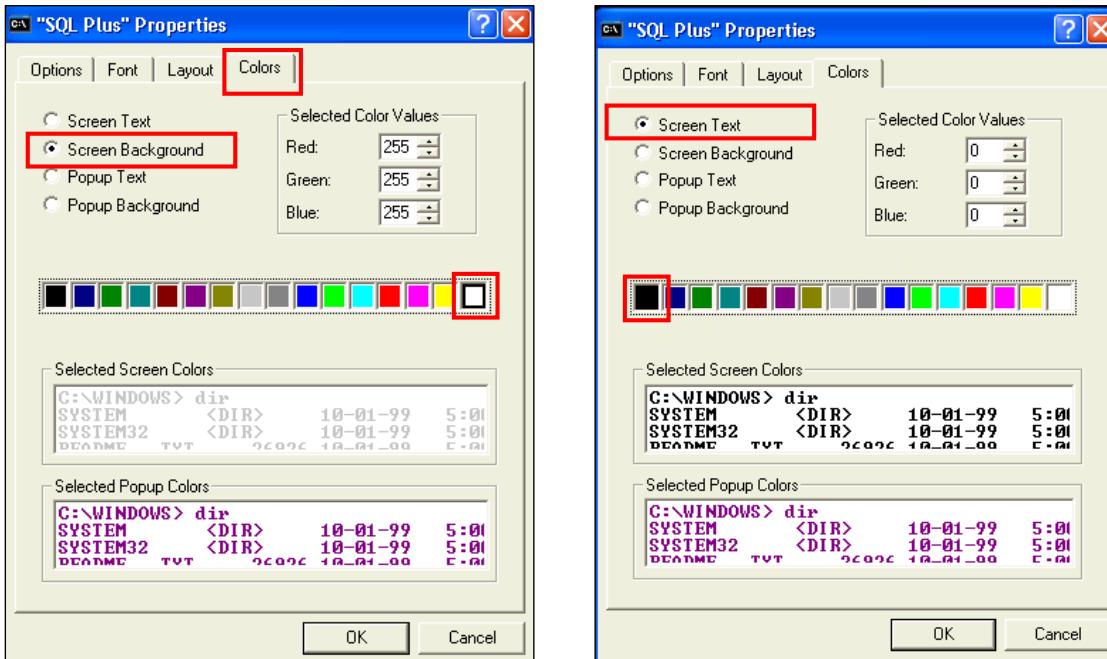
1. Log in to your machine.
2. Enter the `sqlplus` command shown in the slide.

In the syntax:

```
username Your database username  
password Your database password (Your password is visible if you enter it here.)  
@database The database connect string
```

Note: To ensure the integrity of your password, do not enter it at the operating system prompt. Instead, enter only your username. Enter your password at the password prompt.

Customizing the SQL*Plus Environment



ORACLE

D - 7

Copyright © 2008, Oracle. All rights reserved.

Customizing the SQL*Plus Environment

You can optionally change the look of the SQL*Plus environment by using the SQL*Plus Properties dialog box.

In the SQL*Plus window, right-click the title bar and select Properties from the shortcut menu. You can then use the colors tab of the SQL*Plus Properties dialog box to set Screen Background and Screen Text.

Displaying Table Structure

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

```
DESC[RIBE] tablename
```

ORACLE

D - 8

Copyright © 2008, Oracle. All rights reserved.

Displaying Table Structure

In SQL*Plus, you can display the structure of a table by using the DESCRIBE command. The result of the command is a display of column names and data types as well as an indication of whether a column must contain data.

In the syntax:

tablename Is the name of any existing table, view, or synonym that is accessible to the user

To describe the JOB_GRADES table, use this command:

```
SQL> DESCRIBE job_grades
      Name          Null?    Type
-----  -----
GRADE_LEVEL                      VARCHAR2( 3 )
LOWEST_SAL                       NUMBER
HIGHEST_SAL                      NUMBER
```

Displaying Table Structure

```
DESCRIBE departments
```

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

ORACLE

Displaying Table Structure (continued)

The example in the slide displays the information about the structure of the DEPARTMENTS table. In the result:

Null? : Specifies whether a column must contain data (NOT NULL indicates that a column must contain data.)

Type : Displays the data type for a column

The following table describes the data types:

Data Type	Description
NUMBER(<i>p</i> , <i>s</i>)	Number value that has a maximum number of digits <i>p</i> , which is the number of digits to the right of the decimal point <i>s</i>
VARCHAR2(<i>s</i>)	Variable-length character value of maximum size <i>s</i>
DATE	Date and time value between January 1, 4712 B.C., and A.D. December 31, 9999
CHAR(<i>s</i>)	Fixed-length character value of size <i>s</i>

SQL*Plus Editing Commands

- A[PPEND] *text*
- C[HANGE] / *old* / *new*
- C[HANGE] / *text* /
- CL[EAR] BUFF[ER]
- DEL
- DEL *n*
- DEL *m n*

ORACLE

D - 10

Copyright © 2008, Oracle. All rights reserved.

SQL*Plus Editing Commands

SQL*Plus commands are entered one line at a time and are not stored in the SQL buffer.

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

Guidelines

- If you press Enter before completing a command, SQL*Plus prompts you with a line number.
- You terminate the SQL buffer by either entering one of the terminator characters (semicolon or slash) or by pressing Enter twice. The SQL prompt then appears.

SQL*Plus Editing Commands

- `I[NPUT]`
- `I[NPUT] text`
- `L[IST]`
- `L[IST] n`
- `L[IST] m n`
- `R[UN]`
- `n`
- `n text`
- `0 text`

ORACLE

D - 11

Copyright © 2008, Oracle. All rights reserved.

SQL*Plus Editing Commands (continued)

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

Note: You can enter only one SQL*Plus command for each SQL prompt. SQL*Plus commands are not stored in the buffer. To continue a SQL*Plus command on the next line, end the first line with a hyphen (-).

Using LIST, n, and APPEND

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

```
1
1* SELECT last_name
```

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

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

ORACLE

D - 12

Copyright © 2008, Oracle. All rights reserved.

Using LIST, n, and APPEND

- Use the L[IST] command to display the contents of the SQL buffer. The asterisk (*) beside line 2 in the buffer indicates that line 2 is the current line. Your edits apply to the current line.
- Change the number of the current line by entering, at the SQL prompt, the number (*n*) of the line that you want to edit. The new current line is displayed.
- Use the A[PPEND] command to add text to the current line. The newly edited line is displayed. Verify the new contents of the buffer by using the LIST command.

Note: Many SQL*Plus commands, including LIST and APPEND, can be abbreviated to just their first letters. LIST can be abbreviated to L; APPEND can be abbreviated to A.

Using the CHANGE Command

```
LIST
```

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

```
c/employees/departments
```

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

```
LIST
```

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

ORACLE

Using the CHANGE Command

- Use L[IST] to display the contents of the buffer.
- Use the C[HANGE] command to alter the contents of the current line in the SQL buffer. In this case, replace the EMPLOYEES table with the DEPARTMENTS table. The new current line is displayed.
- Use the L[IST] command to verify the new contents of the buffer.

SQL*Plus File Commands

- `SAVE filename`
- `GET filename`
- `START filename`
- `@ filename`
- `EDIT filename`
- `SPOOL filename`
- `EXIT`

ORACLE

D - 14

Copyright © 2008, Oracle. All rights reserved.

