

Oracle Database 11g: Develop PL/SQL Program Units

Volume 1 • Student Guide

D49986GC20

Edition 2.0

September 2009

D63065

ORACLE®

Author

Lauran Serhal

**Technical Contributors
and Reviewers**

Anjulaponni Azhagulekshmi
Christian Bauwens
Christoph Burandt
Zarko Cesljas
Yanti Chang
Salome Clement
Laszlo Czinkoczki
Ingrid DelaHaye
Steve Friedberg
Laura Garza
Joel Goodman
Nancy Greenberg
Manish Pawar
Brian Pottle
Helen Robertson
Tulika Srivastava
Ted Witiuk

Editors

Arijit Ghosh
Raj Kumar

Publishers

Pavithran Adka
Sheryl Domingue

Copyright © 2009, Oracle. All rights reserved.

Disclaimer

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

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

Restricted Rights Notice

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

U.S. GOVERNMENT RIGHTS

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

Trademark Notice

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

I

Introduction

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

Lesson Objectives

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

- Discuss the goals of the course
- Identify the available 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



Lesson Aim

PL/SQL supports many program constructs. In this lesson, you review program units in the form of anonymous blocks, and you are introduced to named PL/SQL blocks. Named PL/SQL blocks are also referred to as subprograms. Named PL/SQL blocks include procedures and functions.

The tables from the Human Resources (HR) schema (which is used for the practices in this course) are briefly discussed. The development tools for writing, testing, and debugging PL/SQL are listed.

Lesson Agenda

- Course objectives and course agenda
- The schema and appendixes used in this course and the available PL/SQL development environments in this course
- Oracle 11g documentation and additional resources

ORACLE®

Course Objectives

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

- Create, execute, and maintain:
 - Procedures and functions with OUT parameters
 - Package constructs
 - Database triggers
- Manage PL/SQL subprograms and triggers
- Use a subset of Oracle-supplied packages to generate screen and file output
- Identify various techniques that impact your PL/SQL code design considerations
- Use the PL/SQL compiler, manage PL/SQL code, and manage dependencies

ORACLE®

I - 4

Copyright © 2009, Oracle. All rights reserved.

Course Objectives

You can develop modularized applications with database procedures by using database objects such as the following:

- Procedures and functions
- Packages
- Database triggers

Modular applications improve:

- Functionality
- Security
- Overall performance

Suggested Course Agenda

Day 1:

- Lesson 1: Introduction
- Lesson 1: Creating Procedures
- Lesson 2: Creating Functions
- Lesson 3: Creating Packages
- Lesson 4: Working with Packages

Day 2:

- Lesson 5: Using Oracle-Supplied Packages in Application Development
- Lesson 6: Using Dynamic SQL
- Lesson 7: Design Considerations for PL/SQL Code
- Lesson 8: Creating Triggers

ORACLE®

Suggested Course Agenda

Day 3:

- Lesson 9: Creating Compound, DDL, and Event Database Triggers
- Lesson 10: Using the PL/SQL Compiler
- Lesson 11: Managing PL/SQL Code
- Lesson 12: Managing Dependencies

ORACLE®

I - 6

Copyright © 2009, Oracle. All rights reserved.

Lesson Agenda

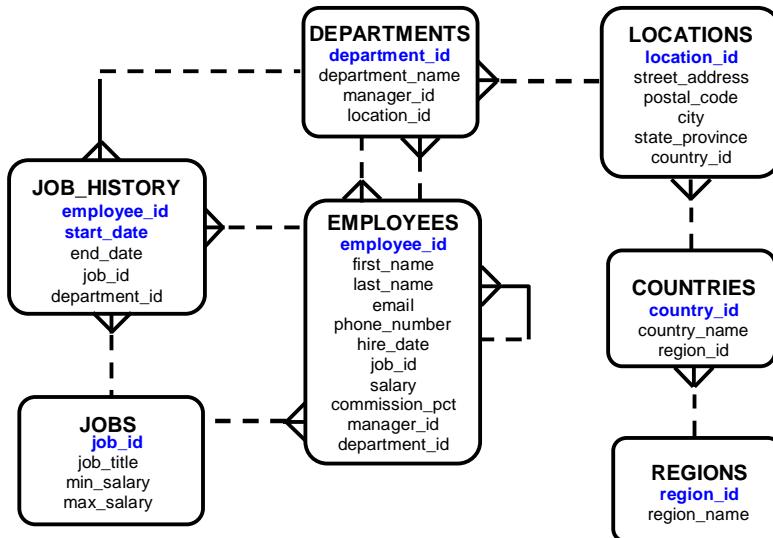
- Course objectives and course agenda
- The schema and appendixes used in this course and the available PL/SQL development environments in this course
- Oracle 11g documentation and additional resources

ORACLE®

I - 7

Copyright © 2009, Oracle. All rights reserved.

The Human Resources (HR) Schema That Is Used in This Course



ORACLE®

The Human Resources (HR) Schema Description

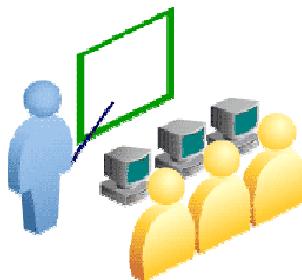
The Human Resources (HR) schema is part of Oracle Sample Schemas that can be installed in an Oracle database. The practice sessions in this course use data from the HR schema.

Table Descriptions

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

Class Account Information

- Cloned HR account IDs are set up for you.
- Your account IDs are ora61 or ora62.
- The password matches your account ID.
- Each machine is assigned one account.
- The instructor has a separate ID.



ORACLE®

I - 9

Copyright © 2009, Oracle. All rights reserved.

Note: Use either account ID ora61 or ora62.

Appendices Used in This Course

- Appendix A: Practices and Solutions
- Appendix AP: Additional Practices and Solutions
- Appendix B: Table Descriptions
- Appendix C: Using SQL Developer
- Appendix D: Using SQL*Plus
- Appendix E: Review of JDeveloper
- Appendix F: Review of PL/SQL
- Appendix G: Studies for Implementing Triggers
- Appendix H: Using the DBMS_SCHEDULER and HTP Packages

ORACLE®

PL/SQL Development Environments

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

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



PL/SQL Development Environments

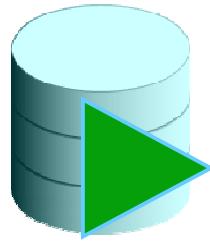
There are many tools that provide an environment for developing PL/SQL code. Oracle provides several tools that can be used to write PL/SQL code. Some of the development tools that are available for use in this course are:

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

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

What Is Oracle SQL Developer?

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



SQL Developer

ORACLE®

I - 12

Copyright © 2009, 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 maintain stored procedures, test SQL statements, and view optimizer plans.

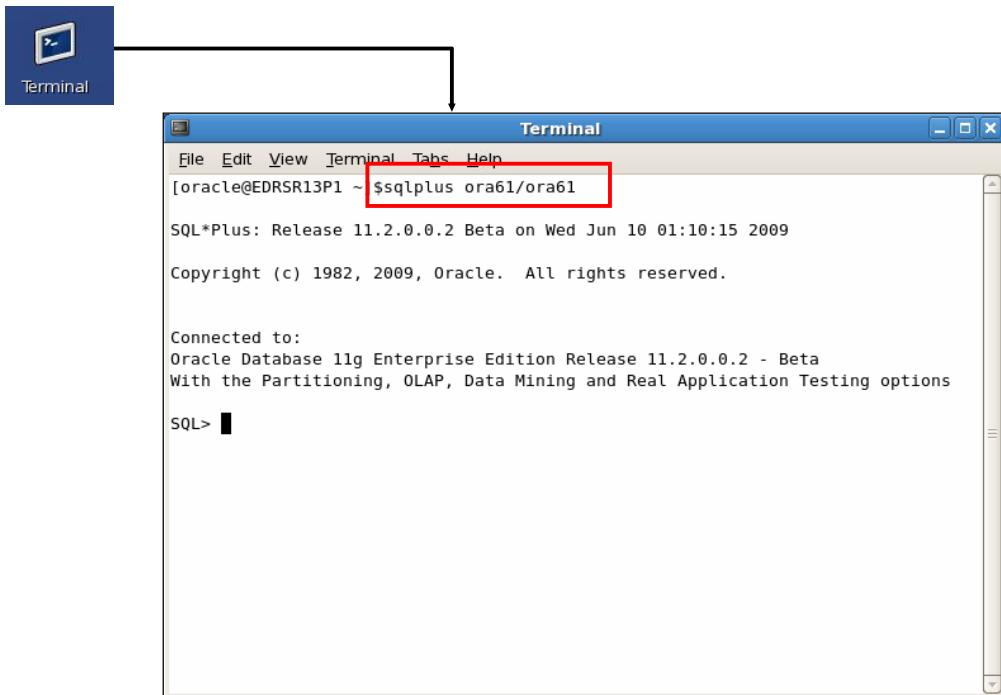
SQL Developer, the visual tool for database development, simplifies the following tasks:

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

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

Note: Appendix C of this course provides an introduction to using the SQL Developer interface. See the appendix now for information about creating a database connection and interacting with data using SQL and PL/SQL, among other topics.

Coding PL/SQL in SQL*Plus



ORACLE®

I - 13

Copyright © 2009, Oracle. All rights reserved.

Coding PL/SQL in SQL*Plus

Oracle SQL*Plus is a graphical user interface (GUI) or 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:

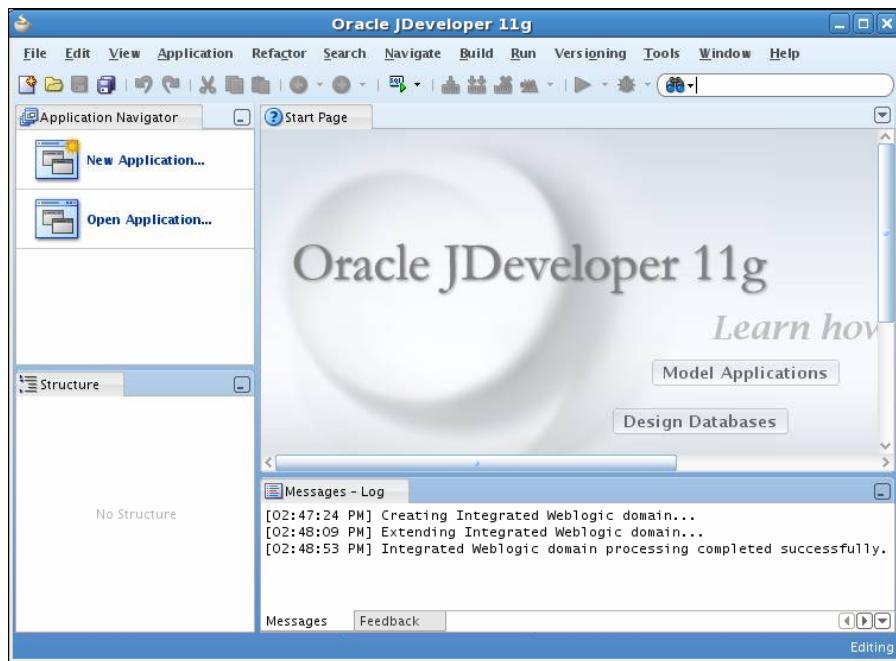
- Shipped with the database
- Installed on a client and on the database server system
- Accessed from an icon or the command line

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

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

Note: For more information about how to use SQL*Plus, see Appendix D.

Coding PL/SQL in Oracle JDeveloper



ORACLE®

I - 14

Copyright © 2009, Oracle. All rights reserved.

Coding PL/SQL in Oracle JDeveloper

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

When coding PL/SQL in JDeveloper, consider the following:

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

Note:

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

Enabling Output of a PL/SQL Block

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

```
SET SERVEROUTPUT ON;
```

2. Use the predefined DBMS_OUTPUT Oracle package and its procedure to display the output as follows:
 - DBMS_OUTPUT.PUT_LINE

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

ORACLE®

I - 15

Copyright © 2009, Oracle. All rights reserved.

Enabling Output of a PL/SQL Block

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

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

1. Execute the following SQL command:

```
SET SERVEROUTPUT ON
```

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

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

Lesson Agenda

- Course objectives and course agenda
- The schema and appendices used in this course and the available PL/SQL development environments in this course
- Oracle 11g documentation and additional resources

ORACLE®

Oracle 11g SQL and PL/SQL Documentation

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

ORACLE®

I - 17

Copyright © 2009, Oracle. All rights reserved.

Oracle 11g SQL and PL/SQL Documentation

Navigate to <http://www.oracle.com/pls/db111/homepage> and click the Master Book List link in the left frame.

Additional Resources

For additional information about the new Oracle 11g SQL and PL/SQL new features, refer to the following:

- Oracle Database 11g: New Features eStudies
- Oracle by Example (OBE) series: Oracle Database 11g:
 - http://www.oracle.com/technology/obe/11gr1_db/admin/11gr1db.html
- What's New in PL/SQL in Oracle Database 11g on the Oracle Technology Network (OTN):
 - http://www.oracle.com/technology/tech/pl_sql/



Summary

In this lesson, you should have learned how to:

- Discuss the goals of the course
- Identify the available 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

The PL/SQL language provides different program constructs for blocks of reusable code. Unnamed or anonymous PL/SQL blocks can be used to invoke SQL and PL/SQL actions, procedures, functions, and package components. Named PL/SQL blocks, otherwise known as subprograms, include:

- Procedures
- Functions
- Package procedures and functions
- Triggers

Oracle supplies several tools to develop your PL/SQL functionality. Oracle provides a client-side or middle-tier PL/SQL run-time environment for Oracle Forms and Oracle Reports, and provides a PL/SQL run-time engine inside the Oracle database. Procedures and functions inside the database can be invoked from any application code that can connect to an Oracle database and execute PL/SQL code.

Practice I Overview: Getting Started

This practice covers the following topics:

- Reviewing the available SQL Developer resources
- Starting SQL Developer and creating a new database connection and browsing your schema tables
- Setting some SQL Developer preferences
- Executing SQL statements and an anonymous PL/SQL block using SQL Worksheet
- Accessing and bookmarking the Oracle Database 11g documentation and other useful Web sites



Practice I: Overview

In this practice, you use SQL Developer to execute SQL statements to examine data in your schema. 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 or JDeveloper environments that are available in this course.

1

Creating Procedures

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

Objectives

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

- Identify the benefits of modularized and layered subprogram design
- Create and call procedures
- Use formal and actual parameters
- Use positional, named, or mixed notation for passing parameters
- Identify the available parameter-passing modes
- Handle exceptions in procedures
- Remove a procedure
- Display the procedures' information

ORACLE®

1 - 2

Copyright © 2009, Oracle. All rights reserved.

Lesson Aim

In this lesson, you learn to create, execute, and remove procedures with or without parameters. Procedures are the foundation of modular programming in PL/SQL. To make procedures more flexible, it is important that varying data is either calculated or passed into a procedure by using input parameters. Calculated results can be returned to the caller of a procedure by using OUT parameters.

To make your programs robust, you should always manage exception conditions by using the exception-handling features of PL/SQL.

Lesson Agenda

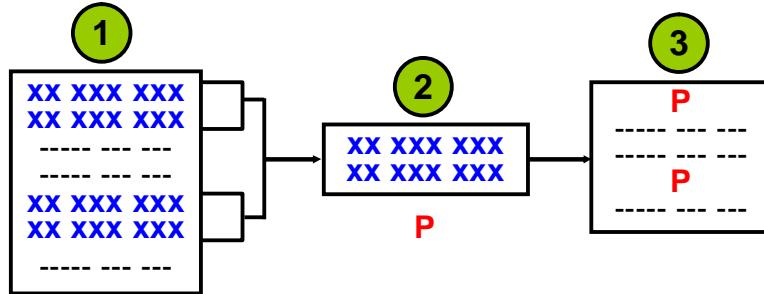
- Using a modularized and layered subprogram design and identifying the benefits of subprograms
- Working with procedures:
 - Creating and calling procedures
 - Identifying the available parameter-passing modes
 - Using formal and actual parameters
 - Using positional, named, or mixed notation
- Handling exceptions in procedures, removing a procedure, and displaying the procedures' information

ORACLE®

1 - 3

Copyright © 2009, Oracle. All rights reserved.

Creating a Modularized Subprogram Design



Modularize code into subprograms.

1. Locate code sequences repeated more than once.
2. Create subprogram P containing the repeated code
3. Modify original code to invoke the new subprogram.

ORACLE®

1 - 4

Copyright © 2009, Oracle. All rights reserved.

Creating a Modularized and Layered Subprogram Design

The diagram illustrates the principle of modularization with subprograms: the creation of smaller manageable pieces of flexible and reusable code. Flexibility is achieved by using subprograms with parameters, which in turn makes the same code reusable for different input values. To modularize existing code, perform the following steps:

1. Locate and identify repetitive sequences of code.
2. Move the repetitive code into a PL/SQL subprogram.
3. Replace the original repetitive code with calls to the new PL/SQL subprogram.

Following this modular and layered approach can help you create code that is easier to maintain, particularly when the business rules change. In addition, keeping the SQL logic simple and free of complex business logic can benefit from the work of Oracle Database Optimizer, which can reuse parsed SQL statements for better use of server-side resources.

Creating a Layered Subprogram Design

Create subprogram layers for your application.

- Data access subprogram layer with SQL logic
- Business logic subprogram layer, which may or may not use the data access layer



Creating a Layered Subprogram Design

Because PL/SQL allows SQL statements to be seamlessly embedded into the logic, it is too easy to have SQL statement spread all over the code. However, it is recommended that you keep the SQL logic separate from the business logic—that is, create a layered application design with a minimum of two layers:

- **Data access layer:** For subroutines to access the data by using SQL statements
- **Business logic layer:** For subprograms to implement the business processing rules, which may or may not call on the data access layer routines

Modularizing Development with PL/SQL Blocks

- PL/SQL is a block-structured language. The PL/SQL code block helps modularize code by using:
 - Anonymous blocks
 - Procedures and functions
 - Packages
 - Database triggers
- The benefits of using modular program constructs are:
 - Easy maintenance
 - Improved data security and integrity
 - Improved performance
 - Improved code clarity

ORACLE®

1 - 6

Copyright © 2009, Oracle. All rights reserved.

Modularizing Development with PL/SQL Blocks

A subprogram is based on standard PL/SQL structures. It contains a declarative section, an executable section, and an optional exception-handling section (for example, anonymous blocks, procedures, functions, packages, and triggers). Subprograms can be compiled and stored in the database, providing modularity, extensibility, reusability, and maintainability.

Modularization converts large blocks of code into smaller groups of code called modules. After modularization, the modules can be reused by the same program or shared with other programs. It is easier to maintain and debug code that comprises smaller modules than it is to maintain code in a single large program. Modules can be easily extended for customization by incorporating more functionality, if required, without affecting the remaining modules of the program.

Subprograms provide easy maintenance because the code is located in one place and any modifications required to the subprogram can, therefore, be performed in this single location. Subprograms provide improved data integrity and security. The data objects are accessed through the subprogram, and a user can invoke the subprogram only if the appropriate access privilege is granted to the user.

Note: Knowing how to develop anonymous blocks is a prerequisite for this course. For detailed information about anonymous blocks, see the course titled *Oracle 11g: PL/SQL Fundamentals*.

Anonymous Blocks: Overview

Anonymous blocks:

- Form the basic PL/SQL block structure
- Initiate PL/SQL processing tasks from applications
- Can be nested within the executable section of any PL/SQL block

```
[DECLARE      -- Declaration Section (Optional)
     variable declarations; ... ]
BEGIN        -- Executable Section (Mandatory)
    SQL or PL/SQL statements;
[EXCEPTION    -- Exception Section (Optional)
    WHEN exception THEN statements; ]
END;         -- End of Block (Mandatory)
```

ORACLE®

1 - 7

Copyright © 2009, Oracle. All rights reserved.

Anonymous Blocks: Overview

Anonymous blocks are typically used for:

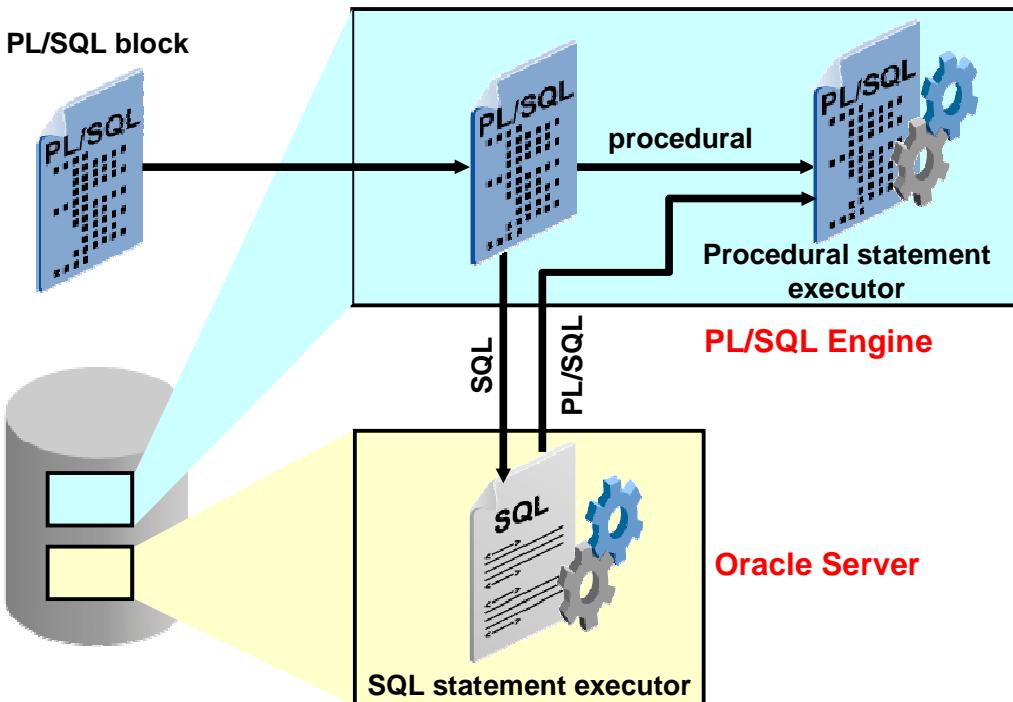
- Writing trigger code for Oracle Forms components
- Initiating calls to procedures, functions, and package constructs
- Isolating exception handling within a block of code
- Nesting inside other PL/SQL blocks for managing code flow control

The DECLARE keyword is optional, but it is required if you declare variables, constants, and exceptions to be used within the PL/SQL block.

BEGIN and END are mandatory and require at least one statement between them, either SQL, PL/SQL, or both.

The exception section is optional and is used to handle errors that occur within the scope of the PL/SQL block. Exceptions can be propagated to the caller of the anonymous block by excluding an exception handler for the specific exception, thus creating what is known as an *unhandled* exception.

PL/SQL Run-time Architecture



PL/SQL Run-time Architecture

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

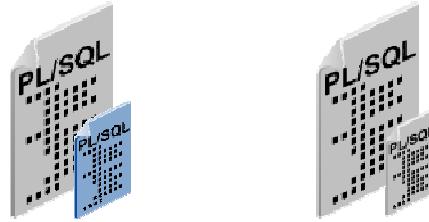
- The Oracle database for executing stored subprograms
- The Oracle Forms client when running client/server applications, or in the Oracle Application Server when using Oracle Forms Services to run Forms on the Web

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

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

What Are PL/SQL Subprograms?

- A PL/SQL subprogram is a named PL/SQL block that can be called with a set of parameters.
- You can declare and define a subprogram within either a PL/SQL block or another subprogram.
- A subprogram consists of a specification and a body.
- A subprogram can be a procedure or a function.
- Typically, you use a procedure to perform an action and a function to compute and return a value.
- Subprograms can be grouped into PL/SQL packages.



ORACLE®

1 - 9

Copyright © 2009, Oracle. All rights reserved.

What Are PL/SQL Subprograms?

A PL/SQL subprogram is a named PL/SQL block that can be called with a set of parameters. You can declare and define a subprogram within either a PL/SQL block or another subprogram.

Subprogram Parts

A subprogram consists of a specification (spec) and a body. To declare a subprogram, you must provide the spec, which includes descriptions of any parameters. To define a subprogram, you must provide both the spec and the body. You can either declare a subprogram first and define it later in the same block or subprogram, or declare and define it at the same time.