SQL*Plus File Commands

SQL statements communicate with the Oracle server. SQL*Plus commands control the environment, format query results, and manage files. You can use the commands described in the following table:

Command	Description
<code>SAV[E] filename [.ext]</code> [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.
<code>GET filename [.ext]</code>	Writes the contents of a previously saved file to the SQL buffer. The default extension for the file name is .sql.
<code>STA[RT] filename [.ext]</code>	Runs a previously saved command file.
<code>@ filename</code>	Runs a previously saved command file (same as START).
<code>ED[IT]</code>	Invokes the editor and saves the buffer contents to a file named afiedt.buf.
<code>ED[IT] [filename[.ext]]</code>	Invokes the editor to edit the contents of a saved file.
<code>SPO[OL] [filename[.ext]]</code> <code>OFF OUT</code>	Stores query results in a file. OFF closes the spool file. OUT closes the spool file and sends the file results to the printer.
<code>EXIT</code>	Quits SQL*Plus.

Using the SAVE, START, and EDIT Commands

```
LIST
```

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

```
SAVE my_query
```

```
Created file my_query
```

```
START my_query
```

```
LAST_NAME
```

```
MANAGER_ID DEPARTMENT_ID
```

```
-----  
King
```

```
90
```

```
Kochhar
```

```
100
```

```
90
```

```
...
```

```
107 rows selected.
```

ORACLE

D - 15

Copyright © 2008, Oracle. All rights reserved.

Using the SAVE, START, and EDIT Commands

SAVE

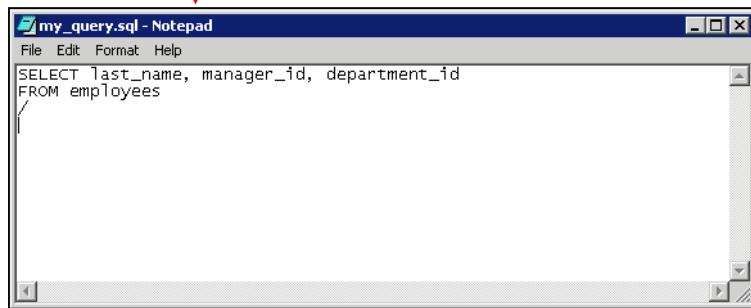
Use the SAVE command to store the current contents of the buffer in a file. In this way, you can store frequently used scripts for use in the future.

START

Use the START command to run a script in SQL*Plus.

Using the SAVE, START, and EDIT Commands

EDIT my_query



ORACLE

D - 16

Copyright © 2008, Oracle. All rights reserved.

Using the SAVE, START, and EDIT Commands (continued)

EDIT

Use the EDIT command to edit an existing script. This opens an editor with the script file in it. When you have made the changes, quit the editor to return to the SQL*Plus command line.

SQL*Plus Enhancements Since Oracle Database 10g

- Changes to the `SET SERVEROUT[PUT]` command
- White space support in file and path names in Windows
- Three new predefined SQL*Plus variables
- The new `RECYCLEBIN` clause of the `SHOW` command
- The new `APPEND`, `CREATE`, and `REPLACE` extensions to the `SPOOL` command
- New error messages for the `COPY` command
- Change in the `DESCRIBE` command behavior
- New `PAGESIZE` default
- New `SQLPLUS` program compatibility option
- Execution statistics information in the `AUTOTRACE` command report

ORACLE

D - 17

Copyright © 2008, Oracle. All rights reserved.

SQL*Plus Enhancements Since Oracle Database 10g

The next section highlights the enhancements to SQL*Plus that were introduced starting with Oracle Database 11g.

Changes to the SERVEROUTPUT Command

- Use the SET SERVEROUT[PUT] command to control whether to display the output of stored procedures or PL/SQL blocks in SQL*Plus.
- The DBMS_OUTPUT line length limit is increased from 255 bytes to 32,767 bytes.
- The default size is now unlimited.
- Resources are not preallocated when SERVEROUTPUT is set.
- Because there is no performance penalty, use UNLIMITED unless you want to conserve physical memory.

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



Changes to the SERVEROUTPUT Command

Most PL/SQL input and output for storing data in database tables, or for querying those tables, is achieved through SQL statements. All other PL/SQL I/O is done through APIs that interact with other programs. For example, the DBMS_OUTPUT package has procedures such as PUT_LINE. To see the result outside of PL/SQL requires another program, such as SQL*Plus, to read and display the data passed to DBMS_OUTPUT.

SQL*Plus does not display DBMS_OUTPUT data unless you first issue the SQL*Plus command SET SERVEROUTPUT ON as follows:

```
SET SERVEROUTPUT ON
```

Note

- SIZE sets the number of bytes of the output that can be buffered within the Oracle Database server. The default is UNLIMITED. *n* cannot be less than 2,000 or greater than 1,000,000.
- For additional information about SERVEROUTPUT, see the *Oracle Database PL/SQL User's Guide and Reference 11g Release 1 (11.1)*.

White Space Support in File and Path Names in Windows

- In Windows, white space can be included in file names and paths.
- Examples of where white space can be used:
 - START, @, @@, RUN, SPOOL, SAVE, and EDIT commands
- To reference files or paths containing spaces, enclose the name or path in double quotation marks.

Examples

```
SAVE "Monthly Report.sql"
START "Monthly Report.sql"
```



White Space Support in File and Path Names in Windows

You can use blank spaces in file names and paths, signified by the use of double quotation marks.

Predefined SQL*Plus Variables

Variable Name	Contains
_CONNECT_IDENTIFIER	Connection identifier used to make the connection, where available
_DATE	Current date or a user-defined fixed string
_EDITOR	Specifies the editor used by the EDIT command
_O_VERSION	Current version of the installed Oracle Database
O_RELEASE	Full release number of the installed Oracle Database
_PRIVILEGE	Privilege level of the current connection
_SQLPLUS_RELEASE	Full release number of the installed SQL*Plus component
_USER	Username used to make the connection



Predefined SQL*Plus Variables

There are eight variables defined during SQL*Plus installation. These variables differ from user-defined variables by having only predefined values.

You can view the value of each of these variables with the DEFINE command. These variables can be accessed and redefined like any other substitution variable. They can be used in TTITLE, in ' & ' substitution variables, or in your SQL*Plus command-line prompt.

You can use the DEFINE command to view the definitions of these eight predefined variables in the same way as you view the other DEFINE definitions. You can also use the DEFINE command to redefine their values, or you can use the UNDEFINE command to remove their definitions and make them unavailable.

Note: For additional information about the SQL*Plus predefined variables, see the *SQL*Plus User's Guide and Reference Release 11.1*.

Using the New Predefined SQL*Plus Variables: Examples

```
-- Change the SQL*Plus prompt to display the connection
-- identifier

SQL> SET SQLPROMPT '_CONNECT_IDENTIFIER > '
orcl >

-- view the predefined value of the _SQLPLUS_RELEASE
-- substitution variable

orcl > DEFINE _SQLPLUS_RELEASE
DEFINE _SQLPLUS_RELEASE = "1002000100" (CHAR)

-- View the user name connected to the current
-- connection.

orcl > DEFINE _USER
DEFINE _USER = "HR" (CHAR)
```



Using the New Predefined SQL*Plus Variables: Examples

To view all predefined and user-defined variable definitions, enter `DEFINE` at the SQL*Plus prompt. All predefined and all user-defined variable definitions are displayed as follows:

```
orcl > DEFINE
DEFINE _DATE = "06-JUL-06" (CHAR)
DEFINE _CONNECT_IDENTIFIER = "orcl" (CHAR)
DEFINE _USER = "HR" (CHAR)
DEFINE _PRIVILEGE = " " (CHAR)
DEFINE _SQLPLUS_RELEASE = "1002000100" (CHAR)
DEFINE _EDITOR = "Notepad" (CHAR)
DEFINE _O_VERSION = "Oracle Database 10g Enterprise Edition
Release 10.2.0.1.0 - Production
With the Partitioning, OLAP and Data Mining options" (CHAR)
DEFINE _O_RELEASE = "1002000100" (CHAR)
```

You can use `UNDEFINE` to remove a substitution variable definition and make it unavailable.

The SHOW Command and the New RECYCLEBIN Clause

```
SHOW RECYC[LEBIN] [original_name]
SELECT * FROM USER_RECYCLEBIN
desc user_recyclebin;
Name          Null?    Type
-----
OBJECT_NAME    NOT NULL VARCHAR2(30)
ORIGINAL_NAME           VARCHAR2(32)
OPERATION           VARCHAR2(9)
TYPE             VARCHAR2(25)
TS_NAME           VARCHAR2(30)
CREATETIME        VARCHAR2(19)
DROPTIME          VARCHAR2(19)
DROPSCN           NUMBER
PARTITION_NAME    VARCHAR2(32)
CAN_UNDROP        VARCHAR2(3)
CAN_PURGE         VARCHAR2(3)
RELATED           NOT NULL NUMBER
BASE_OBJECT        NOT NULL NUMBER
PURGE_OBJECT      NOT NULL NUMBER
SPACE             NUMBER
```

ORACLE

D - 22

Copyright © 2008, Oracle. All rights reserved.

The SHOW Command and the New RECYCLEBIN Clause

Using the SHOW command, you can show objects in the recycle bin that can be reverted with the FLASHBACK BEFORE DROP command. You do not need to remember column names, or interpret the less readable output from the query. The following query returns three columns that are displayed in the slide:

```
SELECT * FROM USER_RECYCLEBIN
```

The SHOW Command and the RECYCLEBIN Clause: Example

```
DROP TABLE test;  
Table dropped.
```

```
SHOW recyclebin
```

ORIGINAL NAME	RECYCLEBIN NAME	OBJECT TYPE	DROP TIME
TEST	BIN\$SefY+qPKSY6mvU8eDT1r+A==\$0	TABLE	2006-07-06 11:12:00

ORACLE

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 you specify
OFF	Stops spooling
OUT	Stops spooling and sends the file to your computer's standard (default) printer

ORACLE

Using the SQL*Plus SPOOL Command

The SPOOL command stores query results in a file, or optionally sends the file to a printer. The SPOOL command was enhanced. You can now append to, or replace an existing file, where previously you could use SPOOL only to create (and replace) a file. REPLACE is the default.

To spool the output generated by commands in a script without displaying the output on the screen, use SET TERMOUT OFF. SET TERMOUT OFF does not affect the output from commands that run interactively.

You must use quotation marks around file names containing white spaces. To create a valid HTML file using the SPOOL APPEND commands, you must use PROMPT or a similar command to create the HTML page header and footer. The SPOOL APPEND command does not parse HTML tags. Set SQLPLUSCOMPAT[IBILITY] to 9.2 or earlier to disable the CREATE, APPEND, and SAVE parameters.

Using the SQL*Plus SPOOL Command: Examples

```
-- Record the output in the new file DIARY using the  
-- default file extension.  
  
SPOOL DIARY CREATE  
  
-- Append the output to the existing file DIARY.  
  
SPOOL DIARY APPEND  
  
-- Record the output to the file DIARY, overwriting the  
-- existing content  
  
SPOOL DIARY REPLACE  
  
-- Stop spooling and print the file on your default printer.  
  
SPOOL OUT
```



The COPY Command: New Error Messages

```
CPY-0002 Illegal or missing APPEND, CREATE, INSERT, or  
REPLACE option

CPY-0003 Internal Error: logical host number out of  
Range

CPY-0004 Source and destination table and column names  
don't match

CPY-0005 Source and destination column attributes don't  
Match

CPY-0006 Select list has more columns than destination  
Table

CPY-0007 Select list has fewer columns than destination  
table
```



The COPY Command: New Error Messages

CPY-0002 Illegal or missing APPEND, CREATE, INSERT, or REPLACE option: An internal COPY function has invoked COPY with a create option (flag) value that is out of range.

CPY-0003 Internal Error: Logical host number out of range: An internal COPY function has been invoked with a logical host number value that is out of range.

CPY-0004 Source and destination table and column names don't match: On an APPEND operation or an INSERT (when the table exists), at least one column name in the destination table does not match the corresponding column name in the optional column name list or in the SELECT command. To correct this, respecify the COPY command, making sure that the column names and their respective order in the destination table match the column names and column order in the optional column list or in the SELECT command.

CPY-0005 Source and destination column attributes don't match: On an APPEND operation or an INSERT (when the table exists), at least one column in the destination table does not have the same data type as the corresponding column in the SELECT command. To correct this, respecify the COPY command, making sure that the data types for the items being selected agree with the destination. Use TO_DATE, TO_CHAR, and TO_NUMBER to make conversions.

The COPY Command: New Error Messages (continued)

CPY-0006 Select list has more columns than destination table: On an APPEND operation or an INSERT (when the table exists), the number of columns in the SELECT command is greater than the number of columns in the destination table. To correct this, respecify the COPY command, making sure that the number of columns being selected agrees with the number in the destination table.

CPY-0007 Select list has fewer columns than destination table: On an APPEND operation or INSERT (when the table exists), the number of columns in the SELECT command is less than the number of columns in the destination table. To correct this, respecify the COPY command, making sure that the number of columns being selected agrees with the number in the destination table.

CPY-0008 More column list names than columns in the destination table: On an APPEND operation or an INSERT (when the table exists), the number of columns in the column name list is greater than the number of columns in the destination table. To correct this, respecify the COPY command, making sure that the number of columns in the column list agrees with the number in the destination table.

CPY-0009 Fewer column list names than columns in the destination table: On an APPEND operation or an INSERT (when the table exists), the number of columns in the column name list is less than the number of columns in the destination table. To correct this, respecify the COPY command, making sure that the number of columns in the column list agrees with the number in the destination table.

CPY-0012 Datatype cannot be copied: An attempt was made to copy a data type that is not supported in the COPY command. Data types supported by the COPY command are CHAR, DATE, LONG, NUMBER, and VARCHAR2. To correct this, respecify the COPY command, making sure that the unsupported data type column is removed.

The COPY Command: New Error Messages

CPY-0008 More column list names than columns in the destination table

CPY-0009 Fewer column list names than columns in the destination table

CPY-0012 Datatype cannot be copied



Change in the DESCRIBE Command Behavior

- Prior to Oracle Database 10g, using DESCRIBE on an invalidated object failed with the error:
 - ORA-24372: invalid object for describe
- The DESCRIBE command continued to fail even if the object had since been validated.
- Starting with Oracle Database 10g, the DESCRIBE command now automatically validates the object and continues if the validation is successful.



The SET PAGESIZE Command

- It sets the number of rows on each page of the output in SQL*Plus.
- The default PAGESIZE has changed from 24 to 14.
- You can set PAGESIZE to zero to suppress all headings, page breaks, titles, the initial blank line, and other formatting information.

```
SET PAGESIZE {14 | n}
```



The SET PAGESIZE Command

The SET PAGESIZE command sets the number of rows displayed on each page. Error and informational messages are not counted in the page size, so pages may not always be exactly the same length. The default page size for SQL*Plus changed from 24 to 14.