Subprogram Types

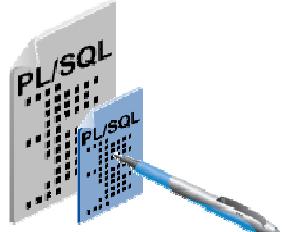
PL/SQL has two types of subprograms: procedures and functions. Typically, you use a procedure to perform an action and a function to compute and return a value.

A procedure and a function have the same structure, except that only a function has some additional items such as the RETURN clause or the RETURN statement.

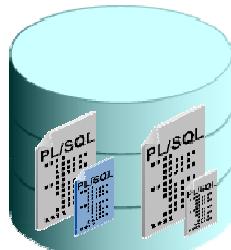
The RETURN clause specifies the data type of the return value (required). A RETURN statement specifies the return value (required). Functions are covered in more detail in the next lesson titled “Creating Functions and Debugging Subprograms.”

Subprograms can be grouped into PL/SQL packages, which make code even more reusable and maintainable. Packages are covered in the packages lessons: Lessons 3 and 4.

The Benefits of Using PL/SQL Subprograms



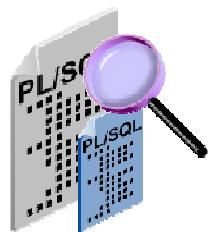
Easy maintenance



Subprograms:
Stored procedures
and functions



Improved data
security and integrity



Improved code clarity



Improved performance

ORACLE®

Benefits of Subprograms

Procedures and functions have many benefits due to modularizing of code:

- **Easy maintenance** is realized because subprograms are located in one place. Modifications need to be done in only one place to affect multiple applications and minimize excessive testing.
- **Improved data security** can be achieved by controlling indirect access to database objects from nonprivileged users with security privileges. The subprograms are by default executed with definer's right. The execute privilege does not allow a calling user direct access to objects that are accessible to the subprogram.
- **Data integrity** is managed by having related actions performed together or not at all.
- **Improved performance** can be realized from reuse of parsed PL/SQL code that becomes available in the shared SQL area of the server. Subsequent calls to the subprogram avoid parsing the code again. Because PL/SQL code is parsed at compile time, the parsing overhead of SQL statements is avoided at run time. Code can be written to reduce the number of network calls to the database, and therefore, decrease network traffic.
- **Improved code clarity** can be attained by using appropriate names and conventions to describe the action of the routines, thereby reducing the need for comments and enhancing the clarity of the code.

Differences Between Anonymous Blocks and Subprograms

Anonymous Blocks	Subprograms
Unnamed PL/SQL blocks	Named PL/SQL blocks
Compiled every time	Compiled only once
Not stored in the database	Stored in the database
Cannot be invoked by other applications	Named and, therefore, can be invoked by other applications
Do not return values	Subprograms called functions must return values.
Cannot take parameters	Can take parameters

ORACLE®

1 - 11

Copyright © 2009, Oracle. All rights reserved.

Differences Between Anonymous Blocks and Subprograms

The table in the slide not only shows the differences between anonymous blocks and subprograms, but also highlights the general benefits of subprograms.

Anonymous blocks are not persistent database objects. They are compiled and executed only once. They are not stored in the database for reuse. If you want to reuse, you must rerun the script that creates the anonymous block, which causes recompilation and execution.

Procedures and functions are compiled and stored in the database in a compiled form.

They are recompiled only when they are modified. Because they are stored in the database, any application can make use of these subprograms based on appropriate permissions. The calling application can pass parameters to the procedures if the procedure is designed to accept parameters. Similarly, a calling application can retrieve a value if it invokes a function or a procedure.

Lesson Agenda

- Using a modularized and layered subprogram design and identifying the benefits of subprograms
- Working with procedures:
 - Creating and calling procedures
 - Identifying the available parameter-passing modes
 - Using formal and actual parameters
 - Using positional, named, or mixed notation
- Handling exceptions in procedures, removing a procedure, and displaying the procedures' information

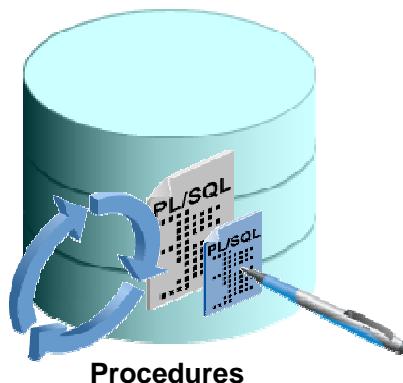
ORACLE®

1 - 12

Copyright © 2009, Oracle. All rights reserved.

What Are Procedures?

- Are a type of subprogram that perform an action
- Can be stored in the database as a schema object
- Promote reusability and maintainability



ORACLE®

1 - 13

Copyright © 2009, Oracle. All rights reserved.

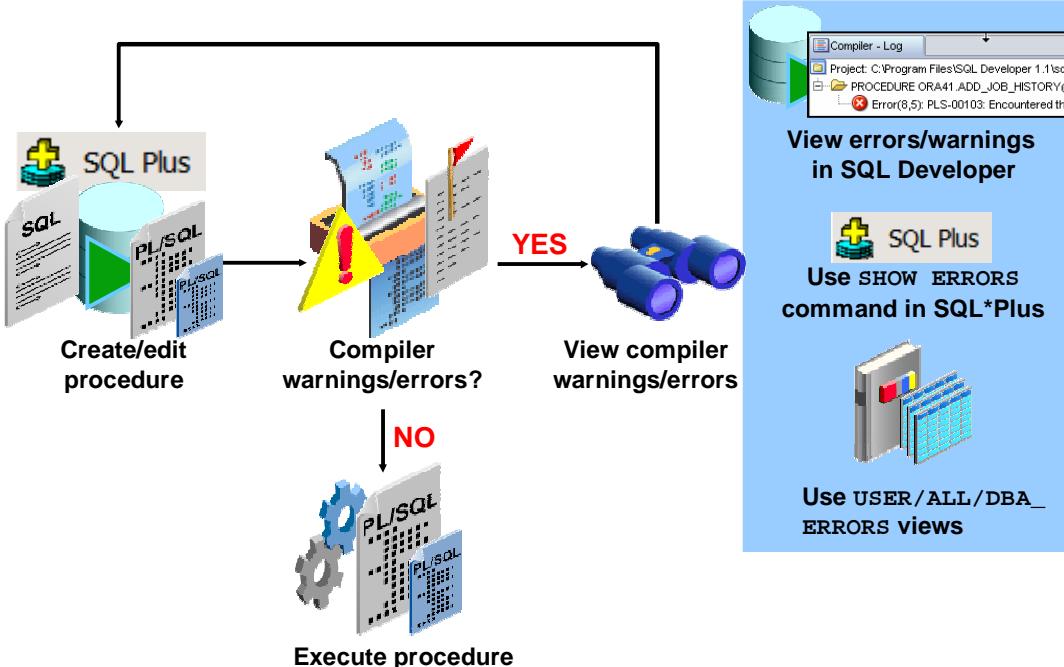
Definition of a Procedure

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. It has a header, a declaration section, an executable section, and an optional exception-handling section. A procedure is invoked by using the procedure name in the execution section of another PL/SQL block.

A procedure is compiled and stored in the database as a schema object. If you are using the procedures with Oracle Forms and Reports, then they can be compiled within the Oracle Forms or Oracle Reports executables.

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

Creating Procedures: Overview



Creating Procedures: Overview

To develop a procedure using a tool such as SQL Developer, perform the following steps:

1. Create the procedure using SQL Developer's Object Navigator tree or the SQL Worksheet area.
2. Compile the procedure. The procedure is created in the database and gets compiled. The CREATE PROCEDURE statement creates and stores the source code and the compiled *m-code* in the database. To compile the procedure, right-click the procedure's name in the Object Navigator tree, and then click Compile.
3. If compilation errors exist, then the *m-code* is not stored and you must edit the source code to make corrections. You cannot invoke a procedure that contains compilation errors. You can view the compilation errors in SQL Developer, SQL*Plus, or the appropriate data dictionary views as shown in the slide.
4. After successful compilation, execute the procedure to perform the desired action. You can run the procedure using SQL Developer or use the EXECUTE command in SQL*Plus.

Note: If compilation errors occur, use a CREATE OR REPLACE PROCEDURE statement to overwrite the existing code if you previously used a CREATE PROCEDURE statement.

Otherwise, drop the procedure first (using DROP) and then execute the CREATE PROCEDURE statement.

Creating Procedures with the SQL CREATE OR REPLACE Statement

- Use the CREATE clause to create a stand-alone procedure that is stored in the Oracle database.
- Use the OR REPLACE option to overwrite an existing procedure.

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

PL/SQL block

Creating Procedures with the SQL CREATE OR REPLACE Statement

You can use the CREATE PROCEDURE SQL statement to create stand-alone procedures that are stored in an Oracle database. A procedure is similar to a miniature program: it performs a specific action. You specify the name of the procedure, its parameters, its local variables, and the BEGIN–END block that contains its code and handles any exceptions.

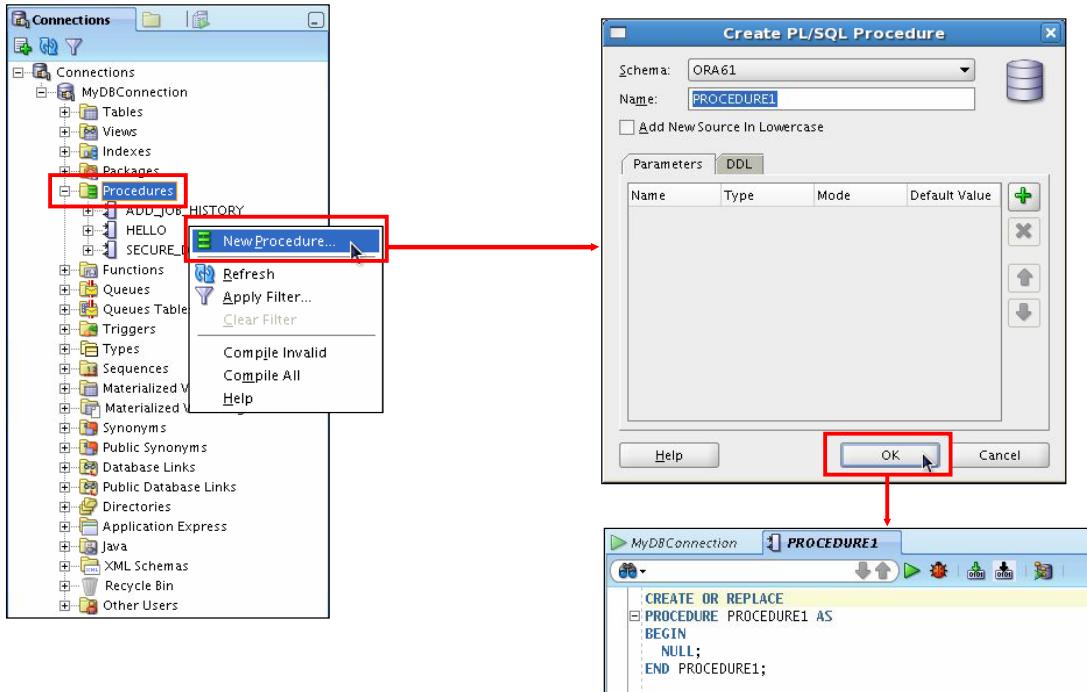
- PL/SQL blocks start with BEGIN, optionally preceded by the declaration of local variables. PL/SQL blocks end with either END or END *procedure_name*.
- The REPLACE option indicates that if the procedure exists, it is dropped and replaced with the new version created by the statement. The REPLACE option does not drop any of the privileges associated with the procedure.

Other Syntactic Elements

- *parameter1* represents the name of a parameter.
- The *mode* option defines how a parameter is used: IN (default), OUT, or IN OUT.
- *datatype1* specifies the parameter data type, without any precision.

Note: Parameters can be considered as local variables. Substitution and host (bind) variables cannot be referenced anywhere in the definition of a PL/SQL stored procedure. The OR REPLACE option does not require any change in object security, as long as you own the object and have the CREATE [ANY] PROCEDURE privilege.

Creating Procedures Using SQL Developer



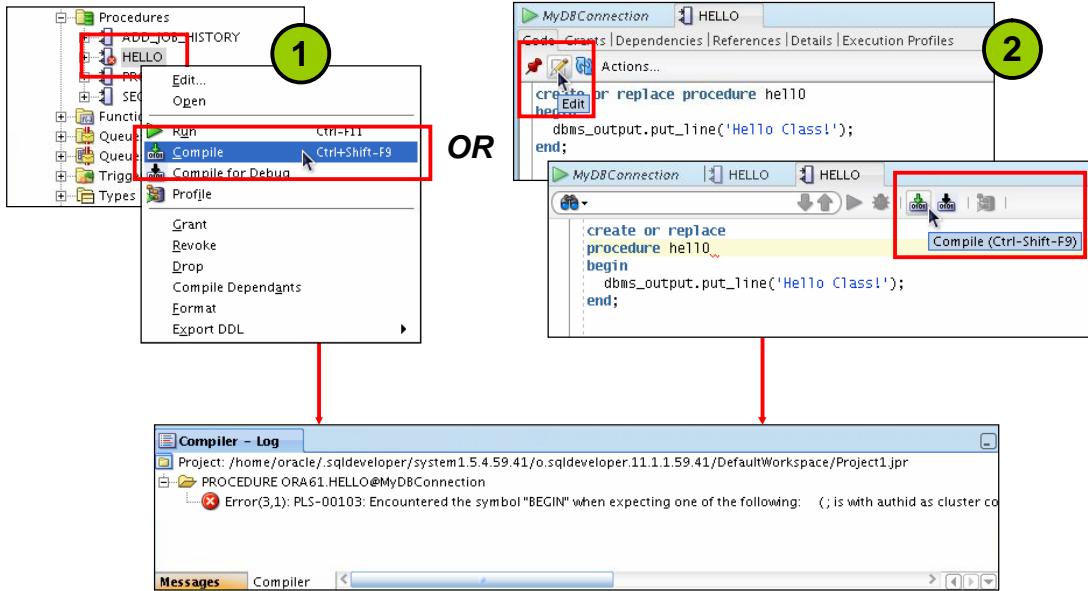
Creating Procedures Using SQL Developer

1. Right-click the **Procedures** node on the **Connections** tabbed page.
2. Select **New Procedure** from the shortcut menu. The **Create PL/SQL Procedure** dialog box is displayed. Specify the information for the new procedure, and then click **OK** to create the subprogram and have it displayed in the Editor window, where you can enter the details.

The components of the **Create PL/SQL Procedure** dialog box are as follows:

- **Schema:** The database schema in which to create the PL/SQL subprogram
- **Name:** The name of the subprogram that must be unique within a schema
- **Add New Source in Lowercase:** If this option is selected, new text appears in lowercase regardless of the case in which you enter it. This option affects only the appearance of the code, because PL/SQL is not case-sensitive in its execution.
- **Parameters tab:** To add a parameter, click the Add (+) icon. For each parameter in the procedure to be created, specify the parameter name, data type, mode, and optionally the default Value. Use the Remove (X) icon and the arrow icons to delete and to move a parameter up or down in the list respectively.
- **DDL tab:** This tab contains a read-only display of a SQL statement that reflects the current definition of the subprogram.

Compiling Procedures and Displaying Compilation Errors in SQL Developer



ORACLE®

1 - 17

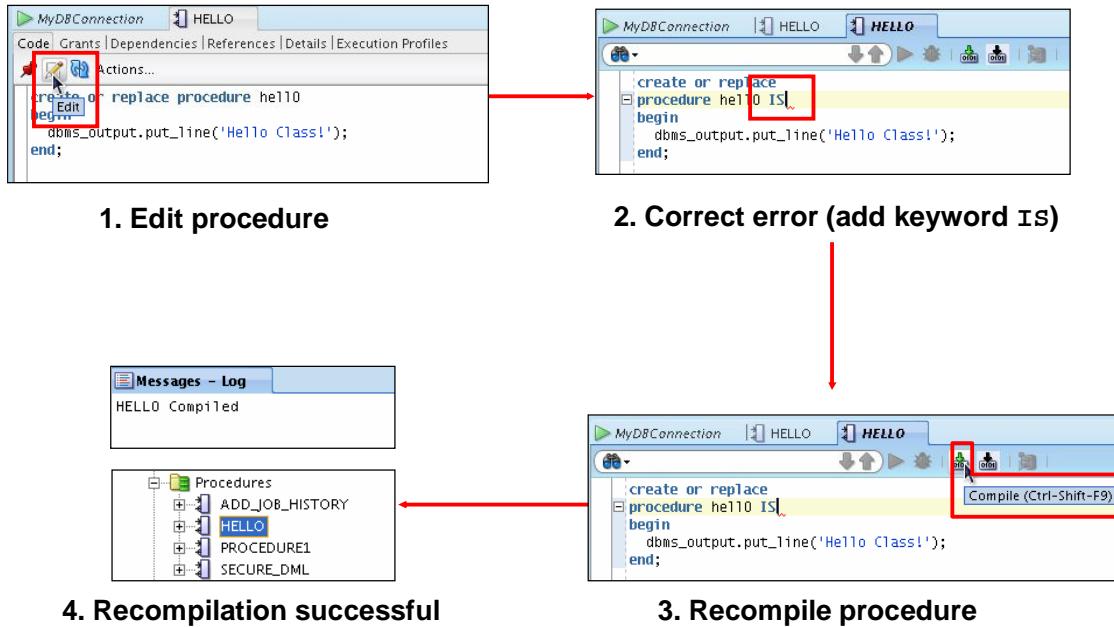
Copyright © 2009, Oracle. All rights reserved.

Compiling Procedures and Displaying Compilation Errors in SQL Developer

You can compile procedures using one of the following two methods:

- Navigate to the Procedures node in the Object Navigator tree. Right-click the procedure's name, and then select Compile from the shortcut menu. To view any compilation messages, view the Messages subtab in the **Compiler – Log** tab.
- Edit the procedure using the Edit icon on the procedure's code toolbar. Make the necessary edits, and then click the Compile icon on the code toolbar. To view any compilation messages, view the Messages subtab in the **Compiler – Log** tab.

Correcting Compilation Errors in SQL Developer



Correcting Compilation Errors in SQL Developer

1. Edit the procedure using the Edit icon on the procedure's code toolbar. A new procedure code tab is opened in Read/Write mode.
2. Make the necessary corrections.
3. Click the Compile icon on the code toolbar.
4. To view any compilation messages, view the Messages subtab in the **Compiler – Log** tab. In addition, if the procedure compiled successfully, the red X on the procedure's name in the Object Navigator tree is removed.

Naming Conventions of PL/SQL Structures Used in This Course

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

ORACLE®

Naming Conventions of PL/SQL Structures Used in This Course

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

What Are Parameters and Parameter Modes?

- Are declared after the subprogram name in the PL/SQL header
- Pass or communicate data between the calling environment and the subprogram
- Are used like local variables but are dependent on their parameter-passing mode:
 - An `IN` parameter mode (the default) provides values for a subprogram to process
 - An `OUT` parameter mode returns a value to the caller
 - An `IN OUT` parameter mode supplies an input value, which may be returned (output) as a modified value

ORACLE®

1 - 20

Copyright © 2009, Oracle. All rights reserved.

What Are Parameters?

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.

Parameters can be thought of as a special form of local variable, whose input values are initialized by the calling environment when the subprogram is called, and whose output values are returned to the calling environment when the subprogram returns control to the caller.

Formal and Actual Parameters

- Formal parameters: Local variables declared in the parameter list of a subprogram specification
- Actual parameters (or arguments): Literal values, variables, and expressions used in the parameter list of the calling subprogram

```
-- Procedure definition, Formal parameters
CREATE PROCEDURE raise_sal(p_id NUMBER, p_sal NUMBER) IS
BEGIN
  . . .
END raise_sal;

-- Procedure calling, Actual parameters (arguments)
v_emp_id := 100;
raise_sal(v_emp_id, 2000)
```

ORACLE®

1 - 21

Copyright © 2009, Oracle. All rights reserved.

Formal and Actual Parameters

Formal parameters are local variables that are declared in the parameter list of a subprogram specification. In the first example, in the `raise_sal` procedure, the variable `p_id` and `p_sal` identifiers represent the formal parameters.

The actual parameters can be literal values, variables, and expressions that are provided in the parameter list of a calling subprogram. In the second example, a call is made to `raise_sal`, where the `v_emp_id` variable provides the actual parameter value for the `p_id` formal parameter and `2000` is supplied as the actual parameter value for `p_sal`. Actual parameters:

- Are associated with formal parameters during the subprogram call
- Can also be expressions, as in the following example:
`raise_sal(v_emp_id, raise+100);`

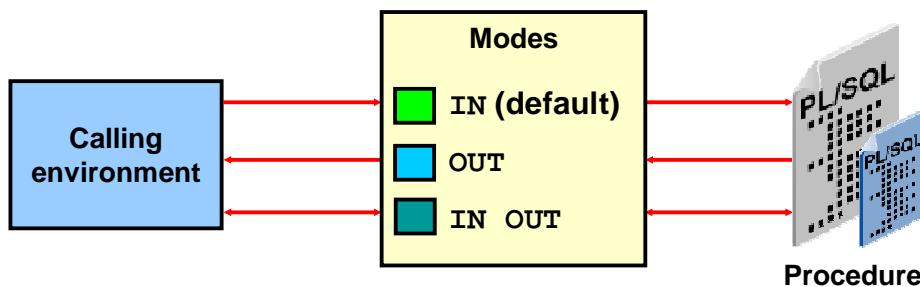
The formal and actual parameters should be of compatible data types. If necessary, before assigning the value, PL/SQL converts the data type of the actual parameter value to that of the formal parameter.

Note: Actual parameters are also referred to as *actual arguments*.

Procedural Parameter Modes

- Parameter modes are specified in the formal parameter declaration, after the parameter name and before its data type.
- The `IN` mode is the default if no mode is specified.

```
CREATE PROCEDURE proc_name(param_name [mode] datatype)
...
```



Procedural Parameter Modes

When you create a procedure, the formal parameter defines a variable name whose value is used in the executable section of the PL/SQL block. The actual parameter is used when invoking the procedure to provide input values or receive output results.

The parameter mode `IN` is the default passing mode—that is, if no mode is specified with a parameter declaration, the parameter is considered to be an `IN` parameter. The parameter modes `OUT` and `IN OUT` must be explicitly specified in their parameter declarations.

The `datatype` parameter is specified without a size specification. It can be specified:

- As an explicit data type
- Using the `%TYPE` definition
- Using the `%ROWTYPE` definition

Note: One or more formal parameters can be declared, each separated by a comma.

Comparing the Parameter Modes

IN	OUT	IN OUT
Default mode	Must be specified	Must be specified
Value is passed into subprogram	Value is returned to the calling environment	Value passed into subprogram; value returned to calling environment
Formal parameter acts as a constant	Uninitialized variable	Initialized variable
Actual parameter can be a literal, expression, constant, or initialized variable	Must be a variable	Must be a variable
Can be assigned a default value	Cannot be assigned a default value	Cannot be assigned a default value

ORACLE®

1 - 23

Copyright © 2009, Oracle. All rights reserved.

Comparing the Parameter Modes

The IN parameter mode is the default mode if no mode is specified in the declaration. The OUT and IN OUT parameter modes must be explicitly specified with the parameter declaration.

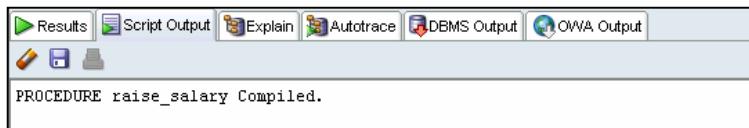
A formal parameter of IN mode cannot be assigned a value and cannot be modified in the body of the procedure. By default, the IN parameter is passed by reference. An IN parameter can be assigned a default value in the formal parameter declaration, in which case the caller need not provide a value for the parameter if the default applies.

An OUT or IN OUT parameter must be assigned a value before returning to the calling environment. The OUT and IN OUT parameters cannot be assigned default values. To improve performance with OUT and IN OUT parameters, the NOCOPY compiler hint can be used to request to pass by reference.

Note: Using NOCOPY is discussed later in this course.

Using the IN Parameter Mode: Example

```
CREATE OR REPLACE PROCEDURE raise_salary
(p_id      IN employees.employee_id%TYPE,
 p_percent IN NUMBER)
IS
BEGIN
  UPDATE employees
  SET    salary = salary * (1 + p_percent/100)
  WHERE employee_id = p_id;
END raise_salary;
/
```



```
EXECUTE raise_salary(176, 10)
```

ORACLE

1 - 24

Copyright © 2009, Oracle. All rights reserved.

Using IN Parameters: Example

The example in the slide shows a procedure with two IN parameters. Running the first slide example creates the `raise_salary` procedure in the database. The second slide example invokes `raise_salary` and provides the first parameter value of 176 for the employee ID, and a salary increase of 10 percent for the second parameter value.

To invoke a procedure by using the SQL Worksheet of SQL Developer or by using SQL*Plus, use the following EXECUTE command shown in the second code example in the slide.

To invoke a procedure from another procedure, use a direct call inside an executable section of the calling block. At the location of calling the new procedure, enter the procedure name and actual parameters. For example:

```
...
BEGIN
  raise_salary (176, 10);
END;
```

Note: IN parameters are passed as read-only values from the calling environment into the procedure. Attempts to change the value of an IN parameter result in a compile-time error.

Using the OUT Parameter Mode: Example

```
CREATE OR REPLACE PROCEDURE query_emp
  (p_id      IN employees.employee_id%TYPE,
   p_name    OUT employees.last_name%TYPE,
   p_salary  OUT employees.salary%TYPE) IS
BEGIN
  SELECT last_name, salary INTO p_name, p_salary
  FROM   employees
  WHERE  employee_id = p_id;
END query_emp;
/
```

```
SET SERVEROUTPUT ON
DECLARE
  v_emp_name employees.last_name%TYPE;
  v_emp_sal  employees.salary%TYPE;
BEGIN
  query_emp(171, v_emp_name, v_emp_sal);
  DBMS_OUTPUT.PUT_LINE(v_emp_name || ' earns ' ||
    to_char(v_emp_sal, '$999,999.00'));
END;
/
```

ORACLE®

1 - 25

Copyright © 2009, Oracle. All rights reserved.

Using the OUT Parameters: Example

In the slide example, you create a procedure with OUT parameters to retrieve information about an employee. The procedure accepts the value 171 for employee ID and retrieves the name and salary of the employee with ID 171 into the two OUT parameters. The query_emp procedure has three formal parameters. Two of them are OUT parameters that return values to the calling environment, shown in the second code box in the slide. The procedure accepts an employee ID value through the p_id parameter. The v_emp_name and v_emp_salary variables are populated with the information retrieved from the query into their two corresponding OUT parameters. The following is the result of running the code in the second code example in the slide. v_emp_name holds the value Smith and v_emp_salary holds the value 7400:

The screenshot shows the Oracle SQL Developer interface. At the top, there is a toolbar with several icons: Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there is a menu bar with File, Edit, Tools, Database, Help, and a View menu. The main area displays the output of an anonymous block. The output shows the message "anonymous block completed" followed by the result "Smith earns \$7,400.00".

```
anonymous block completed
Smith earns $7,400.00
```

Note: Make sure that the data type for the actual parameter variables used to retrieve values from the OUT parameters has a size sufficient to hold the data values being returned.

Using the IN OUT Parameter Mode: Example

Calling environment



```
CREATE OR REPLACE PROCEDURE format_phone
  (p_phone_no IN OUT VARCHAR2) IS
BEGIN
  p_phone_no := '(' || SUBSTR(p_phone_no,1,3) ||
                 ')' || SUBSTR(p_phone_no,4,3) ||
                 '-' || SUBSTR(p_phone_no,7);
END format_phone;
/
```

```
anonymous block completed
b_phone_no
-----
8006330575
anonymous block completed
b_phone_no
-----
(800) 633-0575
```

ORACLE®

Using IN OUT Parameters: Example

Using an IN OUT parameter, you can pass a value into a procedure that can be updated. The actual parameter value supplied from the calling environment can return either the original unchanged value or a new value that is set within the procedure.

Note: An IN OUT parameter acts as an initialized variable.

The slide example creates a procedure with an IN OUT parameter to accept a 10-character string containing digits for a phone number. The procedure returns the phone number formatted with parentheses around the first three characters and a hyphen after the sixth digit—for example, the phone string 8006330575 is returned as (800) 633-0575.

The following code uses the `b_phone_no` host variable of SQL*Plus to provide the input value passed to the `FORMAT_PHONE` procedure. The procedure is executed and returns an updated string in the `b_phone_no` host variable. The output of the following code is displayed in the slide above:

```
VARIABLE b_phone_no VARCHAR2(15)
EXECUTE :b_phone_no := '8006330575'
PRINT b_phone_no
EXECUTE format_phone (:b_phone_no)
PRINT b_phone_no
```

Viewing the OUT Parameters: Using the DBMS_OUTPUT.PUT_LINE Subroutine

Use PL/SQL variables that are printed with calls to the DBMS_OUTPUT.PUT_LINE procedure.

```
SET SERVEROUTPUT ON

DECLARE
    v_emp_name employees.last_name%TYPE;
    v_emp_sal  employees.salary%TYPE;
BEGIN
    query_emp(171, v_emp_name, v_emp_sal);
    DBMS_OUTPUT.PUT_LINE('Name: ' || v_emp_name);
    DBMS_OUTPUT.PUT_LINE('Salary: ' || v_emp_sal);
END;
```

```
anonymous block completed
Name: Smith
Salary: 7400
```

ORACLE®

1 - 27

Copyright © 2009, Oracle. All rights reserved.

Viewing the OUT Parameters: Using the DBMS_OUTPUT Subroutine

The slide example illustrates how to view the values returned from the OUT parameters in SQL*Plus or the SQL Developer Worksheet.

You can use PL/SQL variables in an anonymous block to retrieve the OUT parameter values. The DBMS_OUTPUT.PUT_LINE procedure is called to print the values held in the PL/SQL variables. The SET SERVEROUTPUT must be ON.

Viewing OUT Parameters: Using SQL*Plus Host Variables

1. Use SQL*Plus host variables.
2. Execute QUERY_EMP using host variables.
3. Print the host variables.

```
VARIABLE b_name    VARCHAR2( 25 )
VARIABLE b_sal     NUMBER
EXECUTE query_emp(171, :b_name, :b_sal)
PRINT b_name b_sal
```

```
anonymous block completed
b_name
-----
Smith

b_sal
-----
7400
```

ORACLE®

1 - 28

Copyright © 2009, Oracle. All rights reserved.

Viewing OUT Parameters: Using SQL*Plus Host Variables

The example in the slide demonstrates how to use SQL*Plus host variables that are created using the VARIABLE command. The SQL*Plus variables are external to the PL/SQL block and are known as host or bind variables. To reference host variables from a PL/SQL block, you must prefix their names with a colon (:). To display the values stored in the host variables, you must use the SQL*Plus PRINT command followed by the name of the SQL*Plus variable (without the colon because this is not a PL/SQL command or context).

Note: For details about the VARIABLE command, see the SQL*Plus Command Reference.

Available Notations for Passing Actual Parameters

- When calling a subprogram, you can write the actual parameters using the following notations:
 - Positional: Lists the actual parameters in the same order as the formal parameters
 - Named: Lists the actual parameters in arbitrary order and uses the association operator (`=>`) to associate a named formal parameter with its actual parameter
 - Mixed: Lists some of the actual parameters as positional and some as named
- Prior to Oracle Database 11g, only the positional notation is supported in calls from SQL
- Starting in Oracle Database 11g, named and mixed notation can be used for specifying arguments in calls to PL/SQL subroutines from SQL statements

ORACLE®

1 - 29

Copyright © 2009, Oracle. All rights reserved.

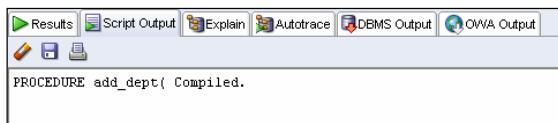
Syntax for Passing Parameters

When calling a subprogram, you can write the actual parameters using the following notations:

- **Positional:** You list the actual parameter values in the same order in which the formal parameters are declared. This notation is compact, but if you specify the parameters (especially literals) in the wrong order, the error can be hard to detect. You must change your code if the procedure's parameter list changes.
- **Named:** You list the actual values in arbitrary order and use the association operator to associate each actual parameter with its formal parameter by name. The PL/SQL **association operator** is an “equal” sign followed by an “is greater than” sign, without spaces: `=>`. The order of the parameters is not significant. This notation is more verbose, but makes your code easier to read and maintain. You can sometimes avoid changing your code if the procedure's parameter list changes, for example, if the parameters are reordered or a new optional parameter is added.
- **Mixed:** You list the first parameter values by their position and the remainder by using the special syntax of the named method. You can use this notation to call procedures that have some required parameters, followed by some optional parameters.

Passing Actual Parameters: Creating the add_dept Procedure

```
CREATE OR REPLACE PROCEDURE add_dept(
    p_name IN departments.department_name%TYPE,
    p_loc  IN departments.location_id%TYPE) IS
BEGIN
    INSERT INTO departments(department_id,
                           department_name, location_id)
    VALUES (departments_seq.NEXTVAL, p_name , p_loc );
END add_dept;
/
```



ORACLE®

1 - 30

Copyright © 2009, Oracle. All rights reserved.

Passing Parameters: Examples

In the slide example, the add_dept procedure declares two IN formal parameters: p_name and p_loc. The values of these parameters are used in the INSERT statement to set the department_name and location_id columns, respectively.

Passing Actual Parameters: Examples

```
-- Passing parameters using the positional notation.  
EXECUTE add_dept ('TRAINING', 2500)
```

anonymous block completed			
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	TRAINING		2500
1 rows selected			

```
-- Passing parameters using the named notation.  
EXECUTE add_dept (p_loc=>2400, p_name=>'EDUCATION')
```

anonymous block completed			
DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
290	EDUCATION		2400
1 rows selected			

ORACLE®

Passing Actual Parameters: Examples

Passing actual parameters by position is shown in the first call to execute `add_dept` in the first code example in the slide. The first actual parameter supplies the value TRAINING for the name formal parameter. The second actual parameter value of 2500 is assigned by position to the loc formal parameter.

Passing parameters using the named notation is shown in the second code example in the slide. The loc actual parameter, which is declared as the second formal parameter, is referenced by name in the call, where it is associated with the actual value of 2400. The name parameter is associated with the value EDUCATION. The order of the actual parameters is irrelevant if all parameter values are specified.

Note: You must provide a value for each parameter unless the formal parameter is assigned a default value. Specifying default values for formal parameters is discussed next.

Using the DEFAULT Option for the Parameters

- Defines default values for parameters
- Provides flexibility by combining the positional and named parameter-passing syntax

```
CREATE OR REPLACE PROCEDURE add_dept(
    p_name departments.department_name%TYPE := 'Unknown',
    p_loc   departments.location_id%TYPE DEFAULT 1700)
IS
BEGIN
    INSERT INTO departments (department_id,
        department_name, location_id)
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);
END add_dept;
```

```
EXECUTE add_dept
EXECUTE add_dept ('ADVERTISING', p_loc => 1200)
EXECUTE add_dept (p_loc => 1200)
```

ORACLE®

1 - 32

Copyright © 2009, Oracle. All rights reserved.

Using the DEFAULT Option for Parameters

You can assign a default value to an IN parameter as follows:

- The assignment operator (:=), as shown for the name parameter in the slide
- The DEFAULT option, as shown for the p_loc parameter in the slide

When default values are assigned to formal parameters, you can call the procedure without supplying an actual parameter value for the parameter. Thus, you can pass different numbers of actual parameters to a subprogram, either by accepting or by overriding the default values as required. It is recommended that you declare parameters without default values first. Then, you can add formal parameters with default values without having to change every call to the procedure.

Note: You cannot assign default values to the OUT and IN OUT parameters.

The second code box in the slide shows three ways of invoking the add_dept procedure:

- The first example assigns the default values for each parameter.
- The second example illustrates a combination of position and named notation to assign values. In this case, using named notation is presented as an example.
- The last example uses the default value for the name parameter, Unknown, and the supplied value for the p_loc parameter.

Using the DEFAULT Option for Parameters (continued)

The following is the result of the second slide code example in the previous slide:

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	TRAINING	2500	
290	EDUCATION	2400	
300	Unknown	1700	
310	ADVERTISING	1200	
320	Unknown	1200	
10	Administration	200	1700
20	Marketing	201	1800
30	Purchasing	114	1700
40	Human Resources	203	2400
50	Shipping	121	1500
60	IT	103	1400
70	Public Relations	204	2700
80	Sales	145	2500
90	Executive	100	1700
100	Finance	108	1700
110	Accounting	205	1700
120	Treasury		1700
130	Corporate Tax		1700
140	Control And Credit		1700
150	Shareholder Services		1700
160	Benefits		1700
170	Manufacturing		1700
180	Construction		1700
190	Contracting		1700
200	Operations		1700
210	IT Support		1700
220	NOC		1700
230	IT Helpdesk		1700
240	Government Sales		1700
250	Retail Sales		1700
260	Recruiting		1700
270	Payroll		1700
32 rows selected			

Usually, you can use named notation to override the default values of formal parameters. However, you cannot skip providing an actual parameter if there is no default value provided for a formal parameter.

Note: All the positional parameters should precede the named parameters in a subprogram call. Otherwise, you receive an error message, as shown in the following example:

```
EXECUTE add_dept(p_name=>'new dept', 'new location')
```

```
Error starting at line 1 in command:  
EXECUTE add_dept(p_name=>'new dept', 'new location')  
Error report:  
ORA-06550: line 1, column 36:  
PLS-00312: a positional parameter association may not follow a named association  
ORA-06550: line 1, column 7:  
PL/SQL: Statement ignored  
06550. 00000 - "line %s, column %s:\n%s"  
*Cause: Usually a PL/SQL compilation error.  
*Action:
```

Calling Procedures

- You can call procedures using anonymous blocks, another procedure, or packages.
- You must own the procedure or have the EXECUTE privilege.

```
CREATE OR REPLACE PROCEDURE process_employees
IS
    CURSOR cur_emp_cursor IS
        SELECT employee_id
        FROM employees;
BEGIN
    FOR emp_rec IN cur_emp_cursor
    LOOP
        raise_salary(emp_rec.employee_id, 10);
    END LOOP;
    COMMIT;
END process_employees;
/
```

PROCEDURE process_employees Compiled.

ORACLE®

1 - 34

Copyright © 2009, Oracle. All rights reserved.

Calling Procedures

You can invoke procedures by using:

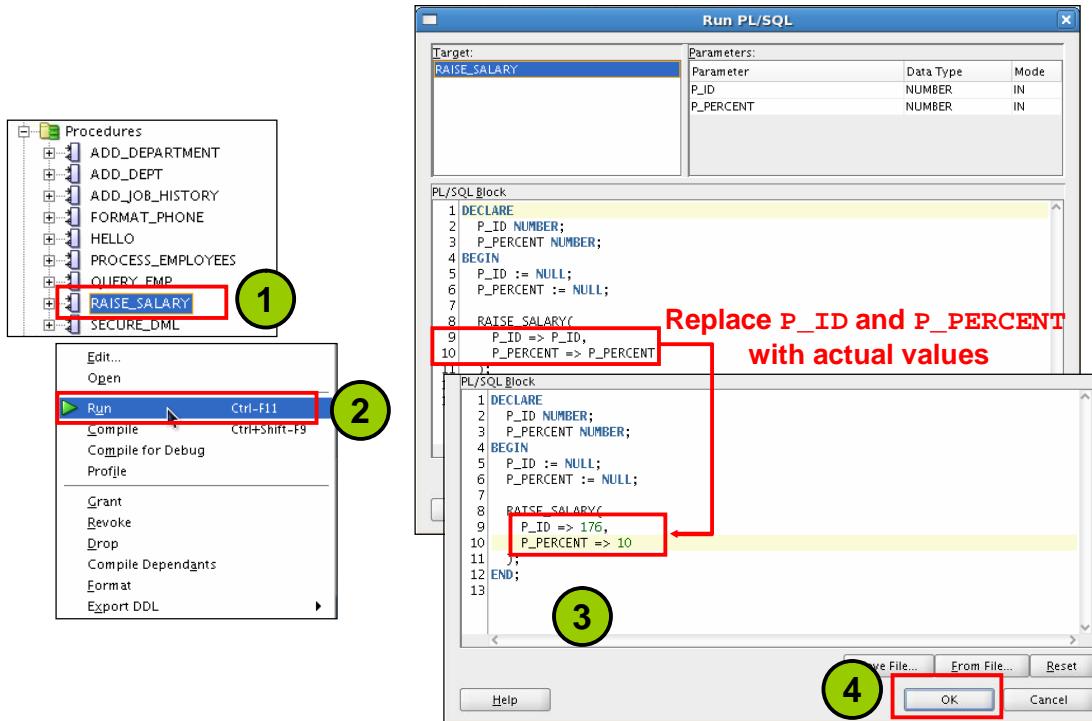
- Anonymous blocks
- Another procedure or PL/SQL subprogram

Examples on the preceding pages have illustrated how to use anonymous blocks (or the EXECUTE command in SQL Developer or SQL*Plus).

The example in the slide shows you how to invoke a procedure from another stored procedure. The PROCESS_EMPLOYEES stored procedure uses a cursor to process all the records in the EMPLOYEES table and passes each employee's ID to the RAISE_SALARY procedure, which results in a 10% salary increase across the company.

Note: You must own the procedure or have the EXECUTE privilege.

Calling Procedures Using SQL Developer



Calling Procedures Using SQL Developer

In the slide example, the `raise_salary` procedure is called to raise the current salary of employee 176 (\$ 8,600) by 10 percent as follows:

1. Right-click the procedure name in the **Procedures** node, and then click **Run**. The **Run PL/SQL** dialog box is displayed.
2. In the **PL/SQL Block** section, change the displayed formal **IN** and **IN/OUT** parameter specifications displayed after the association operator, “=>” to the *actual* values that you want to use for running or debugging the function or procedure. For example, to raise the current salary of employee 176 from 8,600 by 10 percent, you can call the `raise_salary` procedure as shown in the slide. Provide the values for the `ID` and `PERCENT` input parameters that are specified as 176 and 10 respectively. This is done by changing the displayed `ID => ID` with `ID => 176` and `PERCENT => PERCENT` with `PERCENT => 10`.
3. Click **OK**. SQL Developer runs the procedure. The updated salary of 9,460 is shown below:

Results	Script Output	Explain	Autotrace	DBMS Output	OWA Output				
Results:									
<table border="1"> <thead> <tr> <th>EMPLOYEE_ID</th><th>SALARY</th></tr> </thead> <tbody> <tr> <td>1</td><td>176</td></tr> </tbody> </table>						EMPLOYEE_ID	SALARY	1	176
EMPLOYEE_ID	SALARY								
1	176								
<table border="1"> <tbody> <tr> <td>1</td><td>176</td><td>9460</td></tr> </tbody> </table>						1	176	9460	
1	176	9460							

Lesson Agenda

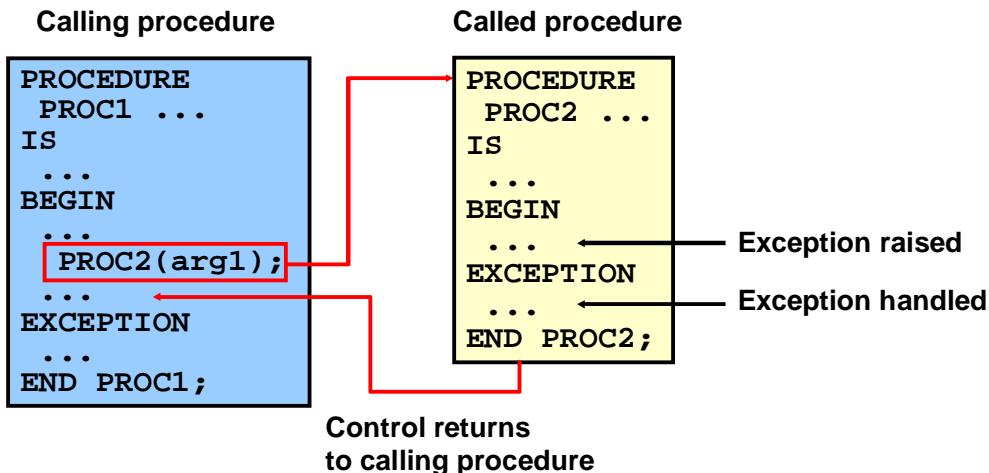
- Using a modularized and layered subprogram design and identifying the benefits of subprograms
- Working with procedures:
 - Creating and calling procedures
 - Identifying the available parameter-passing modes
 - Using formal and actual parameters
 - Using positional, named, or mixed notation
- Handling exceptions in procedures, removing a procedure, and displaying the procedures' information

ORACLE®

1 - 36

Copyright © 2009, Oracle. All rights reserved.

Handled Exceptions



ORACLE®

1 - 37

Copyright © 2009, Oracle. All rights reserved.

Handled Exceptions

When you develop procedures that are called from other procedures, you should be aware of the effects that handled and unhandled exceptions have on the transaction and the calling procedure.

When an exception is raised in a called procedure, the control immediately goes to the exception section of that block. An exception is considered handled if the exception section provides a handler for the exception raised.

When an exception occurs and is handled, the following code flow takes place:

1. The exception is raised.
2. Control is transferred to the exception handler.
3. The block is terminated.
4. The calling program/block continues to execute as if nothing has happened.

If a transaction was started (that is, if any data manipulation language [DML] statements executed before executing the procedure in which the exception was raised), then the transaction is unaffected. A DML operation is rolled back if it was performed within the procedure before the exception.

Note: You can explicitly end a transaction by executing a COMMIT or ROLLBACK operation in the exception section.

Handled Exceptions: Example

```
CREATE PROCEDURE add_department(
    p_name VARCHAR2, p_mgr NUMBER, p_loc NUMBER) IS
BEGIN
    INSERT INTO DEPARTMENTS (department_id,
        department_name, manager_id, location_id)
    VALUES (DEPARTMENTS_SEQ.NEXTVAL, p_name, p_mgr, p_loc);
    DBMS_OUTPUT.PUT_LINE('Added Dept: '|| p_name);
EXCEPTION
    WHEN OTHERS THEN
        DBMS_OUTPUT.PUT_LINE('Err: adding dept: '|| p_name);
END;
```

```
CREATE PROCEDURE create_departments IS
BEGIN
    add_department('Media', 100, 1800);
    add_department('Editing', 99, 1800); X
    add_department('Advertising', 101, 1800); ✓
END;
```

ORACLE®

1 - 38

Copyright © 2009, Oracle. All rights reserved.

Handled Exceptions: Example

The two procedures in the slide are the following:

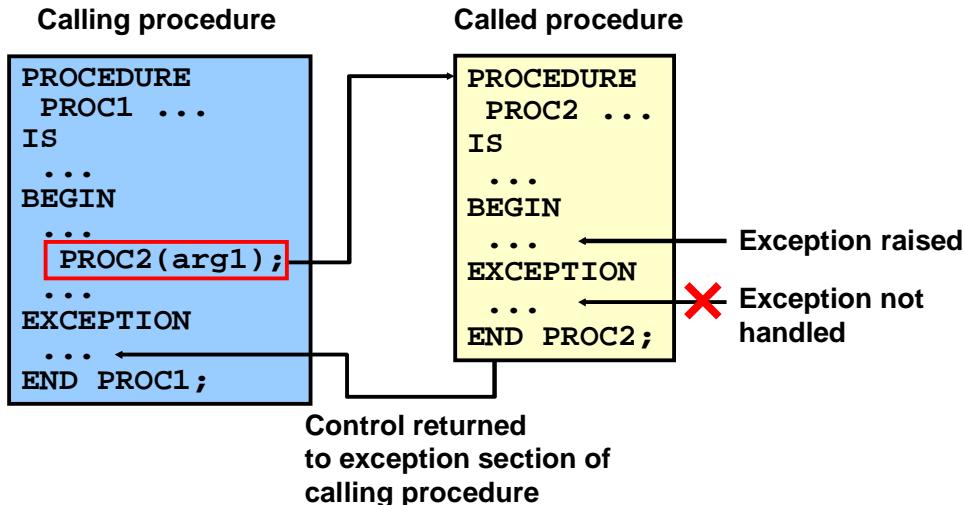
- The `add_department` procedure creates a new department record by allocating a new department number from an Oracle sequence, and sets the `department_name`, `manager_id`, and `location_id` column values using the `p_name`, `p_mgr`, and `p_loc` parameters, respectively.
- The `create_departments` procedure creates more than one department by using calls to the `add_department` procedure.