The SQLPLUS Program and the Compatibility Option

Sets the value of the SQLPLUSCOMPATIBILITY system variable to the SQL*Plus release specified by x.y[.z]

```
SQLPLUS -C[OMPATIBILITY] {x.y[.z]}
```

```
-- x is the version number  
-- y is the release number  
-- z is the update number
```

```
SQLPLUS -C 10.2.0
```



The SQLPLUS Program and the Compatibility Option

The SQL*Plus Compatibility Matrix tabulates the behavior resulting from each SQL*Plus compatibility setting. SQL*Plus compatibility modes can be set in three ways:

- You can include a SET SQLPLUSCOMPATIBILITY command in your site or user profile. On installation, there is no SET SQLPLUSCOMPATIBILITY setting in `glogin.sql`. Therefore, the default compatibility is 10.2.
- You can use the SQLPLUS -C[OMPATIBILITY] {x.y[.z]} command argument at startup to set the compatibility mode of that session.
- You can use the SET SQLPLUSCOMPATIBILITY {x.y[.z]} command during a session to set the SQL*Plus behavior you want for that session.

Note: For a list showing the release of SQL*Plus that introduced the behavior change, see the “SQL*Plus Compatibility Matrix” topic in *SQL*Plus User’s Guide and Reference Release 11.1*.

Using the AUTOTRACE Command

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

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

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



Using the AUTOTRACE Command

EXPLAIN shows the query execution path by performing an EXPLAIN PLAN. STATISTICS displays the SQL statement statistics. The formatting of your AUTOTRACE report may vary depending on the version of the server to which you are connected and the configuration of the server. The additional information and tabular output of AUTOTRACE PLAN is supported when connecting to Oracle Database 10g (Release 10.1) or later. When you connect to an earlier database, the older form of AUTOTRACE reporting is used.

The DBMS_XPLAN package provides an easy way to display the output of the EXPLAIN PLAN command in several, predefined formats.

Notes

- For additional information about the package and subprograms, see the *Oracle Database PL/SQL Packages and Types Reference 10g Release 2 (10.2)* guide or the subsequent versions of this guide.
- For additional information about EXPLAIN PLAN, see *Oracle Database SQL Reference 10g Release 2 (10.2)* or the subsequent versions of this reference.
- For additional information about Execution Plans and the statistics, see *Oracle Database Performance Tuning Guide 10g Release 2 (10.2)* or the subsequent versions of this guide.

Displaying a Plan Table Using the DBMS_XPLAN.DISPLAY Package Function

```
-- Execute an explain plan command on a SELECT
-- statement

EXPLAIN PLAN FOR
SELECT * FROM emp e, dept d
  WHERE e.deptno = d.deptno
    AND e.ename='benoit';

-- Display the plan using the DBMS_XPLAN.DISPLAY table
-- function

SET LINESIZE 130
SET PAGESIZE 0
SELECT * FROM table(DBMS_XPLAN.DISPLAY);
```

ORACLE

D - 33

Copyright © 2008, Oracle. All rights reserved.

Displaying a Plan Table Using the DBMS_XPLAN.DISPLAY Package Function

The query shown in the slide produces the following output:

```
Plan hash value: 3693697075
-----
| Id  | Operation          | Name | Rows | Bytes | Cost (%CPU)| Time      |
| 0   | SELECT STATEMENT   |       | 1    | 57   | 6   (34) | 00:00:01 |
| * 1 | HASH JOIN          |       | 1    | 57   | 6   (34) | 00:00:01 |
| * 2 | TABLE ACCESS FULL | EMP  | 1    | 37   | 3   (34) | 00:00:01 |
| 3   | TABLE ACCESS FULL | DEPT | 4    | 80   | 3   (34) | 00:00:01 |

-----
Predicate Information (identified by operation id):
-----
1 - access("E"."DEPTNO"="D"."DEPTNO")
2 - filter("E"."ENAME"='benoit')

15 rows selected.
```

Summary

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

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



Summary

SQL*Plus is an execution environment that you can use to send SQL commands to the database server and to edit and save SQL commands. You can execute commands from the SQL prompt or from a script file.

JDeveloper

Review of JDeveloper

ORACLE

Copyright © 2008, Oracle. All rights reserved.

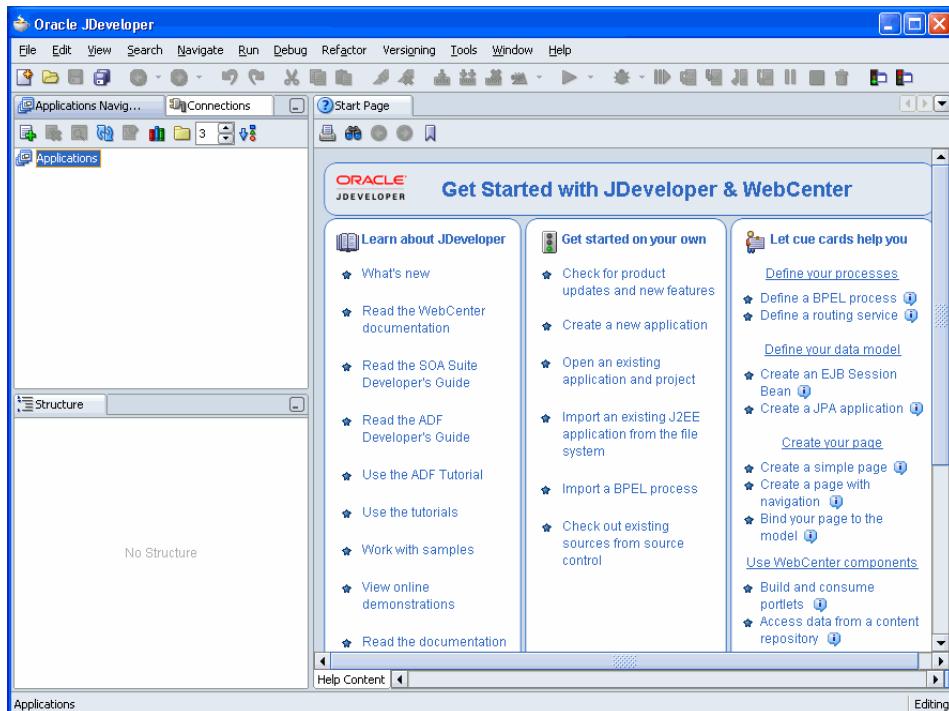
Objectives

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

- Log in to JDeveloper
- Create, compile, and run program units
- Debug PL/SQL programs
- Examine and modify variables



JDeveloper



ORACLE

E - 3

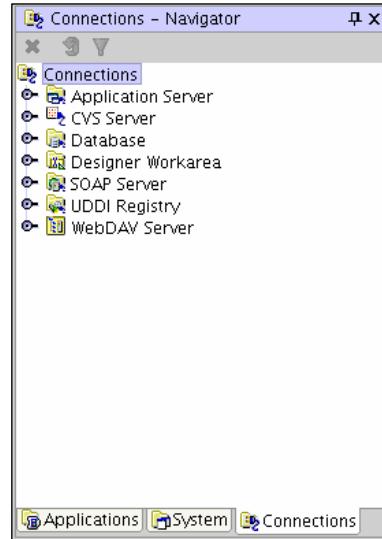
Copyright © 2008, Oracle. All rights reserved.