The `add_department` procedure catches all raised exceptions in its own handler. When `create_departments` is executed, the following output is generated:

```
Added Dept: Media
Err: adding dept: Editing
Added Dept: Advertising
```

The `Editing` department with a `manager_id` of 99 is not inserted because a foreign key integrity constraint violation on `manager_id` ensures that no manager has an ID of 99. Because the exception was handled in the `add_department` procedure, the `create_department` procedure continues to execute. A query on the `DEPARTMENTS` table where the `location_id` is 1800 shows that `Media` and `Advertising` are added but the `Editing` record is not.

Exceptions Not Handled



ORACLE®

Exceptions Not Handled

As discussed earlier, when an exception is raised in a called procedure, control immediately goes to the exception section of that block. If the exception section does not provide a handler for the raised exception, then it is not handled. The following code flow occurs:

1. The exception is raised.
2. The block terminates because no exception handler exists; any DML operations performed within the procedure are rolled back.
3. The exception propagates to the exception section of the calling procedure—that is, control is returned to the exception section of the calling block, if one exists.

If an exception is not handled, then all the DML statements in the calling procedure and the called procedure are rolled back along with any changes to any host variables. The DML statements that are not affected are statements that were executed before calling the PL/SQL code whose exceptions are not handled.

Exceptions Not Handled: Example

```
SET SERVEROUTPUT ON
CREATE PROCEDURE add_department_noex(
    p_name VARCHAR2, p_mgr NUMBER, p_loc NUMBER) IS
BEGIN
    INSERT INTO DEPARTMENTS (department_id,
        department_name, manager_id, location_id)
    VALUES (DEPARTMENTS_SEQ.NEXTVAL, p_name, p_mgr, p_loc);
    DBMS_OUTPUT.PUT_LINE('Added Dept: '|| p_name);
END;
```

```
CREATE PROCEDURE create_departments_noex IS
BEGIN
    add_department_noex('Media', 100, 1800);
    add_department_noex('Editing', 99, 1800);
    add_department_noex('Advertising', 101, 1800);  
X X X
END;
```

ORACLE®

1 - 40

Copyright © 2009, Oracle. All rights reserved.

Exceptions Not Handled: Example

The code example in the slide shows `add_department_noex`, which does not have an exception section. In this case, the exception occurs when the `Editing` department is added. Because of the lack of exception handling in either of the subprograms, no new department records are added into the `DEPARTMENTS` table. Executing the `create_departments_noex` procedure produces a result that is similar to the following:

```
Error starting at line 8 in command:
EXECUTE create_departments_noex
Error report:
ORA-02291: integrity constraint (ORA62.DEPT_MGR_FK) violated - parent key not found
ORA-06512: at "ORA62.ADD_DEPARTMENT_NOEX", line 4
ORA-06512: at "ORA62.CREATE_DEPARTMENTS_NOEX", line 4
ORA-06512: at line 1
02291. 00000 - "integrity constraint (%s.%s) violated - parent key not found"
*Cause: A foreign key value has no matching primary key value.
*Action: Delete the foreign key or add a matching primary key.
```

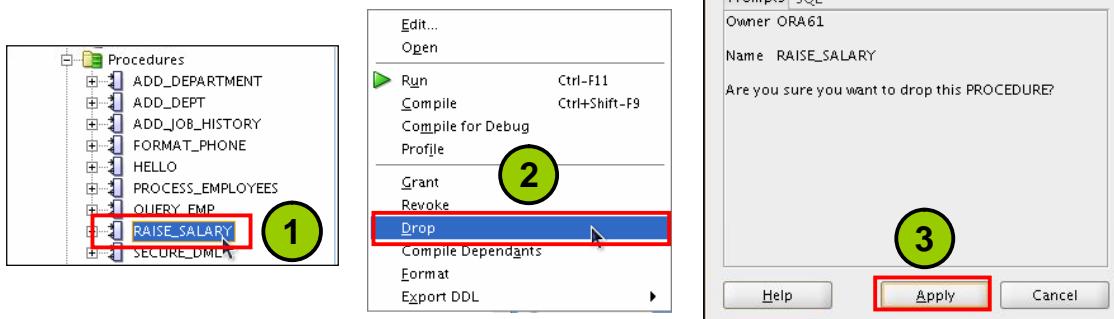
Although the results show that the `Media` department was added, its operation is rolled back because the exception was not handled in either of the subprograms invoked.

Removing Procedures: Using the DROP SQL Statement or SQL Developer

- Using the `DROP` statement:

```
DROP PROCEDURE raise_salary;
```

- Using SQL Developer:



Removing Procedures: Using the `DROP` SQL Statement or SQL Developer

When a stored procedure is no longer required, you can use the `DROP PROCEDURE` SQL statement followed by the procedure's name to remove it as follows:

```
DROP PROCEDURE procedure_name
```

You can also use SQL Developer to drop a stored procedure as follows:

- Right-click the procedure name in the **Procedures** node, and then click **Drop**. The **Drop** dialog box is displayed.
- Click **Apply** to drop the procedure.

Note

- Whether successful or not, executing a data definition language (DDL) command such as `DROP PROCEDURE` commits any pending transactions that cannot be rolled back.
- You might have to refresh the **Procedures** node before you can see the results of the drop operation. To refresh the **Procedures** node, right-click the procedure name in the **Procedures** node, and then click **Refresh**.

Viewing Procedure Information Using the Data Dictionary Views

```
DESCRIBE user_source
```

DESCRIBE user_source		
Name	Null	Type
NAME		VARCHAR2(30)
TYPE		VARCHAR2(12)
LINE		NUMBER
TEXT		VARCHAR2(4000)

4 rows selected

```
SELECT text
  FROM user_source
 WHERE name = 'ADD_DEPT' AND type = 'PROCEDURE'
 ORDER BY line;
```

TEXT
1 PROCEDURE add_dept(
2 p_name IN departments.department_name%TYPE,
3 p_loc IN departments.location_id%TYPE) IS
4
5 BEGIN
6 INSERT INTO departments(department_id, department_name, location_id)
7 VALUES (departments_seq.NEXTVAL, p_name, p_loc);
8 END add_dept;

ORACLE®

Viewing Procedure in the Data Dictionary

The source code for PL/SQL subprograms is stored in the data dictionary tables. The source code is accessible to PL/SQL procedures that are successfully or unsuccessfully compiled. To view the PL/SQL source code stored in the data dictionary, execute a SELECT statement on the following tables:

- The USER_SOURCE table to display PL/SQL code that you own
- The ALL_SOURCE table to display PL/SQL code to which you have been granted the EXECUTE right by the owner of that subprogram code

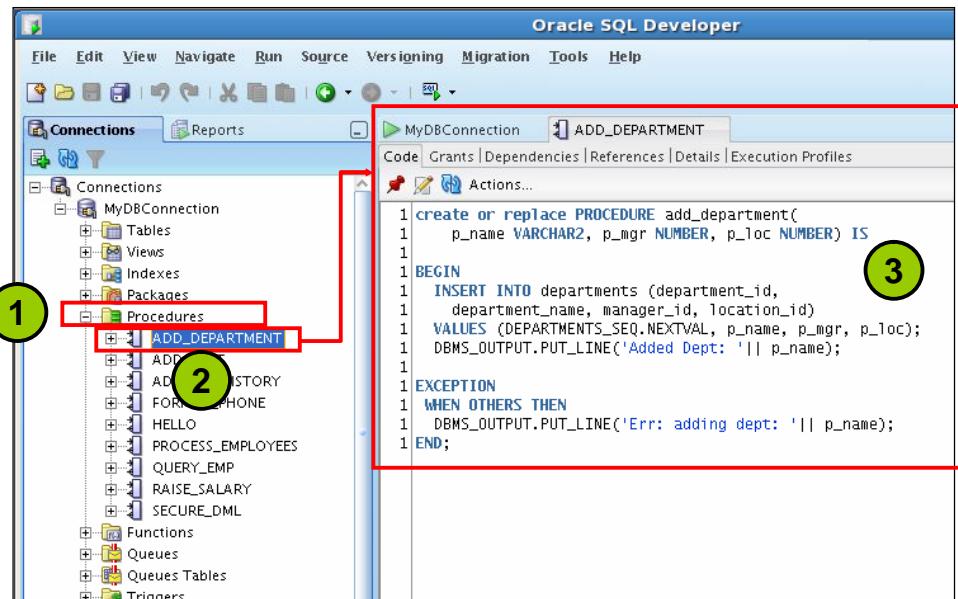
The query example shows all the columns provided by the USER_SOURCE table:

- The TEXT column holds a line of PL/SQL source code.
- The NAME column holds the name of the subprogram in uppercase text.
- The TYPE column holds the subprogram type, such as PROCEDURE or FUNCTION.
- The LINE column stores the line number for each source code line.

The ALL_SOURCE table provides an OWNER column in addition to the preceding columns.

Note: You cannot display the source code for Oracle PL/SQL built-in packages, or PL/SQL whose source code has been wrapped by using a WRAP utility. The WRAP utility converts the PL/SQL source code into a form that cannot be deciphered by humans.

Viewing Procedures Information Using SQL Developer



Viewing Procedures Information Using SQL Developer

To view a procedure's code in SQL Developer, use the following steps:

1. Click the **Procedures** node in the **Connections** tab.
2. Click the procedure's name.
3. The procedure's code is displayed in the **Code** tab as shown in the slide.

Quiz

Formal parameters are literal values, variables, and expressions used in the parameter list of the calling subprogram

1. True
2. False



Answer: 2

Formal and Actual (or arguments) Parameters

Formal parameters: Local variables declared in the parameter list of a subprogram specification.

Actual parameters: Literal values, variables, and expressions used in the parameter list of the calling subprogram.

Summary

In this lesson, you should have learned how to:

- Identify the benefits of modularized and layered subprogram design
- Create and call procedures
- Use formal and actual parameters
- Use positional, named, or mixed notation for passing parameters
- Identify the available parameter-passing modes
- Handle exceptions in procedures
- Remove a procedure
- Display the procedures' information

ORACLE®

1 - 45

Copyright © 2009, Oracle. All rights reserved.

Practice 1 Overview: Creating, Compiling, and Calling Procedures

This practice covers the following topics:

- Creating stored procedures to:
 - Insert new rows into a table using the supplied parameter values
 - Update data in a table for rows that match the supplied parameter values
 - Delete rows from a table that match the supplied parameter values
 - Query a table and retrieve data based on supplied parameter values
- Handling exceptions in procedures
- Compiling and invoking procedures

ORACLE®

1 - 46

Copyright © 2009, Oracle. All rights reserved.

Practice 1: Overview

In this practice, you create, compile, and invoke procedures that issue DML and query commands. You also learn how to handle exceptions in procedures.

If you encounter compilation errors when you execute procedures, you can use the Compiler-Log tab in SQL Developer.

Note: It is recommended to use SQL Developer for this practice.

Important

All practices in this course and the practice solutions assume that you create objects such as procedures, functions, and so on using the SQL Worksheet area in SQL Developer. When you create an object in the SQL Worksheet area, you need to refresh the object node in order for the new object to be displayed in the Navigator tree. To compile the newly created object, you can right-click the object name in the Navigator tree, and then select Compile from the shortcut menu. For example, after you enter the code to create a procedure in the SQL Worksheet area, you click the Run Script icon (or press F5) to run the code. This creates and compiles the procedure.

Alternatively, you can create objects such as procedures using the PROCEDURES node in the Navigator tree, and then compile the procedure. Creating objects using the Navigator tree automatically displays the newly created object.

Creating Functions and Debugging Subprograms

Copyright © 2009, Oracle. All rights reserved.

ORACLE®

Objectives

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

- Differentiate between a procedure and a function
- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function
- Understand the basic functionality of the SQL Developer debugger



Lesson Aim

In this lesson, you learn how to create, invoke, and maintain functions.

Lesson Agenda

- Working with functions:
 - Differentiating between a procedure and a function
 - Describing the uses of functions
 - Creating, invoking, and removing stored functions
- Introducing the SQL Developer debugger

ORACLE®

2 - 3

Copyright © 2009, Oracle. All rights reserved.

Overview of Stored Functions

A function:

- Is a named PL/SQL block that returns a value
- Can be stored in the database as a schema object for repeated execution
- Is called as part of an expression or is used to provide a parameter value for another subprogram
- Can be grouped into PL/SQL packages



Overview of Stored 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.

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.

Functions (and procedures) can be grouped into PL/SQL packages. Packages make code even more reusable and maintainable. Packages are covered in the lessons titled “Creating Packages” and “Working with Packages.”

Creating Functions

The PL/SQL block must have at least one RETURN statement.

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

PL/SQL Block

Syntax for Creating Functions

A function is a PL/SQL block that returns a value. A RETURN statement must be provided to return a value with a data type that is consistent with the function declaration.

You create new functions with the CREATE FUNCTION statement, which may declare a list of parameters, must return one value, and must define the actions to be performed by the standard PL/SQL block.

You should consider the following points about the CREATE FUNCTION statement:

- The REPLACE option indicates that if the function exists, it is dropped and replaced with the new version that is created by the statement.
- The RETURN data type must not include a size specification.
- The PL/SQL block starts with a BEGIN after the declaration of any local variables and ends with an END, optionally followed by the *function_name*.
- There must be at least one RETURN *expression* statement.
- You cannot reference host or bind variables in the PL/SQL block of a stored function.

Note: Although the OUT and IN OUT parameter modes can be used with functions, it is not good programming practice to use them with functions. However, if you need to return more than one value from a function, consider returning the values in a composite data structure such as a PL/SQL record or a PL/SQL table.

The Difference Between Procedures and Functions

Procedures	Functions
Execute as a PL/SQL statement	Invoke as part of an expression
Do not contain RETURN clause in the header	Must contain a RETURN clause in the header
Can pass values (if any) using output parameters	Must return a single value
Can contain a RETURN statement without a value	Must contain at least one RETURN statement

ORACLE®

2 - 6

Copyright © 2009, Oracle. All rights reserved.

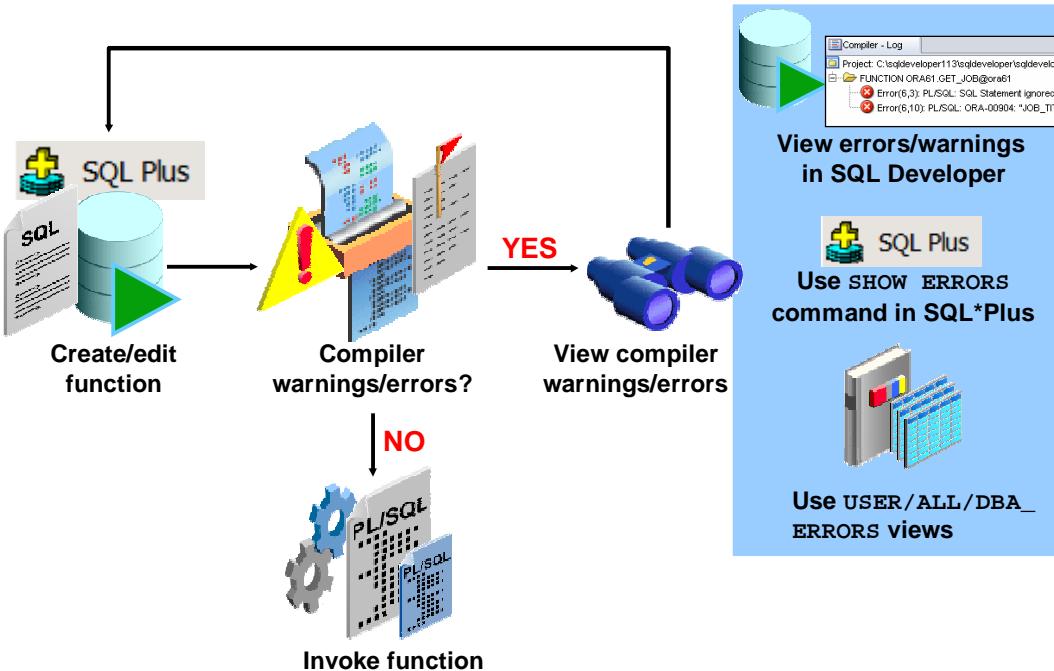
The Difference Between Procedures and Functions

You create a procedure to store a series of actions for later execution. A procedure can contain zero or more parameters that can be transferred to and from the calling environment, but a procedure does not have to return a value. A procedure can call a function to assist with its actions.

Note: A procedure containing a single OUT parameter would be better rewritten as a function returning the value.

You create a function when you want to compute a value that must be returned to the calling environment. A function can contain zero or more parameters that are transferred from the calling environment. Functions typically return only a single value, and the value is returned through a RETURN statement. Functions used in SQL statements should not use OUT or IN OUT mode parameters. Although a function using output parameters can be used in a PL/SQL procedure or block, it cannot be used in SQL statements.

Creating and Running Functions: Overview



Creating and Running Functions: Overview

The diagram in the slide illustrates the basic steps involved in creating and running a function:

1. Create the function using SQL Developer's Object Navigator tree or the SQL Worksheet area.
2. Compile the function. The function is created in the database. The CREATE FUNCTION statement creates and stores source code and the compiled *m-code* in the database. To compile the function, right-click the function's name in the Object Navigator tree, and then click Compile.
3. If there are compilation warning or errors, you can view (and then correct) the warnings or errors using one of the following methods:
 - a. Using the SQL Developer interface (the Compiler – Log tab)
 - b. Using the SHOW ERRORS SQL*Plus command
 - c. Using the USER/ALL/DBA_ERRORS views
4. After successful compilation, invoke the function to return the desired value.

Creating and Invoking a Stored Function Using the CREATE FUNCTION Statement: Example

```
CREATE OR REPLACE FUNCTION get_sal
(p_id employees.employee_id%TYPE) RETURN NUMBER IS
v_sal employees.salary%TYPE := 0;
BEGIN
  SELECT salary
  INTO v_sal
  FROM employees
  WHERE employee_id = p_id;
  RETURN v_sal;
END get_sal;
/
```

FUNCTION get_sal Compiled.

```
-- Invoke the function as an expression or as
-- a parameter value.
```

```
EXECUTE dbms_output.put_line(get_sal(100))
```

anonymous block completed
24000

ORACLE®

2 - 8

Copyright © 2009, Oracle. All rights reserved.

Stored Function: Example

The get_sal function is created with a single input parameter and returns the salary as a number. Execute the command as shown, or save it in a script file and run the script to create the get_sal function.

The get_sal function follows a common programming practice of using a single RETURN statement that returns a value assigned to a local variable. If your function has an exception section, then it may also contain a RETURN statement.

Invoke a function as part of a PL/SQL expression because the function will return 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 function get_sal. In this case, get_sal is invoked first to calculate the salary of the employee with ID 100. The salary value returned is supplied as the value of the DBMS_OUTPUT.PUT_LINE parameter, which displays the result (if you have executed a SET SERVEROUTPUT ON).

Note: 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.

Using Different Methods for Executing Functions

```
-- As a PL/SQL expression, get the results using host variables  
  
VARIABLE b_salary NUMBER  
EXECUTE :b_salary := get_sal(100)
```

```
anonymous block completed  
b_salary  
-----  
24000
```

```
-- As a PL/SQL expression, get the results using a local  
-- variable  
SET SERVEROUTPUT ON  
DECLARE  
    sal employees.salary%type;  
BEGIN  
    sal := get_sal(100);  
    DBMS_OUTPUT.PUT_LINE('The salary is: '|| sal);  
END;  
/
```

```
anonymous block completed  
The salary is: 24000
```

ORACLE®

2 - 9

Copyright © 2009, Oracle. All rights reserved.

Using Different Methods for Executing Functions

If functions are well designed, 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.

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

Using Different Methods for Executing Functions

```
-- Use as a parameter to another subprogram  
  
EXECUTE dbms_output.put_line(get_sal(100))
```

```
anonymous block completed  
24000
```

```
-- Use in a SQL statement (subject to restrictions)  
  
SELECT job_id, get_sal(employee_id)  
FROM employees;
```

```
JOB_ID      GET_SAL(EMPLOYEE_ID)  
-----  
SH_CLERK    2600  
SH_CLERK    2600  
AD_ASST     4400  
MK_MAN      13000
```

```
...  
  
SH_CLERK    3100  
SH_CLERK    3000
```

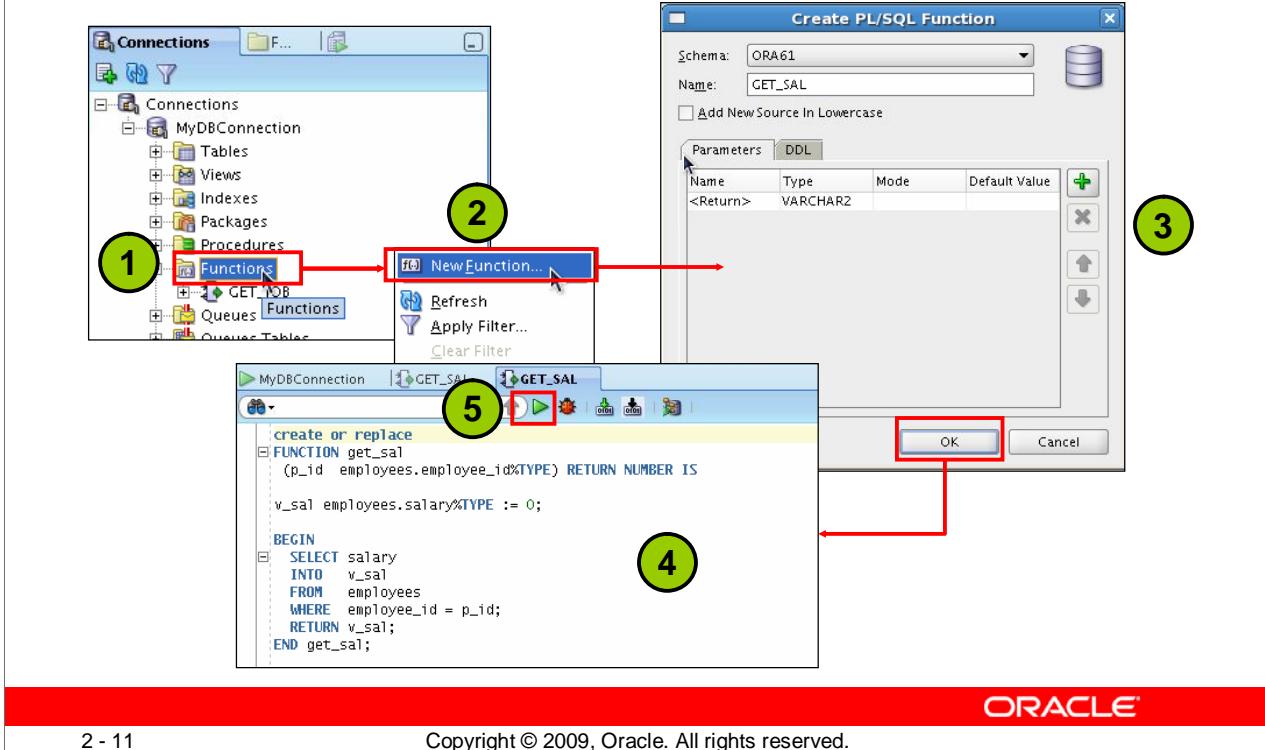
```
107 rows selected
```

ORACLE®

Using Different Methods for Executing Functions (continued)

- **As a parameter to another subprogram:** The first example in the slide demonstrates this usage. The `get_sal` 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 course titled *Oracle Database 11g: SQL Fundamentals I*.
- **As an expression in a SQL statement:** The second example in the slide shows how a function can be used as a single-row function in a SQL statement.

Creating and Compiling Functions Using SQL Developer



Creating and Compiling Functions Using SQL Developer

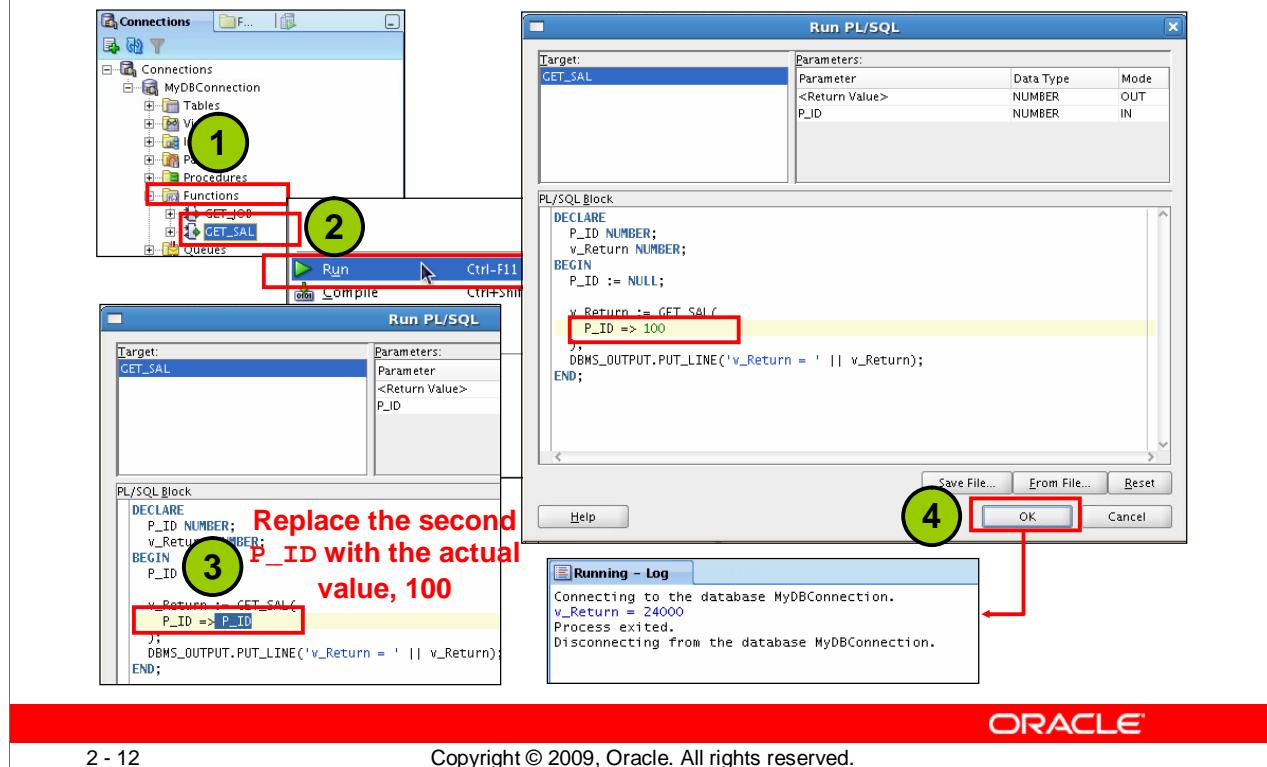
You can create a new function in SQL Developer using the following steps:

1. Right-click the **Functions** node.
2. Select **New Function** from the shortcut menu. The **Create PL/SQL Function** dialog box is displayed.
3. Select the schema, function name, and the parameters list (using the + icon), and then click **OK**. The code editor for the function is displayed.
4. Enter the function's code.
5. To compile the function, click the **Compile** icon.

Note

- To create a new function in SQL Developer, you can also enter the code in the SQL Worksheet, and then click the Run Script icon.
- For additional information about creating functions in SQL Developer, access the appropriate online help topic titled “Create PL/SQL Subprogram Function or Procedure.”

Executing Functions Using SQL Developer



2 - 12

Copyright © 2009, Oracle. All rights reserved.

Executing Functions Using SQL Developer

You can execute a function in SQL Developer using the following steps:

1. Click the **Functions** node.
2. Right-click the function's name, and then select **Run**. The **Run PL/SQL** dialog box is displayed.
3. Replace the second parameter name with the actual parameter value as shown in the slide example.
4. Click **OK**.

Note: For additional information about running functions in SQL Developer, access the online help topic titled “Running and Debugging Functions and Procedures.”

Advantages of User-Defined Functions in SQL Statements

- Can extend SQL where activities are too complex, too awkward, or unavailable with SQL
- Can increase efficiency when used in the WHERE clause to filter data, as opposed to filtering the data in the application
- Can manipulate data values

ORACLE®

2 - 13

Copyright © 2009, Oracle. All rights reserved.

Advantages of User-Defined Functions in SQL Statements

SQL statements can reference PL/SQL user-defined functions anywhere a SQL expression is allowed. For example, a user-defined function can be used anywhere that a built-in SQL function, such as UPPER(), can be placed.

Advantages

- Permits calculations that are too complex, awkward, or unavailable with SQL. Functions increase data independence by processing complex data analysis within the Oracle server, rather than by retrieving the data into an application
- Increases efficiency of queries by performing functions in the query rather than in the application
- Manipulates new types of data (for example, latitude and longitude) by encoding character strings and using functions to operate on the strings

Using a Function in a SQL Expression: Example

```
CREATE OR REPLACE FUNCTION tax(p_value IN NUMBER)
  RETURN NUMBER IS
BEGIN
  RETURN (p_value * 0.08);
END tax;
/
SELECT employee_id, last_name, salary, tax(salary)
FROM   employees
WHERE  department_id = 100;
```

FUNCTION tax(value Compiled.			
EMPLOYEE_ID	LAST_NAME	SALARY	TAX(SALARY)
108	Greenberg	12000	960
109	Faviet	9000	720
110	Chen	8200	656
111	Sciarrra	7700	616
112	Urman	7800	624
113	Popp	6900	552

6 rows selected

ORACLE®

2 - 14

Copyright © 2009, Oracle. All rights reserved.

Function in SQL Expressions: Example

The example in the slide shows how to create a `tax` function to calculate income tax. The function accepts a `NUMBER` parameter and returns the calculated income tax based on a simple flat tax rate of 8%.

To execute the code shown in the slide example in SQL Developer, enter the code in the SQL Worksheet, and then click the **Run Script** icon. The `tax` function is invoked as an expression in the `SELECT` clause along with the employee ID, last name, and salary for employees in a department with ID 100. The return result from the `tax` function is displayed with the regular output from the query.

Calling User-Defined Functions in SQL Statements

User-defined functions act like built-in single-row functions and can be used in:

- The SELECT list or clause of a query
- Conditional expressions of the WHERE and HAVING clauses
- The CONNECT BY, START WITH, ORDER BY, and GROUP BY clauses of a query
- The VALUES clause of the INSERT statement
- The SET clause of the UPDATE statement

ORACLE®

2 - 15

Copyright © 2009, Oracle. All rights reserved.

Calling User-Defined Functions in SQL Statements

A PL/SQL user-defined function can be called from any SQL expression where a built-in single-row function can be called as shown in the following example:

```
SELECT employee_id, tax(salary)
  FROM employees
 WHERE tax(salary) > (SELECT MAX(tax(salary))
                           FROM employees
                          WHERE department_id = 30)
 ORDER BY tax(salary) DESC;
```

EMPLOYEE_ID	TAX(SALARY)
100	1920
101	1360
102	1360
145	1120
146	1080
201	1040
205	960
147	960
108	960
168	920

10 rows selected

Restrictions When Calling Functions from SQL Expressions

- User-defined functions that are callable from SQL expressions must:
 - Be stored in the database
 - Accept only `IN` parameters with valid SQL data types, not PL/SQL-specific types
 - Return valid SQL data types, not PL/SQL-specific types
- When calling functions in SQL statements:
 - You must own the function or have the `EXECUTE` privilege
 - You may need to enable the `PARALLEL_ENABLE` keyword to allow a parallel execution of the SQL statement

ORACLE®

2 - 16

Copyright © 2009, Oracle. All rights reserved.

Restrictions When 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 `IN` and of 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.
- You may need to enable the `PARALLEL_ENABLE` keyword to allow a parallel execution of the SQL statement using the function. Each parallel slave will have private copies of the function's local variables.

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. In addition, it cannot be used to specify a default value for a column. Only stored functions are callable from SQL statements. Stored procedures cannot be called unless invoked from a function that meets the preceding requirements.

Controlling Side Effects When 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 - 17

Copyright © 2009, Oracle. All rights reserved.

Controlling Side Effects When Calling Functions from SQL Expressions

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

Additional restrictions apply when a function is called in expressions of SQL statements:

- When a function is called from a SELECT statement or a parallel UPDATE or DELETE statement, the function cannot modify database tables.
- When a function is called from an UPDATE or DELETE statement, the function cannot query or modify database tables modified by that statement.
- When a function is called from a SELECT, INSERT, UPDATE, or DELETE statement, the function cannot execute directly or indirectly through another subprogram or SQL transaction control statements such as:
 - A COMMIT or ROLLBACK statement
 - A session control statement (such as SET ROLE)
 - A system control statement (such as ALTER SYSTEM)
 - Any DDL statements (such as CREATE) because they are followed by an automatic commit

Restrictions on Calling Functions from SQL: Example

```
CREATE OR REPLACE FUNCTION dml_call_sql(p_sal NUMBER)
  RETURN NUMBER IS
BEGIN
  INSERT INTO employees(employee_id, last_name,
                        email, hire_date, job_id, salary)
  VALUES(1, 'Frost', 'jfrost@company.com',
         SYSDATE, 'SA_MAN', p_sal);
  RETURN (p_sal + 100);
END;
```

```
UPDATE employees
  SET salary = dml_call_sql(2000)
 WHERE employee_id = 170;
```

```
FUNCTION dml_call_sql(p_sal Compiled.
Error starting at line 1 in command:
UPDATE employees
  SET salary = dml_call_sql(2000)
 WHERE employee_id = 170
Error report:
SQL Error: ORA-04091: table ORA62.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "ORA62.DML_CALL_SQL", line 4
04091. 00000 - "table %s.%s is mutating, trigger/function may not see it"
*Cause: A trigger (or a user defined plsql function that is referenced in
this statement) attempted to look at (or modify) a table that was
in the middle of being modified by the statement which fired it.
*Action: Rewrite the trigger (or function) so it does not read that table.
```

ORACLE®

2 - 18

Copyright © 2009, Oracle. All rights reserved.

Restrictions on Calling Functions from SQL: Example

The `dml_call_sql` function in the slide contains an `INSERT` statement that inserts a new record into the `EMPLOYEES` table and returns the input salary value incremented by 100. This function is invoked in the `UPDATE` statement that modifies the salary of employee 170 to the amount returned from the function. The `UPDATE` statement fails with an error indicating that the table is mutating (that is, changes are already in progress in the same table). In the following example, the `query_call_sql` function queries the `SALARY` column of the `EMPLOYEES` table:

```
CREATE OR REPLACE FUNCTION query_call_sql(p_a NUMBER)
  RETURN NUMBER IS
  v_s NUMBER;
BEGIN
  SELECT salary INTO v_s FROM employees
  WHERE employee_id = 170;
  RETURN (v_s + p_a);
END;
```

When invoked from the following `UPDATE` statement, it returns the error message similar to the error message shown in the slide:

```
UPDATE employees SET salary = query_call_sql(100)
WHERE employee_id = 170;
```

Named and Mixed Notation from SQL

- PL/SQL allows arguments in a subroutine call to be specified using positional, named, or mixed notation.
- Prior to Oracle Database 11g, only the positional notation is supported in calls from SQL.
- Starting in Oracle Database 11g, named and mixed notation can be used for specifying arguments in calls to PL/SQL subroutines from SQL statements.
- For long parameter lists, with most having default values, you can omit values from the optional parameters.
- You can avoid duplicating the default value of the optional parameter at each call site.

ORACLE®

Named and Mixed Notation from SQL: Example

```
CREATE OR REPLACE FUNCTION f(
    p_parameter_1 IN NUMBER DEFAULT 1,
    p_parameter_5 IN NUMBER DEFAULT 5)
RETURN NUMBER
IS
    v_var number;
BEGIN
    v_var := p_parameter_1 + (p_parameter_5 * 2);
    RETURN v_var;
END f;
/
```

FUNCTION f() Compiled.

```
SELECT f(p_parameter_5 => 10) FROM DUAL;
```

```
F(P_PARAMETER_5=>10)
-----
21
1 rows selected
```

ORACLE®

2 - 20

Copyright © 2009, Oracle. All rights reserved.

Example of Using Named and Mixed Notation from a SQL Statement

In the example in the slide, the call to the function f within the SQL SELECT statement uses the named notation. Before Oracle Database 11g, you could not use the named or mixed notation when passing parameters to a function from within a SQL statement. Before Oracle Database 11g, you received the following error:

```
SELECT f(p_parameter_5 => 10) FROM DUAL;
```

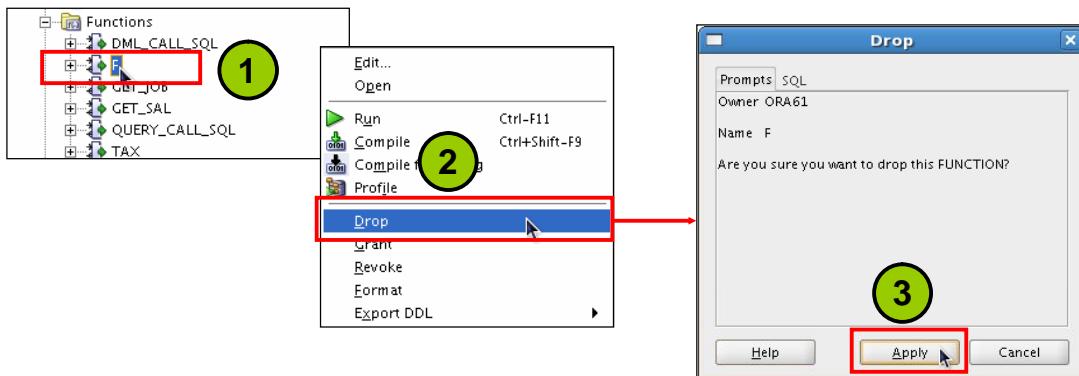
ORA-00907: missing right parenthesis

Removing Functions: Using the DROP SQL Statement or SQL Developer

- Using the **DROP statement**:

```
DROP FUNCTION f;
```

- Using **SQL Developer**:



Removing Functions

Using the **DROP** statement

When a stored function is no longer required, you can use a SQL statement in SQL*Plus to drop it. To remove a stored function by using SQL*Plus, execute the **DROP FUNCTION** SQL command.

Using **CREATE OR REPLACE** Versus **DROP** and **CREATE**

The **REPLACE** clause in the **CREATE OR REPLACE** syntax is equivalent to dropping a function and re-creating it. When you use the **CREATE OR REPLACE** syntax, the privileges granted on this object to other users remain the same. When you **DROP** a function and then re-create it, all the privileges granted on this function are automatically revoked.

Using **SQL Developer**

To drop a function in SQL Developer, right-click the function name in the **Functions** node, and then select **Drop**. The **Drop** dialog box is displayed. To drop the function, click **Apply**.

Viewing Functions Using Data Dictionary Views

```
DESCRIBE USER_SOURCE
```

Name	Null	Type
NAME		VARCHAR2(30)
TYPE		VARCHAR2(12)
LINE		NUMBER
TEXT		VARCHAR2(4000)

4 rows selected

```
SELECT text
  FROM user_source
 WHERE type = 'FUNCTION'
 ORDER BY line;
```

TEXT
1 FUNCTION tax(p_value IN NUMBER)
2 FUNCTION query_call_sql(p_a NUMBER) RETURN NUMBER IS
3 FUNCTION get_sal
4 FUNCTION dml_call_sql(p_sal NUMBER)
5 RETURN NUMBER IS
6 RETURN NUMBER IS
7 (p_id employees.employee_id%TYPE) RETURN NUMBER IS
8 v_s NUMBER;

ORACLE®

2 - 22

Copyright © 2009, Oracle. All rights reserved.

Viewing Functions Using Data Dictionary Views

The source code for PL/SQL functions is stored in the data dictionary tables. The source code is accessible for PL/SQL functions that are successfully or unsuccessfully compiled. To view the PL/SQL function code stored in the data dictionary, execute a SELECT statement on the following tables where the TYPE column value is FUNCTION:

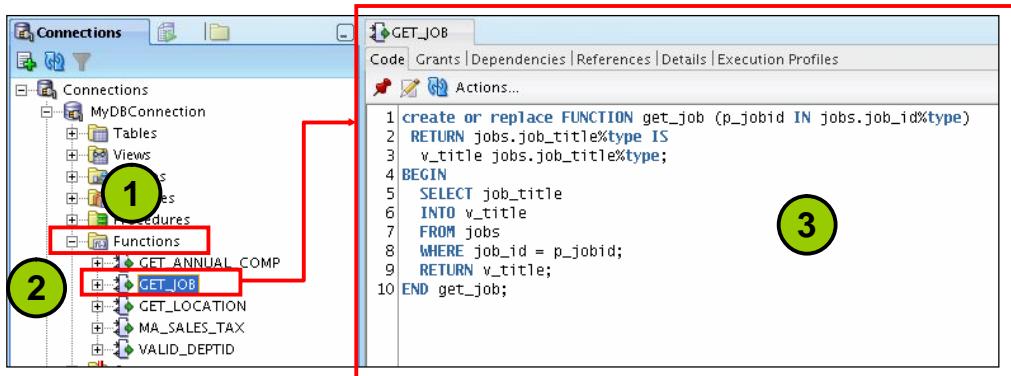
- The USER_SOURCE table to display the PL/SQL code that you own
- The ALL_SOURCE table to display the PL/SQL code to which you have been granted the EXECUTE right by the owner of that subprogram code

The second example in the slide uses the USER_SOURCE table to display the source code for all the functions in your schema.

You can also use the USER_OBJECTS data dictionary view to display a list of your function names.

Note: The output of the second code example in the slide was generated using the Execute Statement (F9) icon on the toolbar to provide better-formatted output.

Viewing Functions Information Using SQL Developer



Viewing Functions Information Using SQL Developer

To view a function's code in SQL Developer, use the following steps:

1. Click the **Functions** node in the **Connections** tab.
2. Click the function's name.
3. The function's code is displayed in the **Code** tab as shown in the slide.

Quiz

A PL/SQL stored function:

1. Can be invoked as part of an expression
2. Must contain a RETURN clause in the header
3. Must return a single value
4. Must contain at least one RETURN statement
5. Does not contain a RETURN clause in the header



Answers: 1, 2, 3, 4

Practice 2-1: Overview

This practice covers the following topics:

- Creating stored functions:
 - To query a database table and return specific values
 - To be used in a SQL statement
 - To insert a new row, with specified parameter values, into a database table
 - Using default parameter values
- Invoking a stored function from a SQL statement
- Invoking a stored function from a stored procedure

ORACLE®

2 - 25

Copyright © 2009, Oracle. All rights reserved.

Practice 2-1: Overview

It is recommended to use SQL Developer for this practice.

Lesson Agenda

- Working with functions:
 - Differentiating between a procedure and a function
 - Describing the uses of functions
 - Creating, invoking, and removing stored functions
- Introducing the SQL Developer debugger

ORACLE®

2 - 26

Copyright © 2009, Oracle. All rights reserved.

Debugging PL/SQL Subprograms Using the SQL Developer Debugger

- You can use the debugger to control the execution of your PL/SQL program.
- To debug a PL/SQL subprogram, a *security administrator* needs to grant the following privileges to the application developer:
 - DEBUG ANY PROCEDURE
 - DEBUG CONNECT SESSION

```
GRANT DEBUG ANY PROCEDURE TO ora61;
GRANT DEBUG CONNECT SESSION TO ora61;
```

ORACLE®

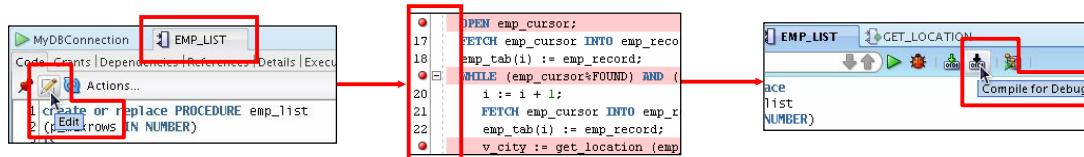
2 - 27

Copyright © 2009, Oracle. All rights reserved.

Moving Through Code While Debugging

The SQL Developer debugger enables you to control the execution of your program. You can control whether your program executes a single line of code, an entire subprogram (procedure or function), or an entire program block. By manually controlling when the program should run and when it should pause, you can quickly move over the sections that you know work correctly and concentrate on the sections that are causing problems.

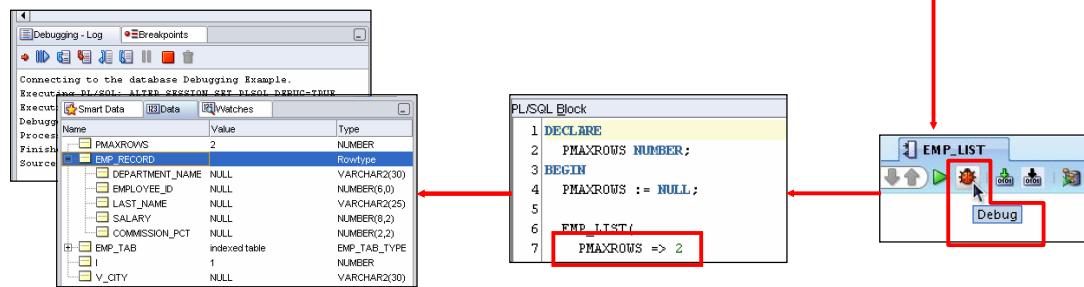
Debugging a Subprogram: Overview



1. Edit procedure

2. Add breakpoints

3. Compile for Debug

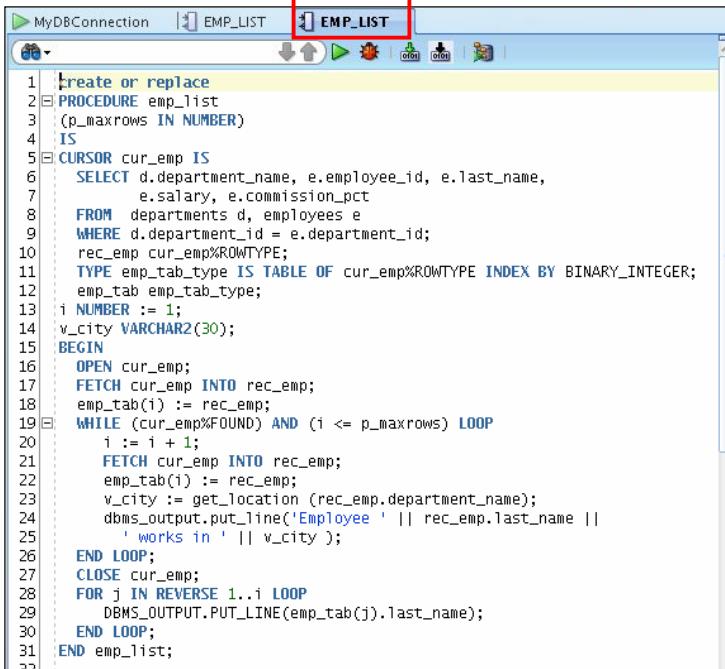


6. Choose debugging tool, and monitor data

5. Enter parameter value(s)

4. Debug

The Procedure or Function Code Editing Tab



A screenshot of the Oracle SQL Developer interface. The title bar shows 'MyDBConnection' and the tab 'EMP_LIST'. The main area contains PL/SQL code for a procedure named 'emp_list'. The code uses a cursor to select employee details from the 'employees' and 'departments' tables, stores the results in a table type 'emp_tab_type', and then loops through the results to output each employee's name and department location. A red box highlights the 'EMP_LIST' tab in the top navigation bar.

```
1 | create or replace
2 | PROCEDURE emp_list
3 | (p_maxrows IN NUMBER)
4 | IS
5 | CURSOR cur_emp IS
6 |   SELECT d.department_name, e.employee_id, e.last_name,
7 |         e.salary, e.commission_pct
8 |   FROM departments d, employees e
9 |   WHERE d.department_id = e.department_id;
10|   rec_emp cur_emp%ROWTYPE;
11|   TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
12|   emp_tab emp_tab_type;
13|   i NUMBER := 1;
14|   v_city VARCHAR2(30);
15| BEGIN
16|   OPEN cur_emp;
17|   FETCH cur_emp INTO rec_emp;
18|   emp_tab(i) := rec_emp;
19|   WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
20|     i := i + 1;
21|     FETCH cur_emp INTO rec_emp;
22|     emp_tab(i) := rec_emp;
23|     v_city := get_location (rec_emp.department_name);
24|     dbms_output.put_line('Employee ' || rec_emp.last_name ||
25|       ' works in ' || v_city );
26|   END LOOP;
27|   CLOSE cur_emp;
28|   FOR j IN REVERSE 1..i LOOP
29|     DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
30|   END LOOP;
31| END emp_list;
32|
```

ORACLE®

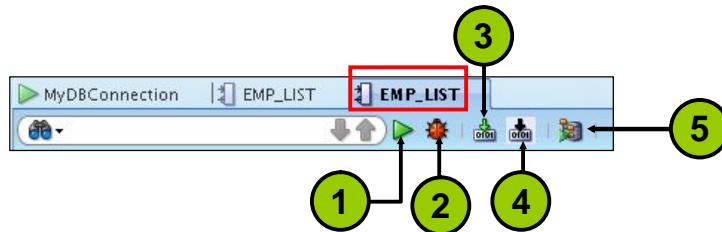
2 - 29

Copyright © 2009, Oracle. All rights reserved.