JDeveloper

Oracle JDeveloper 11g is an integrated development environment (IDE) for developing and deploying Java applications and Web services. It supports every stage of the software development life cycle (SDLC), from modeling through deploying. It has the features to use the latest industry standards for Java, Extensible Markup Language (XML), and SQL while developing an application.

Oracle JDeveloper 11g initiates a new approach to Java 2 Platform, Enterprise Edition (J2EE) development with features that enable visual and declarative development. This innovative approach makes J2EE development simple and efficient.

Connections Navigator

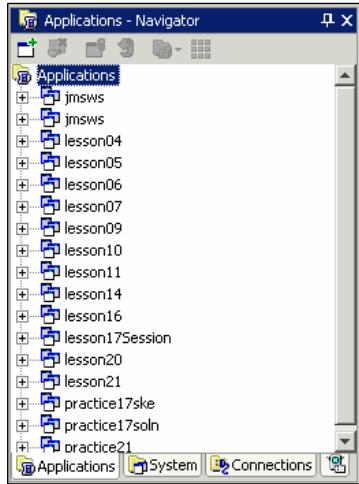


ORACLE

Connections Navigator

Using Oracle JDeveloper 11g, you can store the information necessary to connect to a database in an object called “connection.” A connection is stored as part of the IDE settings, and can be exported and imported for easy sharing among groups of users. A connection serves several purposes: from browsing the database and building applications, all the way to deployment.

Applications Navigator

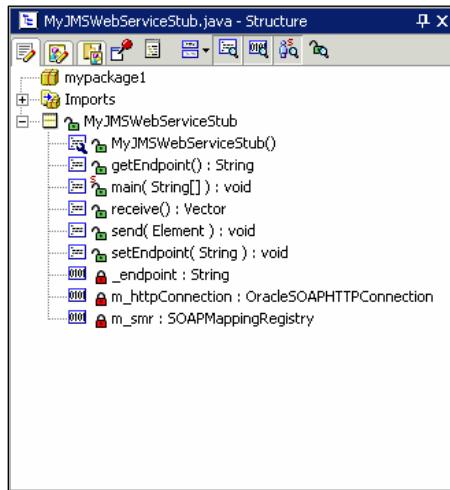


ORACLE

Applications Navigator

The Applications navigator gives you a logical view of your application and the data it contains. The Applications navigator provides an infrastructure that the different extensions can plug into and use to organize their data and menus in a consistent, abstract manner. While the Applications navigator can contain individual files (such as Java source files), it is designed to consolidate complex data. Complex data types such as entity objects, Unified Modeling Language (UML) diagrams, Enterprise JavaBeans (EJB), or Web services appear in this navigator as single nodes. The raw files that make up these abstract nodes appear in the Structure window.

Structure Window



ORACLE

E - 6

Copyright © 2008, Oracle. All rights reserved.

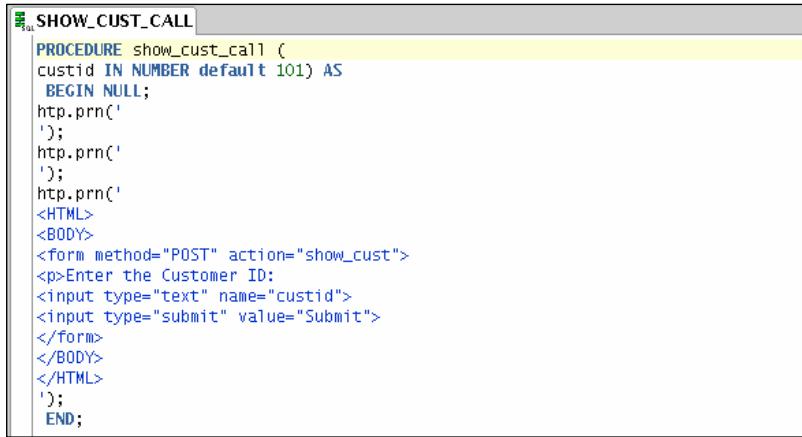
Structure Window

The Structure window offers a structural view of the data in the document that is currently selected in the active window of those windows that participate in providing structure: the navigators, the editors and viewers, and the Property Inspector.

In the Structure window, you can view the document data in many ways. The structures available for display are based on the document type. For a Java file, you can view code structure, user interface (UI) structure, or UI model data. For an XML file, you can view XML structure, design structure, or UI model data.

The Structure window is dynamic, always tracking the current selection of the active window (unless you freeze the window's contents on a particular view), as is pertinent to the currently active editor. When the current selection is a node in the navigator, the default editor is assumed. To change the view of the structure for the current selection, click a different structure tab.

Editor Window



```
SHOW_CUST_CALL
PROCEDURE show_cust_call (
  custid IN NUMBER default 101) AS
BEGIN NULL;
  htp.prn('
');
  htp.prn('
');
  htp.prn('
');
  htp.prn('
<HTML>
<BODY>
<form method="POST" action="show_cust">
<p>Enter the Customer ID:</p>
<input type="text" name="custid">
<input type="submit" value="Submit">
</form>
</BODY>
</HTML>
');
END;
```

ORACLE

Editor Window

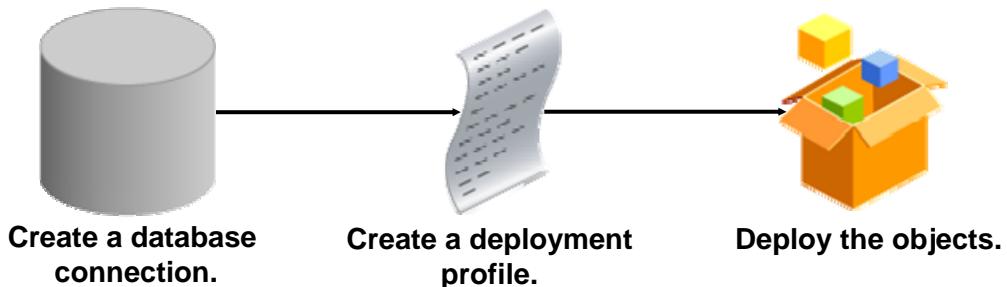
You can view all your project files in a single editor window, you can open multiple views of the same file, or you can open multiple views of different files.

The tabs at the top of the editor window are the document tabs. Clicking a document tab gives that file focus, bringing it to the foreground of the window in the current editor.

The tabs at the bottom of the editor window for a given file are the editor tabs. Selecting an editor tab opens the file in that editor.

Deploying Java Stored Procedures

Before deploying the Java stored procedures, perform the following steps:



ORACLE

Deploying Java Stored Procedures

Create a deployment profile for the Java stored procedures, deploy the classes and then, optionally, any public static methods in JDeveloper by using the settings in the profile.

Deploying to the database uses the information provided in the Deployment Profile Wizard and two Oracle Database utilities:

- `loadjava` loads the Java class containing the stored procedures to an Oracle database.
- `publish` generates the PL/SQL call-specific wrappers for the loaded public static methods. Publishing enables the Java methods to be called as PL/SQL functions or procedures.

Publishing Java to PL/SQL

The screenshot shows two windows from Oracle SQL Developer. The top window displays the Java code for `FormatCreditCardNo.java`, which contains a static method `formatCard` that formats a credit card number by inserting spaces every four digits. The bottom window shows the publication of this Java code as a PL/SQL procedure named `ccformat`.

```
FormatCreditCardNo.java [CCFORMAT]
public class FormatCreditCardNo
{
    public static final void formatCard(String[] cardno)
    {
        int count=0, space=0;
        String oldcc=cardno[0];
        // System.out.println("Printing the card no initially "+oldcc);
        String[] newcc= {" "};
        while (count<16)
        {
            newcc[0] += oldcc.charAt(count);
            space++;
            if (space ==4)
            {   newcc[0] += " "; space=0; }
            count++;
        }
        cardno[0]=newcc[0];
    }
}
```

```
FormatCreditCardNo.java [CCFORMAT]
PROCEDURE ccformat (x IN OUT varchar2)
AS LANGUAGE JAVA
NAME 'FormatCreditCardNo.formatCard(java.lang.String[])';
```

ORACLE

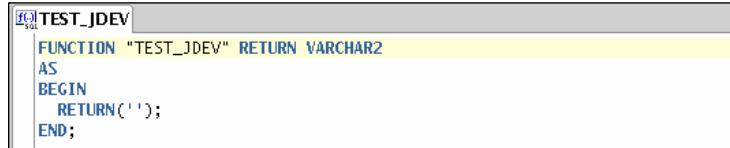
E - 9

Copyright © 2008, Oracle. All rights reserved.

Publishing Java to PL/SQL

The slide shows the Java code and how to publish the Java code in a PL/SQL procedure.

Creating Program Units



```
TEST_JDEV
FUNCTION "TEST_JDEV" RETURN VARCHAR2
AS
BEGIN
  RETURN(' ');
END;
```

Skeleton of the function

ORACLE

E - 10

Copyright © 2008, Oracle. All rights reserved.

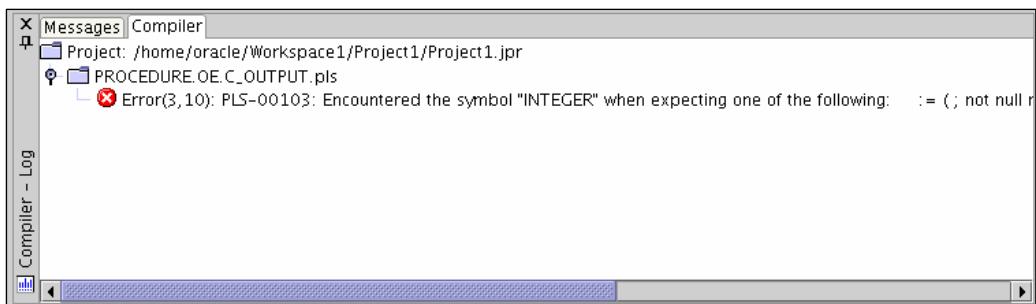
Creating Program Units

To create a PL/SQL program unit:

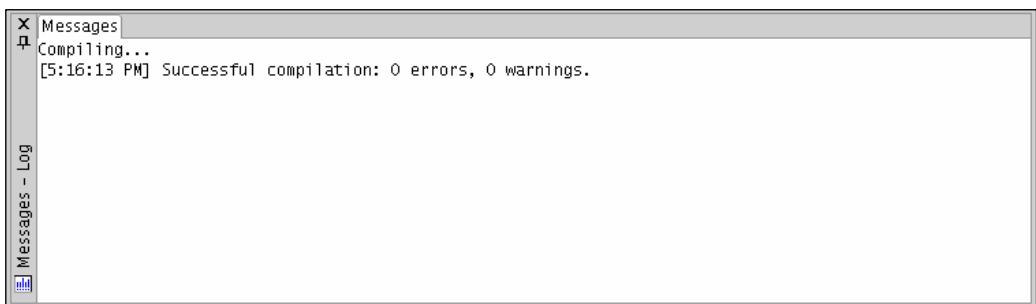
1. Select View > Connections navigator.
2. Expand Database and select a database connection.
3. In the connection, expand a schema.
4. Right-click a folder corresponding to the object type (Procedures, Packages, and Functions).
5. Choose New PL/SQL object_type. The Create PL/SQL dialog box appears for the function, package, or procedure.
6. Enter a valid name for the function, package, or procedure, and click OK.

A skeleton definition is created and opened in the Code Editor. You can then edit the subprogram to suit your need.

Compiling



Compilation with errors



Compilation without errors

ORACLE

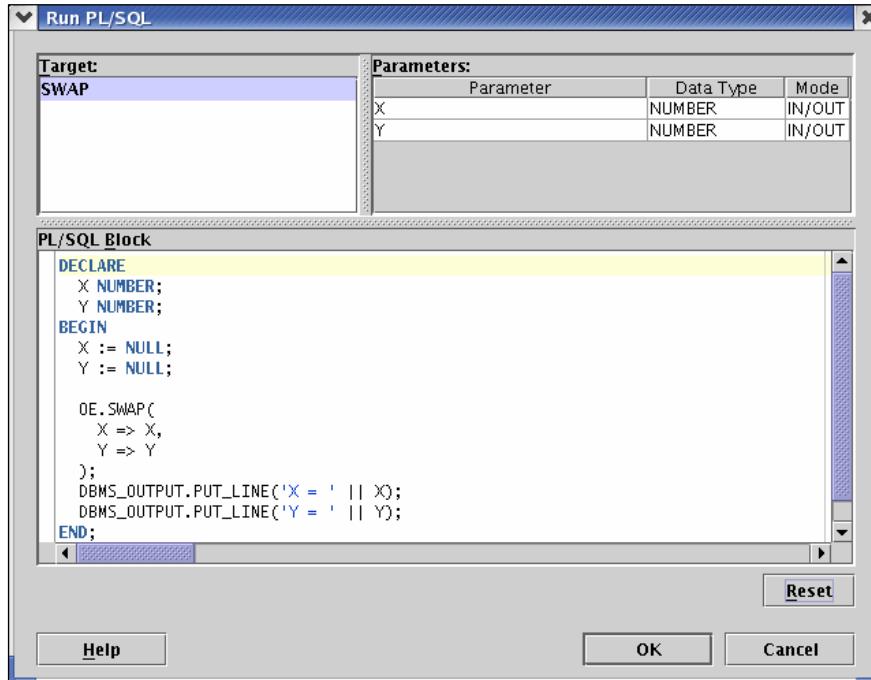
E - 11

Copyright © 2008, Oracle. All rights reserved.

Compiling

After editing the skeleton definition, you need to compile the program unit. Right-click the PL/SQL object to be compiled in the Connections navigator, and select Compile. Alternatively, you can press CTRL + SHIFT + F9 to compile.

Running a Program Unit



ORACLE

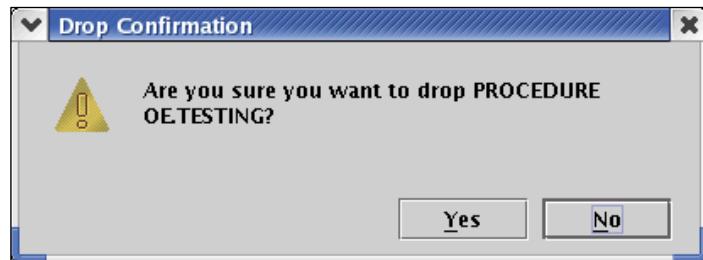
E - 12

Copyright © 2008, Oracle. All rights reserved.

Running a Program Unit

To execute the program unit, right-click the object and select Run. The Run PL/SQL dialog box appears. You may need to change the NULL values with reasonable values that are passed into the program unit. After you change the values, click OK. The output is displayed in the Message-Log window.