The Procedure or Function Code Tab

This tab displays a toolbar and the text of the subprogram, which you can edit. You can set and unset breakpoints for debugging by clicking to the left of the thin vertical line beside each statement with which you want to associate a breakpoint. (When a breakpoint is set, a red circle is displayed.)

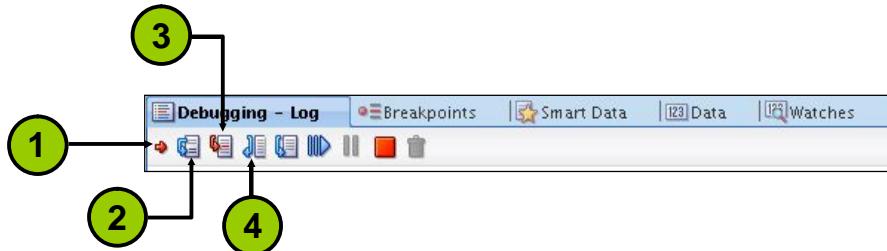
The Procedure or Function Tab Toolbar



Icon	Description
1. Run	Starts normal execution of the function or procedure, and displays the results in the Running - Log tab
2. Debug	Executes the subprogram in debug mode, and displays the Debugging - Log tab, which includes the debugging toolbar for controlling execution
3. Compile	Compiles the subprogram
4. Compile for Debug	Compiles the subprogram so that it can be debugged
5. Profile	Displays the Profile window that you use to specify parameter values for running, debugging, or profiling a PL/SQL function or procedure

ORACLE®

The Debugging – Log Tab Toolbar



Icon	Description
1. Find Execution Point	Goes to the next execution point
2. Step Over	Bypasses the next subprogram and goes to the next statement after the subprogram
3. Step Into	Executes a single program statement at a time. If the execution point is located on a call to a subprogram, it steps into the first statement in that subprogram
4. Step Out	Leaves the current subprogram and goes to the next statement with a breakpoint

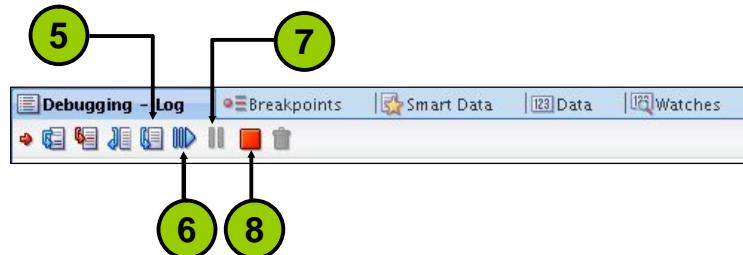
ORACLE®

The Debugging – Log Tab Toolbar

The Debugging - Log contains the debugging toolbar and informational messages.

1. **Find Execution Point:** Goes to the execution point (the next line of source code to be executed by the debugger)
2. **Step Over:** Bypasses the next subprogram (unless the subprogram has a breakpoint) and goes to the next statement after the subprogram. If the execution point is located on a subprogram call, it runs that subprogram without stopping (instead of stepping into it), and then positions the execution point on the statement that follows the call. If the execution point is located on the last statement of a subprogram, Step Over returns from the subprogram, placing the execution point on the line of code that follows the call to the subprogram from which you are returning.
3. **Step Into:** Executes a single program statement at a time. If the execution point is located on a call to a subprogram, Step Into steps into that subprogram and places the execution point on its first statement. If the execution point is located on the last statement of a subprogram, Step Into returns from the subprogram, placing the execution point on the line of code that follows the call to the subprogram from which you are returning.
4. **Step Out:** Leaves the current subprogram and goes to the next statement

The Debugging – Log Tab Toolbar



Icon	Description
5. Step to End of Method	Goes to the last statement of the current subprogram
6. Resume	Continues execution
7. Pause	Halts execution but does not exit
8. Terminate	Halts and exits the execution

ORACLE®

The Debugging – Log Tab Toolbar (continued)

5. **Step to End of Method:** Goes to the last statement of the current subprogram
6. **Resume:** Continues execution
7. **Pause:** Halts execution but does not exit, thus allowing you to resume execution
8. **Terminate:** Halts and exits the execution. You cannot resume execution from this point; instead, to start running or debugging from the beginning of the subprogram, click the Run or Debug icon in the Source tab toolbar.

Additional Tabs

Debugging - Log		
Breakpoints		
Smart Data		
Data		
Watches		
Name	Value	Type
P_MAXROWS	100	NUMBER
REC_EMP		Rowtype
EMP_TAB	indexed table	EMP_TAB_TYPE
I	1	NUMBER
V_CITY	NULL	VARCHAR2(30)

Tab	Description
Breakpoints	Displays breakpoints, both system-defined and user-defined.
Smart Data	Displays information about variables. You can specify these preferences by right-clicking in the Smart Data window and selecting Preferences.
Data	Located under the code text area; displays information about all variables
Watches	Located under the code text area; displays information about watches

ORACLE®

Additional Tabs

Setting Expression Watches

A watch enables you to monitor the changing values of variables or expressions as your program runs. After you enter a watch expression, the Watches 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.

A watch evaluates an expression according to the current context, which is controlled by the selection in the Stack window. If you move to a new context, the expression is reevaluated for the new context. If the execution point moves to a location where any of the variables in the watch expression are undefined, the entire watch expression becomes undefined. If the execution point returns to a location where the watch expression can be evaluated, the Watches window again displays the value of the watch expression.

To open the Watches window, click View, then Debugger, then Watches.

To add a watch, right-click in the Watches window and select Add Watch. To edit a watch, right-click in the Watches window and select Edit Watch.

Note: If you cannot see some of the debugging tabs described in this lesson, you can re-display such tabs by using the View > Debugger menu option.

Debugging a Procedure Example: Creating a New emp_list Procedure

```
1 CREATE OR REPLACE PROCEDURE emp_list(pmaxrows IN NUMBER) AS
2 CURSOR emp_cursor IS
3 SELECT d.department_name,
4       e.employee_id,
5       e.last_name,
6       e.salary,
7       e.commission_pct
8 FROM departments d,
9      employees e
10 WHERE d.department_id = e.department_id;
11 emp_record emp_cursor % rowtype;
12 type emp_tab_type IS TABLE OF emp_cursor % rowtype INDEX BY binary_integer;
13 emp_tab emp_tab_type;
14 i NUMBER := 1;
15 v_city VARCHAR2(30);
16 BEGIN
17
18   OPEN emp_cursor;
19   FETCH emp_cursor
20   INTO emp_record;
21   emp_tab(i) := emp_record;
22   WHILE(emp_cursor % FOUND)
23     AND(i <= pmaxrows)
24   LOOP
25     i := i + 1;
26     FETCH emp_cursor
27     INTO emp_record;
28     emp_tab(i) := emp_record;
29     v_city := get_location(emp_record.department_name);
30     DBMS_OUTPUT.PUT_LINE('Employee ' || emp_record.last_name || ' works in ' || v_city);
31   END LOOP;
32
33   CLOSE emp_cursor;
34   FOR j IN REVERSE 1 .. i
35   LOOP
36     DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
37   END LOOP;
38 END emp_list;
```

ORACLE®

2 - 34

Copyright © 2009, Oracle. All rights reserved.

Debugging a Procedure Example: Creating a New emp_list Procedure

The emp_list procedure gathers employee information such as the employee's department name, ID, name, salary, and commission percentage. The procedure creates a record to store the employee's information. The procedure also creates a table that can hold multiple records of employees. "i" is a counter.

The code opens the cursor and fetches the employees' records. The code also checks whether or not there are more records to fetch or whether the number of records fetched so far is less than the number of records that you specify. The code eventually prints out the employees' information. The procedure also calls the get_location function that returns the name of the city in which an employee works.

Note: Make sure that you are displaying the procedure code in edit mode. To edit the procedure code, click the Edit icon on the procedure's toolbar.

Debugging a Procedure Example: Creating a New get_location Function

```
1 CREATE OR REPLACE FUNCTION get_location(p_deptname IN VARCHAR2) RETURN VARCHAR2 AS
2   v_loc_id NUMBER;
3   v_city VARCHAR2(30);
4   BEGIN
5     SELECT d.location_id,
6           l.city
7     INTO v_loc_id,
8          v_city
9     FROM departments d,
10          locations l
11    WHERE UPPER(department_name) = UPPER(p_deptname)
12      AND d.location_id = l.location_id;
13    RETURN v_city;
14  END get_location;
```

ORACLE®

2 - 35

Copyright © 2009, Oracle. All rights reserved.

Debugging a Procedure Example: Creating a New Procedure

This function returns the city in which an employee works. It is called from the emp_list procedure.

Setting Breakpoints and Compiling emp_list for Debug Mode

The screenshot shows the Oracle SQL Developer interface with the PL/SQL Editor open. The editor window title is "EMP_LIST". The code for the "emp_list" procedure is displayed, with several lines highlighted by red boxes:

- Line 18: `OPEN cur_emp;`
- Line 20: `WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP`
- Line 22: `FETCH cur_emp INTO rec_emp;`
- Line 23: `emp_tab(i) := rec_emp;`
- Line 24: `v_city := get_location(rec_emp.department_name);`
- Line 25: `dbms_output.put_line('Employee ' || rec_emp.last_name ||`
- Line 26: `" works in " || v_city);`
- Line 28: `CLOSE cur_emp;`
- Line 29: `FOR j IN REVERSE 1..i LOOP`
- Line 30: `DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);`
- Line 31: `END LOOP;`
- Line 32: `END emp_list;`

In the bottom right corner of the editor, there is a "Compile for Debug" button, which is also highlighted with a red box.

The "Messages - Log" tab at the bottom of the editor shows the message "EMP_LIST Compiled".

ORACLE®

Setting Breakpoints and Compiling emp_list for Debug Mode

In the example in the slide, the `emp_list` procedure is displayed in edit mode, and four breakpoints are added to various locations in the code. To compile the procedure for debugging, right-click the code, and then select `Compile for Debug` from the shortcut menu. The `Messages – Log` tab displays the message that the procedure was compiled.

Compiling the get_location Function for Debug Mode

The screenshot shows the Oracle SQL Developer interface. In the top navigation bar, the tab 'GET_LOCATION' is selected. Below the tabs, there is a toolbar with several icons. A red box highlights the 'Compile for Debug' icon, which is located near the end of the toolbar. The main code editor window contains the PL/SQL code for the 'get_location' function:

```
1 create or replace
2 FUNCTION get_location
3 ( p_deptname IN VARCHAR2 ) RETURN VARCHAR2
4 AS
5   v_loc_id NUMBER;
6   v_city  VARCHAR2(30);
7 BEGIN
8   SELECT d.location_id, l.city INTO v_loc_id, v_city
9   FROM departments d, locations l
10  WHERE upper(department_name) = upper(p_deptname)
11  and d.location_id = l.location_id;
12  RETURN v_city;
13 END GET_LOCATION;
14
```

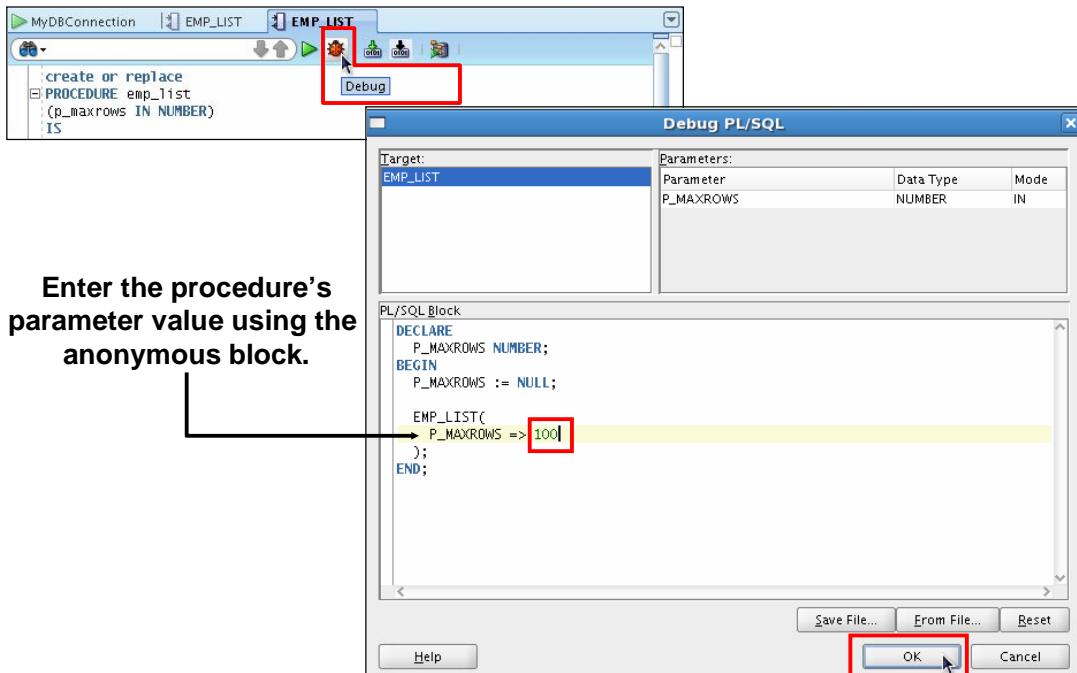
In the bottom right corner of the code editor, there is a small context menu with the option 'Compile for Debug' highlighted by a red box. At the bottom of the interface, there is a 'Messages - Log' tab. This tab displays the message 'EMP_LIST Compiled' and 'GET_LOCATION Compiled'. The 'Messages' tab is also visible.

ORACLE®

Compiling the get_location Function for Debug Mode

In the example in the slide, the `get_location` function is displayed in edit mode. To compile the function for debugging, right-click the code, and then select `Compile for Debug` from the shortcut menu. The `Messages – Log` tab displays the message that the function was compiled.

Debugging emp_list and Entering Values for the PMAXROWS Parameter



Debugging emp_list and Entering Values for the PMAXROWS Parameter

The next step in the debugging process is to debug the procedure using any of the several available methods mentioned earlier, such as clicking the Debug icon on the procedure's toolbar. An anonymous block is displayed, where you are prompted to enter the parameters for this procedure. `emp_list` has one parameter, `PMAXROWS`, which specifies the number of records to return. Replace the second `PMAXROWS` with a number such as 100, and then click OK.

Debugging emp_list: Step Into (F7) the Code

Program control stops at first breakpoint.

1

```
1 CREATE OR REPLACE PROCEDURE emp_list
2  (p_maxrows IN NUMBER)
3  IS
4  CURSOR cur_emp IS
5    SELECT d.department_name, e.employee_id, e.last_name,
6      e.salary, e.commission_pct
7    FROM departments d, employees e
8    WHERE d.department_id = e.department_id;
9    rec_emp cur_emp%ROWTYPE;
10   TYPE emp_tab_type IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
11   emp_tab emp_tab_type;
12   i NUMBER := 1;
13   v_city VARCHAR2(30);
14  BEGIN
15    OPEN cur_emp;
16    FETCH cur_emp INTO rec_emp;
17    emp_tab(i) := rec_emp;
18    WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
19      i := i + 1;
20      FETCH cur_emp INTO rec_emp;
21      emp_tab(i) := rec_emp;
22      v_city := get_location (rec_emp.department_name);
23      dbms_output.put_line('Employee ' || rec_emp.last_name ||
24        ' works in ' || v_city );
25    END LOOP;
26    CLOSE cur_emp;
27    FOR j IN REVERSE 1..i LOOP
28      DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
29    END LOOP;
30  END emp_list;
```

Debugging - Log Breakpoints Smart Data Data Watches
Debugger accepted connection from database on port 4000.
Processing 59 classes that have already been prepared...
Finished processing prepared classes.
Source breakpoint occurred at line 15 of EMP_LIST.pls.

ORACLE®

The Step Into Debugging Tool

The Step Into command executes a single program statement at a time. If the execution point is located on a call to a subprogram, the Step Into command steps into that subprogram and places the execution point on the subprogram's first statement. If the execution point is located on the last statement of a subprogram, choosing Step Into causes the debugger to return from the subprogram, placing the execution point on the line of code that follows the call to the subprogram you are returning from. The term *single stepping* refers to using Step Into to run successively through the statements in your program code. You can step into a subprogram in any of the following ways: Select Debug > Step Into, press F7, or click the Step Into icon in the Debugging – Log toolbar.

In the example in the slide, the program control is stopped at the first breakpoint in the code. The arrow next to the breakpoint indicated that this is the line of code that will be executed next. Note the various tabs at the bottom of the window.

Note: The Step Into and Step Over commands offer the simplest way of moving through your program code. While the two commands are very similar, they each offer a different way to control code execution.

Debugging emp_list: Step Into (F7) the Code

The screenshot shows the Oracle SQL Developer interface with a PL/SQL editor window titled 'EMP_LIST'. The code is as follows:

```

1  WHEN d.department_id = e.department_id;
2  rec_emp%ROWTYPE;
3  TYPE emp_rec IS TABLE OF cur_emp%ROWTYPE INDEX BY BINARY_INTEGER;
4  emp_tab : emp_rec;
5  NUMBER;
6  v_city VARCHAR2(30);
7
8  BEGIN
9    OPEN cur_emp;
10   FETCH cur_emp INTO rec_emp;
11   emp_tab(1) := rec_emp;
12   WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
13     i := i + 1;
14     FETCH cur_emp INTO rec_emp;
15     emp_tab(i) := rec_emp;
16     v_city := get_location (rec_emp.department_id);
17     dbms_output.put_line('Employee ' || rec_emp.last_name || ' works in ' || v_city );
18   END LOOP;
19   CLOSE cur_emp;
20   FOR j IN REVERSE 1..i LOOP
21     DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
22   END LOOP;
23 END emp_list;
24
25
26
27
28
29
30
31

```

A red box highlights the line 'OPEN cur_emp;'. A green circle labeled '1' is over this line. A red box highlights the line 'FETCH cur_emp INTO rec_emp;'. A green circle labeled '2' is over this line. A red box highlights the line 'SELECT d.department_name, e.employee_id, e.salary, e.commission_pct'. A green circle labeled '3' is over this line.

Step Into (F7):
Steps into and executes the cursor code. Control is transferred to cursor definition.

Debugging emp_list: Step Into the Code

Selecting the Step Into command executes a single program statement at a time. If the execution point is located on a call to a subprogram, the Step Into command steps into that subprogram and places the execution at the first statement in the subprogram.

In the example in the slide, pressing F7 executes the line of code at the first breakpoint. In this case, program control is transferred to the section where the cursor is defined.

Viewing the Data

```
18 OPEN emp_cursor;
20 FETCH emp_cursor
```

```
16 OPEN cur_emp;
17 WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
```

Name	Value	Type
P_MAXROWS	100	NUMBER
REC_EMP	Rowtype	
EMP_TAB	IndexedTable	EMP_TAB_TYPE
_values	1	EMP_TAB_TYPE elem...
V_CITY	NULL	VARCHAR2(30)

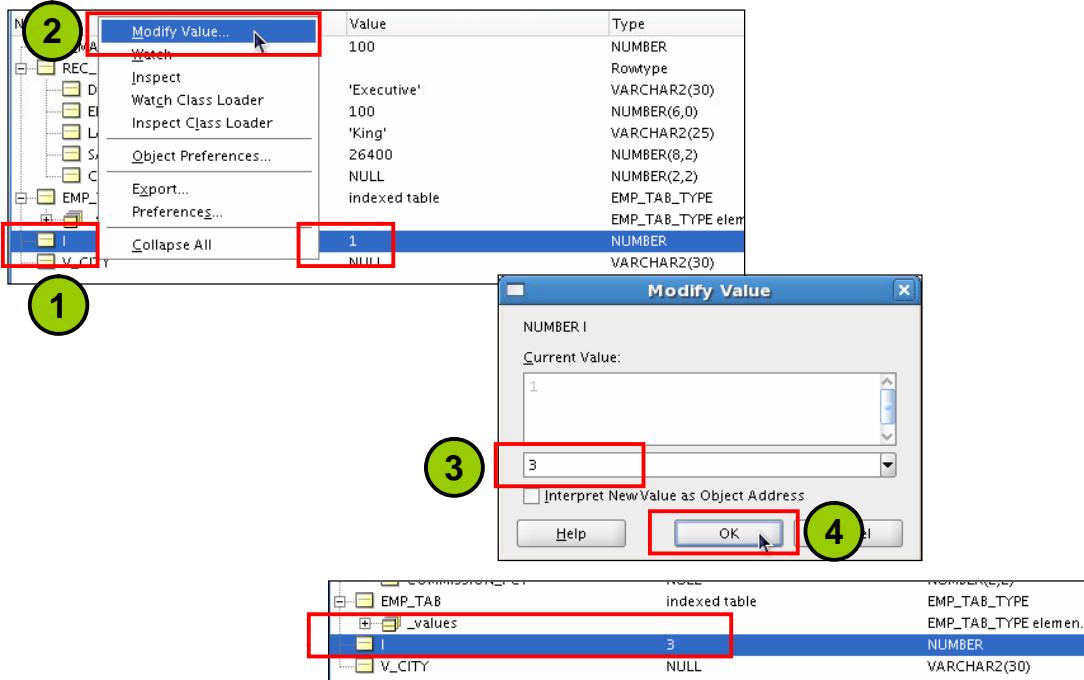
Name	Value	Type
P_MAXROWS	100	NUMBER
REC_EMP	Rowtype	
EMP_TAB	IndexedTable	EMP_TAB_TYPE
_values	1	EMP_TAB_TYPE elem...
V_CITY	NULL	VARCHAR2(30)

ORACLE®

Viewing the Data

While you are debugging your code, you can use the Data tab to display and modify the variables. You can also set watches to monitor a subset of the variables displayed in the Data tab. To display or hide the Data, Smart Data, and Watch tabs: Select View > Debugger, and then select the tabs that you want to display or hide.

Modifying the Variables While Debugging the Code



Modifying the Variables While Debugging the Code

To modify the value of a variable in the Data tab, right-click the variable name, and then select Modify Value from the shortcut menu. The Modify Value window is displayed. The current value for the variable is displayed. You can enter a new value in the second text box, and then click OK.