Dropping a Program Unit



ORACLE

E - 13

Copyright © 2008, Oracle. All rights reserved.

Dropping a Program Unit

To drop a program unit, right-click the object and select Drop. The Drop Confirmation dialog box appears; click Yes. The object is dropped from the database.

Debugging PL/SQL Programs

- JDeveloper support two types of debugging:
 - Local
 - Remote
- You need the following privileges to perform PL/SQL debugging:
 - DEBUG ANY PROCEDURE
 - DEBUG CONNECT SESSION



Debugging PL/SQL Programs

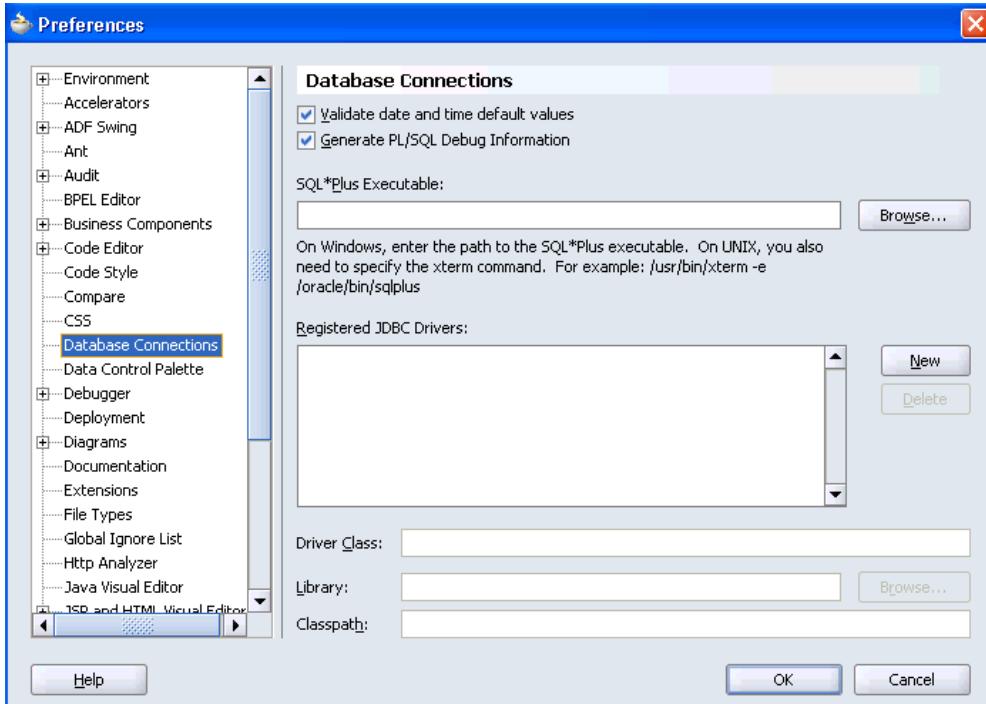
JDeveloper offers both local and remote debugging. A local debugging session is started by setting breakpoints in source files, and then starting the debugger. Remote debugging requires two JDeveloper processes: a debugger and a debuggee, which may reside on a different platform.

To debug a PL/SQL program, you must compile it in the INTERPRETED mode. You cannot debug a PL/SQL program that is compiled in the NATIVE mode. This mode is set in the database's `init.ora` file.

PL/SQL programs must be compiled with the DEBUG option enabled. This option can be enabled in various ways. Using SQL*Plus, execute `ALTER SESSION SET PLSQL_DEBUG = true` to enable the DEBUG option. Then you can create or recompile the PL/SQL program that you want to debug. Another way of enabling the DEBUG option is by using the following command in SQL*Plus:

```
ALTER <procedure, function, package> <name> COMPILE DEBUG;
```

Debugging PL/SQL Programs



ORACLE

Debugging PL/SQL Programs (continued)

Before you start debugging, make sure that the Generate PL/SQL Debug Information check box is selected. You can access the dialog box by using Tools > Preferences > Database Connections.

Instead of manually testing PL/SQL functions and procedures as you may be accustomed to doing from within SQL*Plus or by running a dummy procedure in the database, JDeveloper enables you to test these objects in an automatic way. With Oracle JDeveloper 11g, you can run and debug PL/SQL program units. For example, you can specify the parameters being passed or the values returned from a function, thereby giving you more control over what is run and providing you output details about what was tested.

Note: The procedures or functions in the Oracle database can be either stand-alone or within a package.

Debugging PL/SQL Programs (continued)

To run or debug functions, procedures, or packages, perform the following steps:

1. Create a database connection by using the Database Wizard.
2. In the navigator, expand the Database node to display the specific database username and schema name.
3. Expand the Schema node.
4. Expand the appropriate node depending on what you are debugging: Procedure, Function, or Package body.
5. (Optional for debugging only) Select the function, procedure, or package that you want to debug and double-click to open it in the Code Editor.
6. (Optional for debugging only) Set a breakpoint in your PL/SQL code by clicking to the left of the margin.

Note: The breakpoint must be set on an executable line of code. If the debugger does not stop, the breakpoint may have not been set on an executable line of code (ensure that the breakpoint is verified). Also, verify that the debugging PL/SQL prerequisites are met. In particular, make sure that the PL/SQL program is compiled in the INTERPRETED mode.

7. Make sure that either the Code Editor or the procedure in the navigator is currently selected.
8. Click the Debug button on the toolbar; or, if you want to run without debugging, click the Run button on the toolbar.
9. The Run PL/SQL dialog box appears.

- Select a target that is the name of the procedure or function that you want to debug. Note that the content in the Parameters and PL/SQL Block boxes changes dynamically when the target changes.

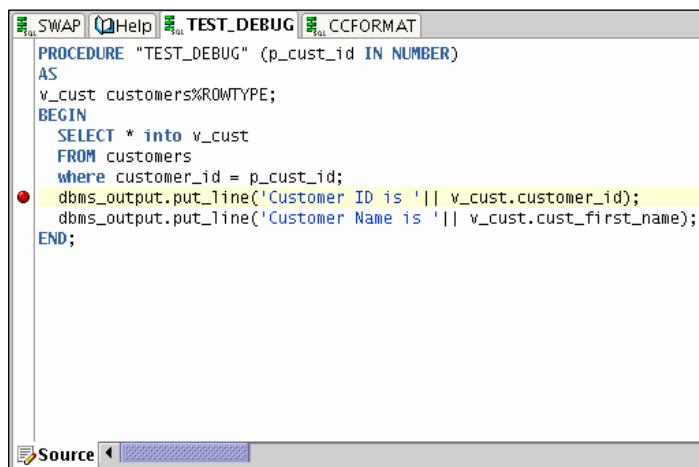
Note: You will have a choice of target only if you choose to run or debug a package that contains multiple program units.

- The Parameters box lists the target's arguments (if applicable).
- The PL/SQL Block box displays the code that was custom-generated by JDeveloper for the selected target. Depending on what the function or procedure does, you may need to replace the NULL values with reasonable values so that they are passed into the procedure, function, or package. In some cases, you may need to write additional code to initialize the values to be passed as arguments. In this case, you can edit the PL/SQL block text as necessary.

10. Click OK to execute or debug the target.
11. Analyze the output information that is displayed in the Log window.

In the case of functions, the return value is displayed. DBMS_OUTPUT messages are also displayed.