Debugging emp_list: Step Over the Code

```
14 BEGIN
15   OPEN cur_emp;
16   FETCH cur_emp INTO rec_emp;
17   emp_tab(i) := rec_emp;
18   WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
19     i := i + 1;
20     FETCH cur_emp INTO rec_emp;
21     emp_tab(i) := rec_emp;
22     v_city := get_location (rec_emp.department_name);
23     dbms_output.put_line('Employee ' || rec_emp.last_name ||
24                           ' works in ' || v_city );
```

1 F8

Step Over (F8):
Executes the Cursor
(same as F7),
but control is not transferred
to Open Cursor code

```
14 BEGIN
15   OPEN cur_emp;
16   FETCH cur_emp INTO rec_emp;
17   emp_tab(i) := rec_emp;
18   WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
19     i := i + 1;
20     FETCH cur_emp INTO rec_emp;
21     emp_tab(i) := rec_emp;
22     v_city := get_location (rec_emp.department_name);
23     dbms_output.put_line('Employee ' || rec_emp.last_name ||
24                           ' works in ' || v_city );
```

2 F8

```
14 BEGIN
15   OPEN cur_emp;
16   FETCH cur_emp INTO rec_emp;
17   emp_tab(i) := rec_emp;
18   WHILE (cur_emp%FOUND) AND (i <= p_maxrows) LOOP
19     i := i + 1;
20     FETCH cur_emp INTO rec_emp;
21     emp_tab(i) := rec_emp;
22     v_city := get_location (rec_emp.department_name);
23     dbms_output.put_line('Employee ' || rec_emp.last_name ||
24                           ' works in ' || v_city );
```

3 F8

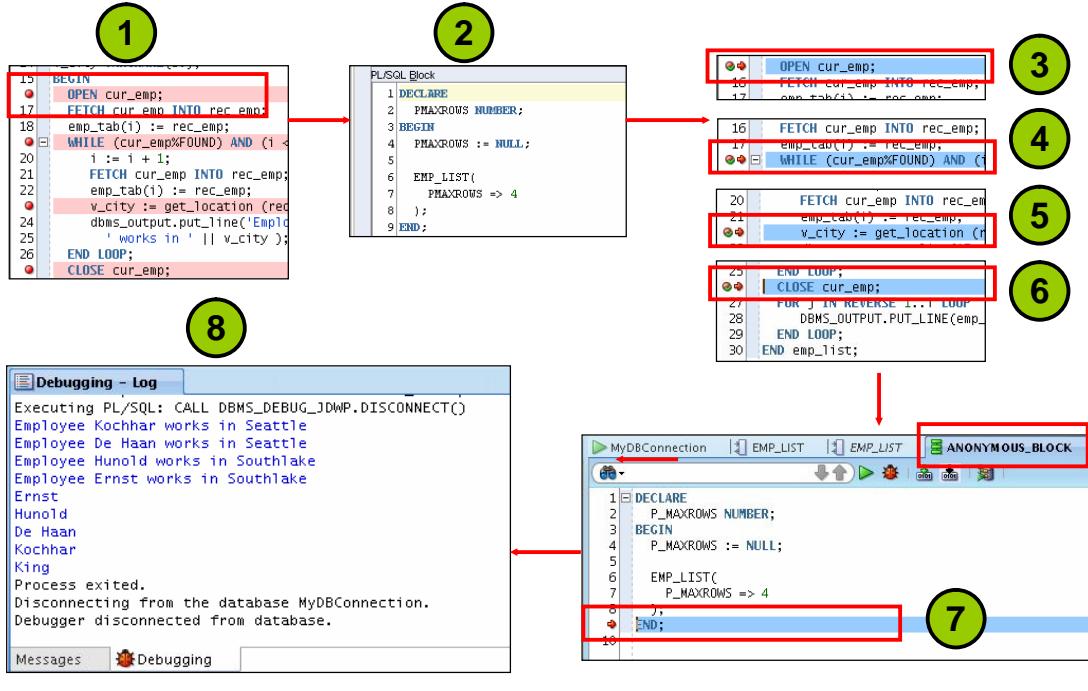
ORACLE®

The Step Over Debugging Tool

The Step Over command, like Step Into, enables you to execute program statements one at a time. However, if you issue the Step Over command when the execution point is located on a subprogram call, the debugger runs that subprogram without stopping (instead of stepping into it), and then positions the execution point on the statement that follows the subprogram call. If the execution point is located on the last statement of a subprogram, choosing Step Over causes the debugger to return from the subprogram, placing the execution point on the line of code that follows the call to the subprogram you are returning from. You can step over a subprogram in any of the following ways: Select Debug > Step Over, press F8, or click the Step Over icon in the Debugging – Log toolbar.

In the example in the slide, stepping over will execute the open cursor line without transferring program control to the cursor definition as was the case with the Step Into option example.

Debugging emp_list: Step Out of the Code (Shift + F7)



Debugging emp_list: Step Out of the Code (Shift + F7)

The Step Out of the code option leaves the current subprogram and goes to the next statement subprogram. In the example in the slide, starting with step 3, if you press Shift + F7, program control leaves the procedure and goes to the next statement in the anonymous block. Continuing to press Shift + F7 will take you to the next anonymous block that prints the contents of the SQL buffer.

Debugging emp_list: Run to Cursor (F4)

```
10  emp_record emp_cursor%ROWTYPE;
11  TYPE emp_tab_type IS TABLE OF emp_cursor%ROWTYPE INDEX BY BINARY_INTEGER;
12  emp_tab emp_tab_type;
13  i NUMBER := 1;
14  v_city VARCHAR2(30);
15  BEGIN
16      OPEN emp_cursor;
17      FETCH emp_cursor INTO emp_record;
18      emp_tab(i) := emp_record;
19      WHILE (emp_cursor%FOUND) AND (i <= pMaxRows) LOOP
20          i := i + 1;
21          FETCH emp_cursor INTO emp_record;
22          emp_tab(i) := emp_record;
23          v_city := get_location (emp_record.department_name);
24          dbms_output.put_line('Employee ' || emp_record.last_name ||
25                                ' works in ' || v_city );
26      END LOOP;
27      CLOSE emp_cursor;
28      FOR j IN REVERSE 1..1 LOOP
29          DBMS_OUTPUT.PUT_LINE(emp_tab(j).last_name);
30      END LOOP;
31  END emp_list;
32 
```

Run to Cursor F4:
Run to your cursor location
without having to single
step or set a breakpoint.

Name	Value	Type
PMAXROWS	5	NUMBER
EMP_RECORD		Rowtype
DEPARTMENT_NAME	'Administration'	VARCHAR2..
EMPLOYEE_ID	200	NUMBER(6,0)
LAST_NAME	'Whalen'	VARCHAR2..
SALARY	4400	NUMBER(8,2)
COMMISSION_PCT	NULL	NUMBER(2,2)
EMP_TAB		indexed table
I	1	NUMBER
V_CITY	NULL	VARCHAR2..

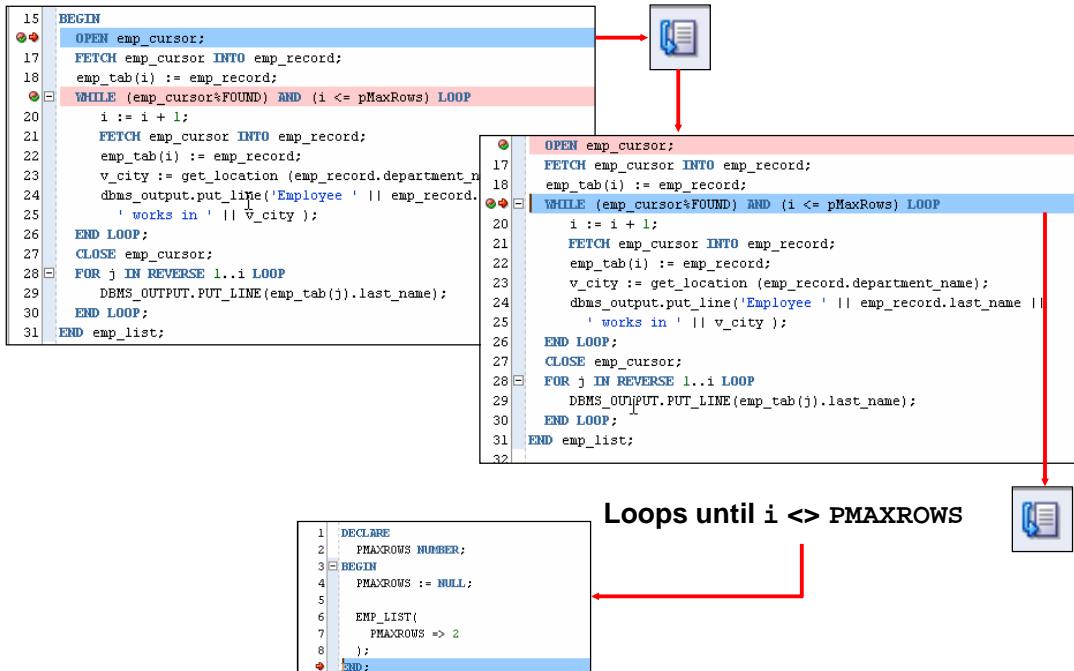
Running to the Cursor Location

When stepping through your application code in the debugger, you may want to run to a particular location without having to single step or set a breakpoint. To run to a specific program location: In a subprogram editor, position your text cursor on the line of code where you want the debugger to stop. You can run to the cursor location using any of the following procedures: In the procedure editor, right-click and choose Run to Cursor, choose the Debug > Run to Cursor option from the main menu, or press F4.

Any of the following conditions may result:

- When you run to the cursor, your program executes without stopping, until the execution reaches the location marked by the text cursor in the source editor.
- If your program never actually executes the line of code where the text cursor is, the Run to Cursor command will cause your program to run until it encounters a breakpoint or until it finishes.

Debugging emp_list: Step to End of Method



ORACLE®

2 - 46

Copyright © 2009, Oracle. All rights reserved.

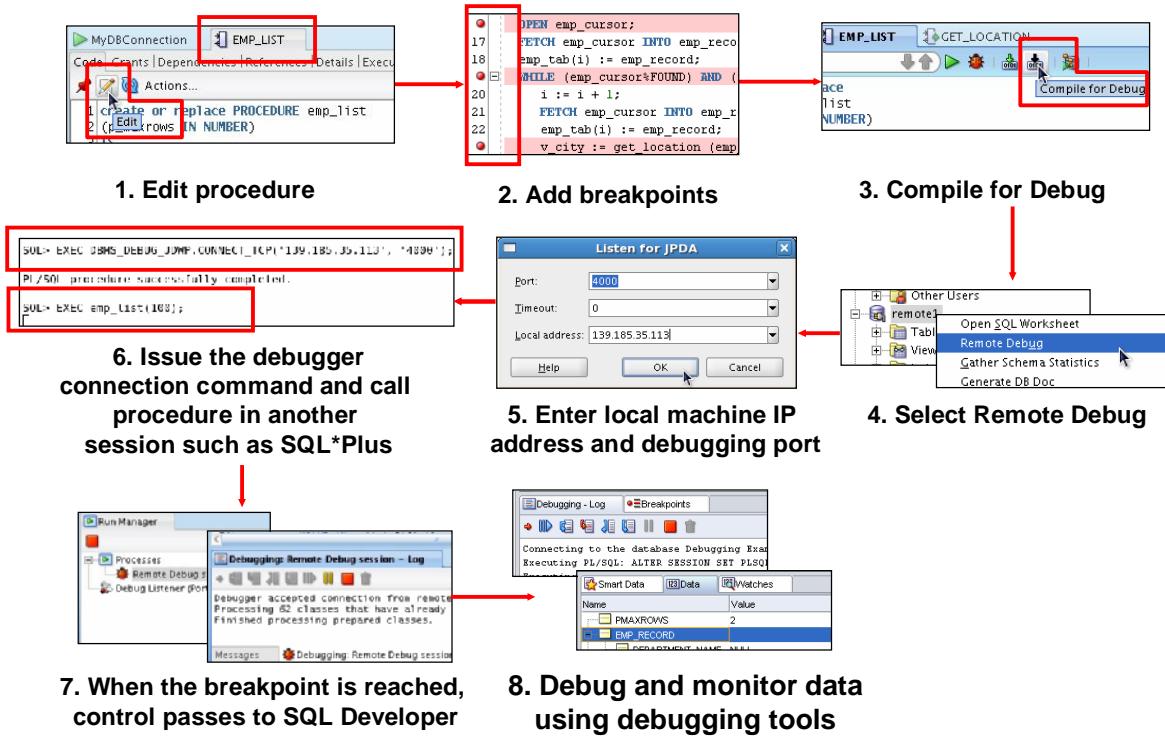
Debugging emp_list: Step to End of Method

Step to End of Method goes to the last statement in the current subprogram or to the next breakpoint if there are any in the current subprogram.

In the example in the slide, the Step to End of Method debugging tool is used. Because there is a second breakpoint, selecting Step to End of Method transfers control to that breakpoint.

Selecting Step to End of Method again goes through the iterations of the while loop first, and then transfers the program control to the next executable statement in the anonymous block.

Debugging a Subprogram Remotely: Overview



Remote Debugging

You can use remote debugging to connect to any PL/SQL code in a database and debug it, regardless of where the database is located as long as you have access to the database and have created a connection to it. Remote debugging is when something else, other than you as a developer, kicks off a procedure that you can then debug. Remote debugging and local debugging have many steps in common. To debug remotely, perform the following steps:

1. Edit the subprogram that you would like to debug.
2. Add the breakpoints in the subprogram.
3. Click the **Compile for Debug** icon in the toolbar.
4. Right-click the connection for the remote database, and select **Remote Debug**.
5. In the **Debugger - Attach to JPDA** dialog box, enter the local machine IP address and port number, such as 4000, and then click **OK**.
6. Issue the debugger connection command using a different session such as SQL*Plus, and then call the procedure in that session.
7. When a breakpoint is reached, control is passed back to the original SQL Developer session and the debugger toolbar is displayed.
8. Debug the subprogram and monitor the data using the debugging tools discussed earlier.

Practice 2-2 Overview: Introduction to the SQL Developer Debugger

This practice covers the following topics:

- Creating a procedure and a function
- Inserting breakpoints in the procedure
- Compiling the procedure and function for debug mode
- Debugging the procedure and stepping into the code
- Displaying and modifying the subprograms' variables



Practice 2-2: Overview

You must use SQL Developer for this practice. This practice introduces you to the basic functionality of the SQL Developer debugger.

Summary

In this lesson, you should have learned how to:

- Differentiate between a procedure and a function
- Describe the uses of functions
- Create stored functions
- Invoke a function
- Remove a function
- Understand the basic functionality of the SQL Developer debugger



Summary

A function is a named PL/SQL block that must return a value. Generally, you create a function to compute and return a value, and you create a procedure to perform an action.

A function can be created or dropped.

A function is invoked as a part of an expression.

3

Creating Packages

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

Objectives

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

- Describe packages and list their components
- Create a package to group together related variables, cursors, constants, exceptions, procedures, and functions
- Designate a package construct as either public or private
- Invoke a package construct
- Describe the use of a bodiless package

ORACLE®

3 - 2

Copyright © 2009, Oracle. All rights reserved.

Lesson Aim

In this lesson, you learn what a package is and what its components are. You also learn how to create and use packages.

Lesson Agenda

- Identifying the benefits and the components of packages
- Working with packages:
 - Creating the package specification and body
 - Invoking the package subprograms
 - Removing a package
 - Displaying the package information

ORACLE®

3 - 3

Copyright © 2009, Oracle. All rights reserved.

What Are PL/SQL Packages?

- A package is a schema object that groups logically related PL/SQL types, variables, and subprograms.
- Packages usually have two parts:
 - A specification (spec)
 - A body
- The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package.
- The body defines the queries for the cursors and the code for the subprograms.
- Enable the Oracle server to read multiple objects into memory at once.

ORACLE®

3 - 4

Copyright © 2009, Oracle. All rights reserved.

PL/SQL Packages: Overview

PL/SQL packages enable you to bundle related PL/SQL types, variables, data structures, exceptions, and subprograms into one container. For example, a Human Resources package can contain hiring and firing procedures, commission and bonus functions, and tax exemption variables.

A package usually consists of two parts 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. Subsequent access to constructs in the same package does not require disk input/output (I/O).

Advantages of Using Packages

- Modularity: Encapsulating related constructs
- Easier maintenance: Keeping logically related functionality together
- Easier application design: Coding and compiling the specification and body separately
- Hiding information:
 - Only the declarations in the package specification are visible and accessible to applications
 - Private constructs in the package body are hidden and inaccessible
 - All coding is hidden in the package body

ORACLE®

3 - 5

Copyright © 2009, Oracle. All rights reserved.

Advantages of Using Packages

Packages provide an alternative to creating procedures and functions as stand-alone schema objects, and they offer several benefits.

Modularity and easier maintenance: You encapsulate logically related programming structures in a named module. Each package is easy to understand, and the interface between packages is simple, clear, and well defined.

Easier application design: All you need initially is the interface information in the package specification. You can code and compile a specification without its body. Thereafter, stored subprograms that reference the package can compile as well. You need not define the package body fully until you are ready to complete the application.

Hiding information: You decide which constructs are public (visible and accessible) and which are private (hidden and inaccessible). Declarations in the package specification are visible and accessible to applications. The package body hides the definition of the private constructs, so that only the package is affected (not your application or any calling programs) if the definition changes. This enables you to change the implementation without having to recompile the calling programs. Also, by hiding implementation details from users, you protect the integrity of the package.

Advantages of Using Packages

- Added functionality: Persistence of public variables and cursors
- Better performance:
 - The entire package is loaded into memory when the package is first referenced.
 - There is only one copy in memory for all users.
 - The dependency hierarchy is simplified.
- Overloading: Multiple subprograms of the same name



Advantages of Using Packages (continued)

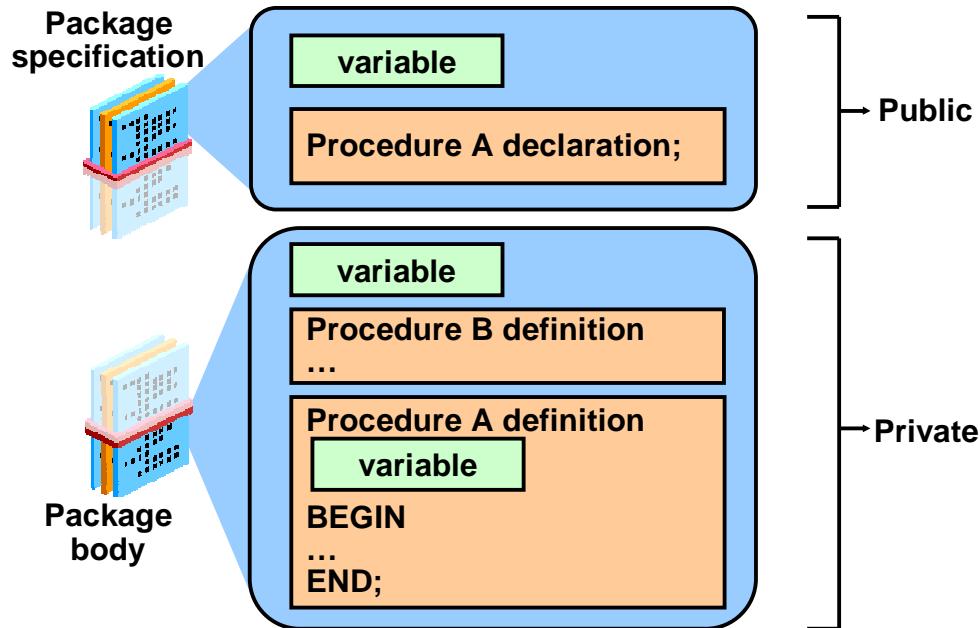
Added functionality: Packaged public variables and cursors persist for the duration of a session. Thus, they can be shared by all subprograms that execute in the environment. They also enable you to maintain data across transactions without having to store it in the database. Private constructs also persist for the duration of the session but can be accessed only within the package.

Better performance: When you call a packaged subprogram the first time, the entire package is loaded into memory. Later calls to related subprograms in the package, therefore, require no further disk I/O. Packaged subprograms also stop cascading dependencies and thus avoid unnecessary compilation.

Overloading: With packages, you can overload procedures and functions, which means you can create multiple subprograms with the same name in the same package, each taking parameters of different number or data type.

Note: Dependencies are covered in detail in the lesson titled “Managing Dependencies.”

Components of a PL/SQL Package



Components of a PL/SQL Package

You create a package in two parts:

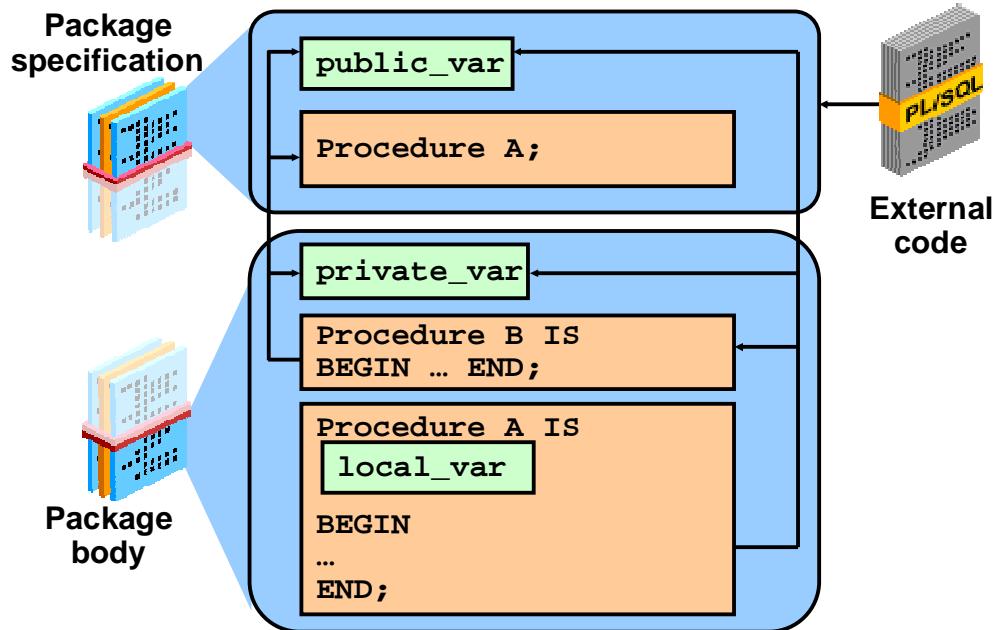
- The **package specification** is the interface to your applications. It declares the public types, variables, constants, exceptions, cursors, and subprograms available for use. The package specification may also include PRAGMAS, which are directives to the compiler.
- The **package body** defines its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

Public components are declared in the package specification. The specification defines a public application programming interface (API) for users of 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 and can be referenced only by other constructs within the same package body. Private components can reference the public components of a package.

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

Internal and External Visibility of a Package's Components



Internal and External Visibility of a Package's Components

The visibility of a component describes whether that component can be seen, that is, referenced and used by other components or objects. The visibility of components depends on whether they are *locally* or *globally* declared.

Local components are visible within the structure in which they are declared, such as:

- Variables defined in a subprogram can be referenced within that subprogram, and are not visible to external components—for example, `local_var` can be used in procedure A.
- Private package variables, which are declared in a package body, can be referenced by other components in the same package body. They are not visible to any subprograms or objects that are outside the package. For example, `private_var` can be used by procedures A and B within the package body, but not outside the package.

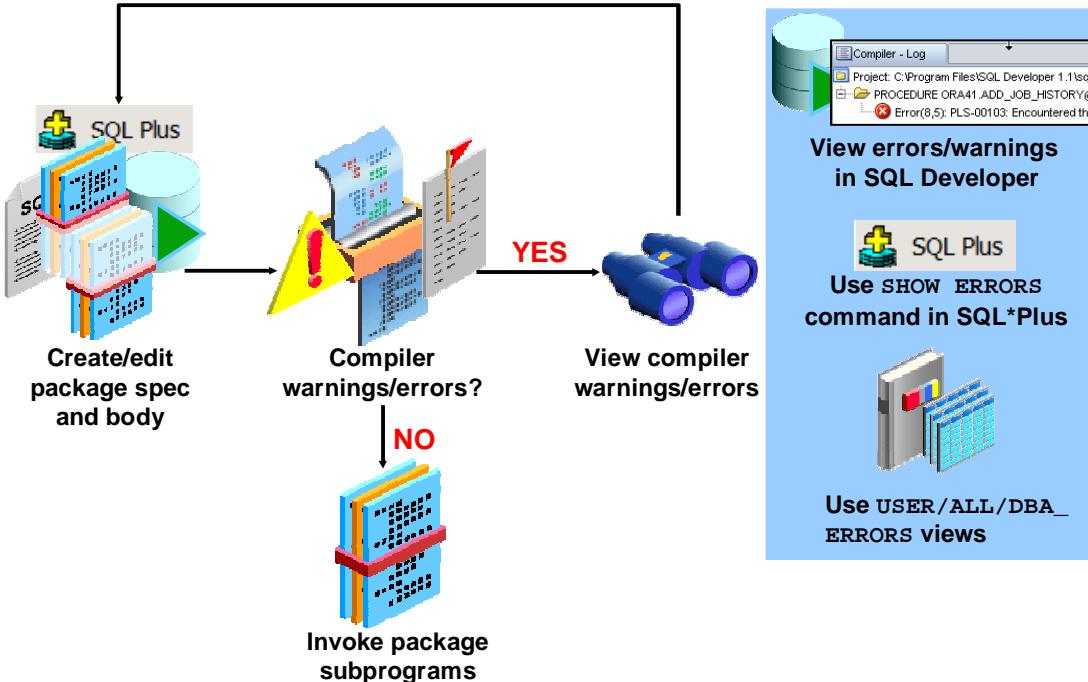
Globally declared components are visible internally and externally to the package, such as:

A public variable, which is declared in a package specification, can be referenced and changed outside the package (for example, `public_var` can be referenced externally).