Setting Breakpoints



The screenshot shows the Oracle SQL Developer interface. The title bar reads "SWAP | Help | TEST_DEBUG | CCFORMAT". The main window displays a PL/SQL procedure named "TEST_DEBUG". A red circular breakpoint icon is positioned to the left of the second line of code, which is a SELECT statement. The code is as follows:

```
PROCEDURE "TEST_DEBUG" (p_cust_id IN NUMBER)
AS
v_cust customers%ROWTYPE;
BEGIN
  SELECT * into v_cust
  FROM customers
  where customer_id = p_cust_id;
  dbms_output.put_line('Customer ID is'|| v_cust.customer_id);
  dbms_output.put_line('Customer Name is'|| v_cust.cust_first_name);
END;
```

The bottom of the window shows tabs for "Source" and "Script".

ORACLE

E - 17

Copyright © 2008, Oracle. All rights reserved.

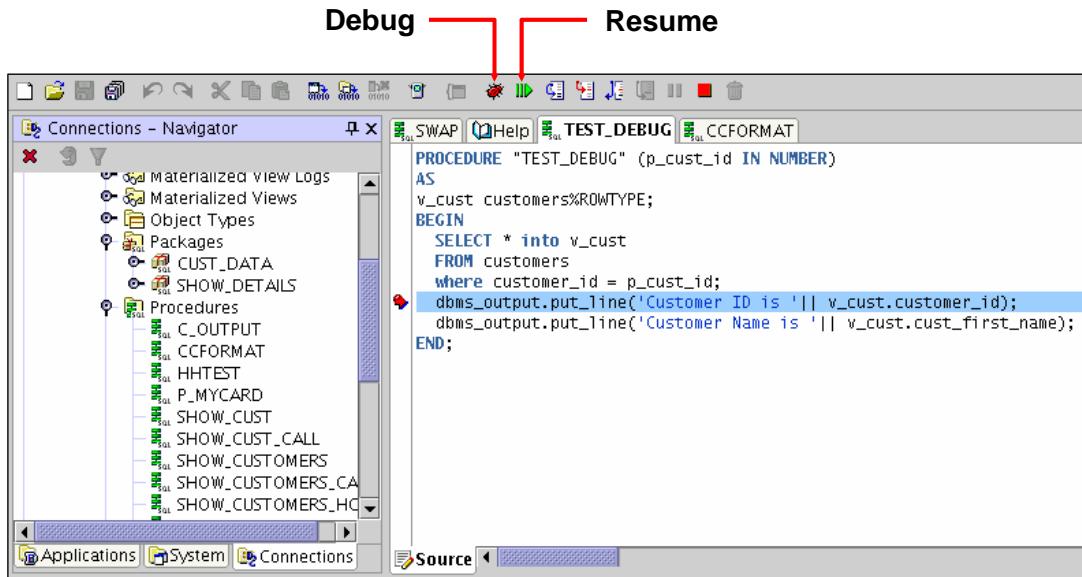
Setting Breakpoints

You can use breakpoints to examine the values of the variables in your program. A breakpoint is a trigger in a program that, when reached, pauses program execution thereby allowing you to examine the values of some or all program variables. By setting breakpoints in potential problem areas of your source code, you can run your program until its execution reaches a location you want to debug.

When your program execution encounters a breakpoint, it pauses, and the debugger displays the line containing the breakpoint in the Code Editor. You can then use the debugger to view the state of your program. Breakpoints are flexible in that they can be set before you begin a program run or at any time while you are debugging.

To set a breakpoint in the Code Editor, click the margin on the left, next to a line of executable code. Breakpoints set on comment lines, blank lines, declaration, and any other nonexecutable lines of code are not verified by the debugger and are treated as invalid.

Stepping Through Code



ORACLE

Stepping Through Code

After setting the breakpoint, start the debugger by clicking the Debug icon. The debugger pauses the program execution at the point where the breakpoint is set. At this point, you can check the values of the variables. You can continue with the program execution by clicking the Resume icon. The debugger then moves on to the next breakpoint. After executing all breakpoints, the debugger stops the execution of the program and displays the results in the Debugging – Log area.

Examining and Modifying Variables

Data		
Name	Value	Type
• P_CUST_ID	103	NUMBER
• V_CUST		Rowtype

Data window

ORACLE

E - 19

Copyright © 2008, Oracle. All rights reserved.

Examining and Modifying Variables

When the debugger is ON, you can examine and modify the values of variables in the Data, Smart Data, and Watches windows. You can modify program data values during a debugging session as a way to test hypothetical bug fixes during a program run. If you find that a modification fixes a program error, you can exit the debugging session, fix your program code accordingly, and recompile the program to make the fix permanent.

You use the Data window to display information about the variables in your program. The Data window displays the arguments, local variables, and static fields for the current context, which is controlled by the selection in the Stack window. If you move to a new context, the Data window is updated to show the data for the new context. If the current program was compiled without debug information, you cannot see the local variables.

Examining and Modifying Variables

Name	Value	Type
v_cust		Rowtype
v_cust.customer_id	103	NUMBER(6,0)

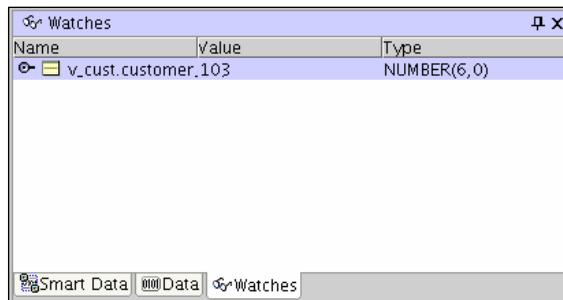
Smart Data window

ORACLE

Examining and Modifying Variables (continued)

Unlike the Data window that displays all variables in your program, the Smart Data window displays only the data that is relevant to the source code that you are stepping through.

Examining and Modifying Variables



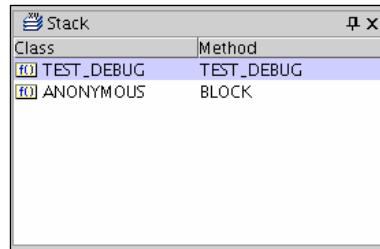
Watches window

ORACLE

Examining and Modifying Variables (continued)

A watch enables you to monitor the changing values of variables or expressions as your program runs. After you enter a watch expression, the Watch window displays the current value of the expression. As your program runs, the value of the watch changes as your program updates the values of the variables in the watch expression.

Examining and Modifying Variables



Stack window

ORACLE

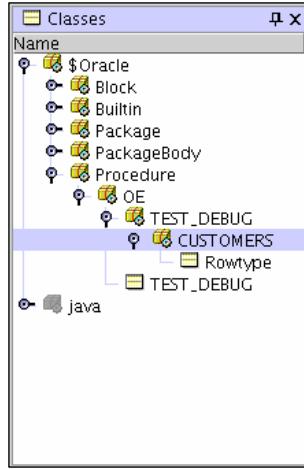
E - 22

Copyright © 2008, Oracle. All rights reserved.

Examining and Modifying Variables (continued)

You can activate the Stack window by using View > Debugger > Stack. It displays the call stack for the current thread. When you select a line in the Stack window, the Data window, Watch window, and all other windows are updated to show data for the selected class.

Examining and Modifying Variables



Classes window

ORACLE

Examining and Modifying Variables (continued)

The Classes window displays all classes that are currently being loaded to execute the program. If used with Oracle Java Virtual Machine (OJVM), it also shows the number of instances of a class and the memory used by those instances.

Summary

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

- Create, compile, and run program units
- Debug PL/SQL programs
- Examine and modify variables