- A package subprogram in the specification can be called from external code sources (for example, procedure A can be called from an environment external to the package).

Note: Private subprograms, such as procedure B, can be invoked only with public subprograms, such as procedure A, or other private package constructs. A public variable declared in the package specification is a global variable.

Developing PL/SQL Packages: Overview



Developing PL/SQL Packages

The graphic in the slide illustrates the basic steps involved in developing and using a package:

1. Create the procedure using SQL Developer's Object Navigator tree or the SQL Worksheet area.
2. Compile the package. The package is created in the database. The CREATE PACKAGE statement creates and stores source code and the compiled *m-code* in the database. To compile the package, right-click the package's name in the Object Navigator tree, and then click Compile.
3. If there are no compilation warnings or errors, you execute any public construct within the package specification from an Oracle Server environment.
4. If there are compilation warning or errors, you can view (and then correct) the warnings or errors using one of the following methods:
 - Using the SQL Developer interface (the Compiler – Log tab)
 - Using the SHOW ERRORS SQL*Plus command
 - Using the USER/ALL/DBA_ERRORS views

Lesson Agenda

- Identifying the benefits and the components of packages
- Working with packages:
 - Creating the package specification and body
 - Invoking the package subprograms
 - Removing a package
 - Displaying the package information

ORACLE®

3 - 10

Copyright © 2009, Oracle. All rights reserved.

Creating the Package Specification: Using the CREATE PACKAGE Statement

```
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 the constructs declared in a package specification are visible to users who are granted privileges on the package.

ORACLE®

3 - 11

Copyright © 2009, 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.

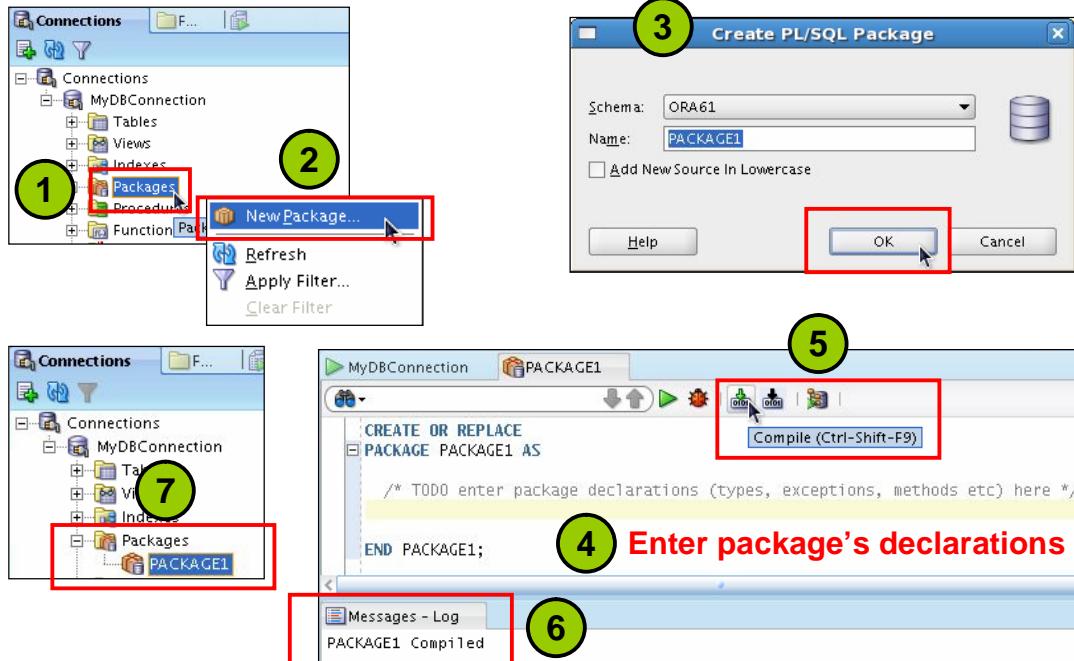
The following are definitions of items in the package syntax:

- **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 specification** specifies the public procedure or function declarations.

The package specification should contain procedure and function headings terminated by a semicolon, without the IS (or AS) keyword and its PL/SQL block. The implementation of a procedure or function that is declared in a package specification is done in the package body.

The Oracle database stores the specification and body of a package separately. This enables you to change the implementation of a program construct in the package body without invalidating other schema objects that call or reference the program construct.

Creating the Package Specification: Using SQL Developer

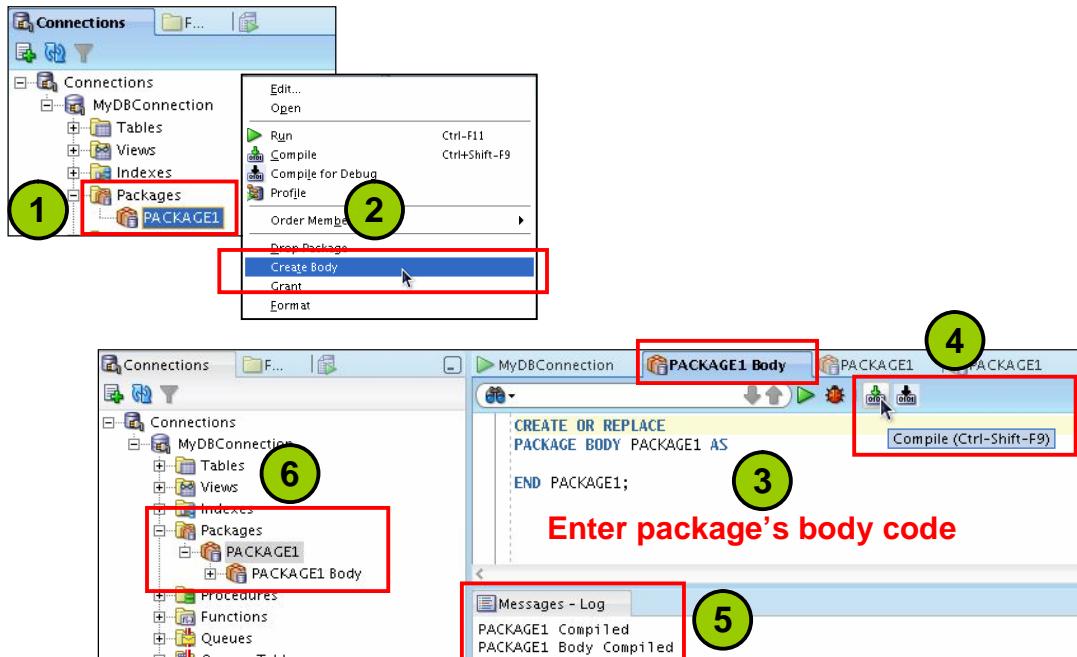


Creating the Package Specification: Using SQL Developer

You can use SQL Developer to create the package specification as follows:

1. Right-click the **Packages** node in the Connections navigation tree.
2. Select **New Package** from the shortcut menu.
3. In the **Create PL/SQL Package** window, select the schema name, enter the name for the new package, and then click OK. A tab for the new package is displayed along with the shell for the new package.
4. Enter the code for the new package.
5. Compile or save (using the Save icon on the main toolbar) the new package.
6. The **Messages – Log** tab displays whether or not the compilation was successful.
7. The newly created package is displayed under the **Packages** node in the Connections navigation tree.

Creating the Package Body: Using SQL Developer



Creating the Package Body: Using SQL Developer

You can use SQL Developer to create the package body as follows:

1. Right-click the package name for which you are creating a body in the **Packages** node in the Connections navigation tree.
2. Select **Create Body** from the shortcut menu. A tab for the new package body is displayed along with the shell for the new package body.
3. Enter the code for the new package body.
4. Compile or save the package body.
5. The **Messages – Log** tab displays whether or not the compilation was successful.
6. The newly created package body is displayed under the **Packages** node in the Connections navigation tree.

Example of a Package Specification: comm_pkg

```
-- The package spec with a public variable and a
-- public procedure that are accessible from
-- outside the package.

CREATE OR REPLACE PACKAGE comm_pkg IS
    v_std_comm NUMBER := 0.10;  --initialized to 0.10
    PROCEDURE reset_comm(p_new_comm NUMBER);
END comm_pkg;
/
```

- V_STD_COMM is a *public* global variable initialized to 0.10.
- RESET_COMM is a *public* procedure used to reset the standard commission based on some business rules. It is implemented in the package body.

ORACLE®

3 - 14

Copyright © 2009, Oracle. All rights reserved.

Example of Package Specification: comm_pkg

The example in the slide creates a package called comm_pkg used to manage business processing rules for commission calculations.

The v_std_comm public (global) variable is declared to hold a maximum allowable percentage commission for the user session, and it is initialized to 0.10 (that is, 10%).

The reset_comm public procedure is declared to accept a new commission percentage that updates the standard commission percentage if the commission validation rules are accepted. The validation rules for resetting the commission are not made public and do not appear in the package specification. The validation rules are managed by using a private function in the package body.

Creating the Package Body

```
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 supporting private constructs. When creating a package body, perform the following steps:

- Specify the OR REPLACE option to overwrite an existing 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.
- Complete the implementation for all procedures or functions declared in the package specification within the package body.

The following are 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 specification** 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.

Example of a Package Body: comm_pkg

```
CREATE OR REPLACE PACKAGE BODY comm_pkg IS
    FUNCTION validate(p_comm NUMBER) RETURN BOOLEAN IS
        v_max_comm    employees.commission_pct%type;
    BEGIN
        SELECT MAX(commission_pct) INTO v_max_comm
        FROM   employees;
        RETURN (p_comm BETWEEN 0.0 AND v_max_comm);
    END validate;

    PROCEDURE reset_comm (p_new_comm NUMBER) IS
    BEGIN
        IF validate(p_new_comm) THEN
            v_std_comm := p_new_comm; -- reset public var
        ELSE
            RAISE_APPLICATION_ERROR(
                -20210, 'Bad Commission');
        END IF;
    END reset_comm;
END comm_pkg;
```

ORACLE®

3 - 16

Copyright © 2009, Oracle. All rights reserved.

Example of a Package Body: comm_pkg

The slide shows the complete package body for comm_pkg, with a private function called validate to check for a valid commission. The validation requires that the commission be positive and less than the highest commission among existing employees. The reset_comm procedure invokes the private validation function before changing the standard commission in v_std_comm. In the example, note the following:

- The v_std_comm variable referenced in the reset_comm procedure is a public variable. Variables declared in the package specification, such as v_std_comm, can be directly referenced without qualification.
- The reset_comm procedure implements the public definition in the specification.
- In the comm_pkg body, the validate function is private and is directly referenced from the reset_comm procedure without qualification.

Note: The validate function appears before the reset_comm procedure because the reset_comm procedure references the validate function. It is possible to create forward declarations for subprograms in the package body if their order of appearance needs to be changed. If a package specification declares only types, constants, variables, and exceptions without any subprogram specifications, then the package body is unnecessary. However, the body can be used to initialize items declared in the package specification.

Invoking the Package Subprograms: Examples

```
-- Invoke a function within the same package:  
CREATE OR REPLACE PACKAGE BODY comm_pkg IS ...  
  PROCEDURE reset_comm(p_new_comm NUMBER) IS  
    BEGIN  
      IF validate(p_new_comm) THEN  
        v_std_comm := p_new_comm;  
      ELSE ...  
      END IF;  
    END reset_comm;  
END comm_pkg;
```

```
-- Invoke a package procedure from SQL*Plus:  
EXECUTE comm_pkg.reset_comm(0.15)
```

```
-- Invoke a package procedure in a different schema:  
EXECUTE scott.comm_pkg.reset_comm(0.15)
```

ORACLE®

3 - 17

Copyright © 2009, Oracle. All rights reserved.

Invoking Package Subprograms

After the package is stored in the database, you can invoke public or private subprograms within the same package, or public subprograms if external to the package. Fully qualify the subprogram with its package name when invoked externally from the package. Use the package_name . subprogram syntax.

Fully qualifying a subprogram when invoked within the same package is optional.

Example 1: Invokes the validate function from the reset_comm procedure within the same package. The package name prefix is not required; it is optional.

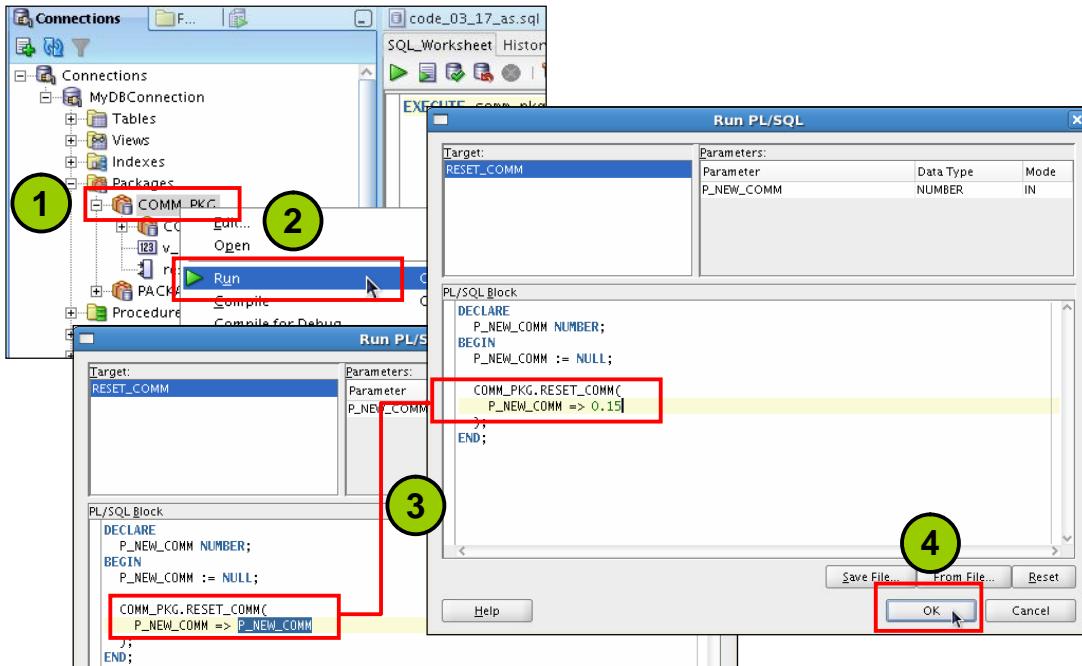
Example 2: Calls the reset_comm procedure from SQL*Plus (an environment external to the package) to reset the prevailing commission to 0.15 for the user session.

Example 3: Calls the reset_comm procedure that is owned by a schema user called SCOTT. Using SQL*Plus, the qualified package procedure is prefixed with the schema name. This can be simplified by using a synonym that references the schema . package_name.

Assume that a database link named NY has been created for a remote database in which the reset_comm package procedure is created. To invoke the remote procedure, use:

```
EXECUTE comm_pkg.reset_comm@NY(0.15)
```

Invoking the Package Subprograms: Using SQL Developer



ORACLE®

3 - 18

Copyright © 2009, Oracle. All rights reserved.

Invoking the Package Subprograms: Using SQL Developer

You can use SQL Developer to invoke a package's subprogram as follows:

1. Right-click the package's name in the Packages node in the Navigation tree.
2. Select **Run** from the floating menu. The **Run PL/SQL** window is displayed. You can use the **Run PL/SQL** window to specify parameter values for running a PL/SQL function or procedure. (If you specify a package, select a function or procedure in the package.) Specify the following:
 - a. **Target:** Select the name of the function or procedure to run.
 - b. **Parameters:** This section lists each parameter for the specified target. The mode of each parameter can be **IN** (the value is passed in), **OUT** (the value is returned), or **IN/OUT** (the value is passed in, and the result of the function or procedure's action is stored in the parameter).
3. In the **PL/SQL Block** section, change the formal **IN** and **IN/OUT** parameter specifications in this block to actual values that you want to use for running the function or procedure. For example, to specify **0 . 15** as the value for an input parameter named **P_NEW_COMM**, change **P_NEW_COMM => P_NEW_COMM** to **P_NEW_COMM => 0 . 15**.
4. Click **OK**. SQL Developer runs the function or procedure.

Creating and Using Bodiless Packages

```
CREATE OR REPLACE PACKAGE global_consts IS
    c_mile_2_kilo CONSTANT NUMBER := 1.6093;
    c_kilo_2_mile CONSTANT NUMBER := 0.6214;
    c_yard_2_meter CONSTANT NUMBER := 0.9144;
    c_meter_2_yard CONSTANT NUMBER := 1.0936;
END global_consts;
```

```
SET SERVEROUTPUT ON
BEGIN
    DBMS_OUTPUT.PUT_LINE('20 miles = ' ||
        20 * global_consts.c_mile_2_kilo || ' km');
END;
```

```
SET SERVEROUTPUT ON
CREATE FUNCTION mtr2yrd(p_m NUMBER) RETURN NUMBER IS
BEGIN
    RETURN (p_m * global_consts.c_meter_2_yard);
END mtr2yrd;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(mtr2yrd(1))
```

ORACLE®

3 - 19

Copyright © 2009, Oracle. All rights reserved.

Creating and Using Bodiless Packages

The variables and constants declared within stand-alone subprograms exist only for the duration that the subprogram executes. To provide data that exists for the duration of the user session, create a package specification containing public (global) variables and constant declarations. In this case, create a package specification without a package body, known as a *bodiless package*. As discussed earlier in this lesson, if a specification declares only types, constants, variables, and exceptions, then the package body is unnecessary.

Examples

The first code box in the slide creates a bodiless package specification with several constants to be used for conversion rates. A package body is not required to support this package specification. It is assumed that the `SET SERVEROUTPUT ON` statement was issued before executing the code examples in the slide.

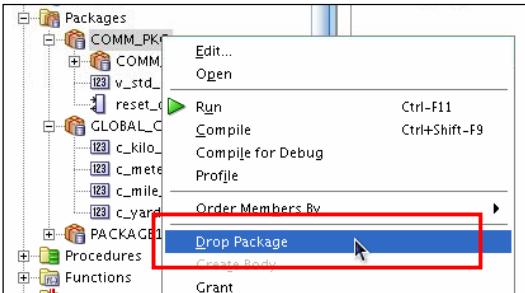
The second code box references the `c_mile_2_kilo` constant in the `global_consts` package by prefixing the package name to the identifier of the constant.

The third example creates a stand-alone function `c_mtr2yrd` to convert meters to yards, and uses the constant conversion rate `c_meter_2_yard` declared in the `global_consts` package. The function is invoked in a `DBMS_OUTPUT.PUT_LINE` parameter.

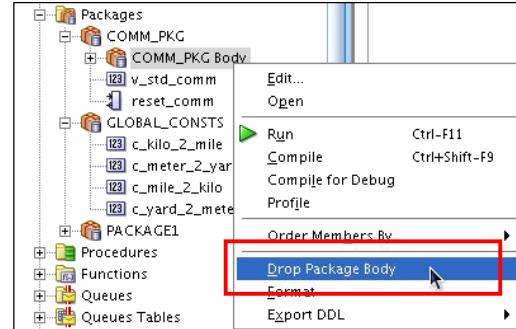
Rule to be followed: When referencing a variable, cursor, constant, or exception from outside the package, you must qualify it with the name of the package.

Removing Packages: Using SQL Developer or the SQL DROP Statement

Drop package specification and body



Drop package body only



```
-- Remove the package specification and body  
DROP PACKAGE package_name;
```

```
-- Remove the package body only  
DROP PACKAGE BODY package_name;
```

ORACLE®

Removing Packages

When a package is no longer required, you can use a SQL statement in SQL Developer to remove it. A package has two parts; therefore, you can remove the whole package, or you can remove only the package body and retain the package specification.

Viewing Packages Using the Data Dictionary

```
-- View the package specification.  
SELECT text  
FROM user_source  
WHERE name = 'COMM_PKG' AND type = 'PACKAGE'  
ORDER BY LINE;
```

TEXT
1 PACKAGE comm_pkg IS
2 std_comm NUMBER := 0.10; --initialized to 0.10
3 PROCEDURE reset_comm(new_comm NUMBER);
4 END comm_pkg;

```
-- View the package body.  
SELECT text  
FROM user_source  
WHERE name = 'COMM_PKG' AND type = 'PACKAGE BODY'  
ORDER BY LINE;
```

TEXT
1 PACKAGE BODY comm_pkg IS
2 FUNCTION validate(comm NUMBER) RETURN BOOLEAN IS
3 max_comm employees.commission_pct%type;
4 BEGIN
5 SELECT MAX(commission_pct) INTO max_comm
6 FROM employees;
7 RETURN (comm BETWEEN 0.0 AND max_comm);

ORACLE®

3 - 21

Copyright © 2009, Oracle. All rights reserved.

Viewing Packages in the Data Dictionary

The source code for PL/SQL packages is also stored in the USER_SOURCE and ALL_SOURCE data dictionary views. The USER_SOURCE table is used to display PL/SQL code that you own. The ALL_SOURCE table is used to display PL/SQL code to which you have been granted the EXECUTE right by the owner of that subprogram code and provides an OWNER column in addition to the preceding columns.

When querying the package, use a condition in which the TYPE column is:

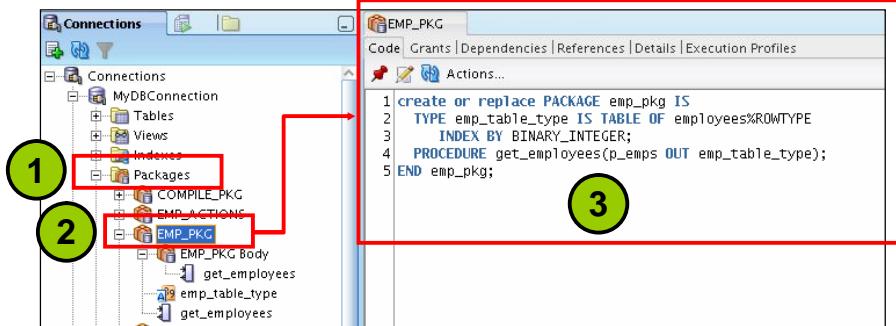
- Equal to 'PACKAGE' to display the source code for the package specification
- Equal to 'PACKAGE BODY' to display the source code for the package body

You can also view the package specification and body in SQL Developer using the package name in the Packages node.

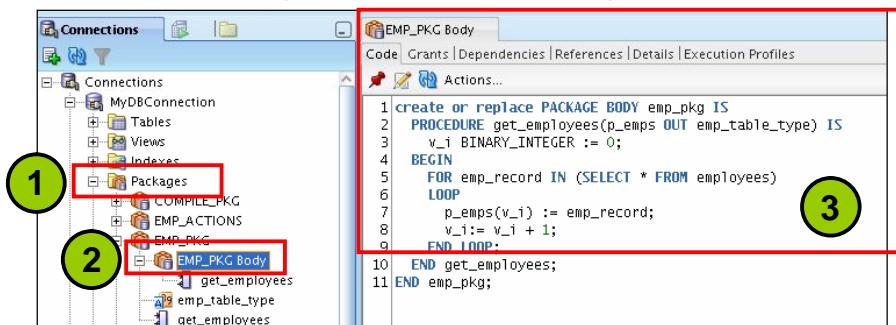
Note: You cannot display the source code for Oracle PL/SQL built-in packages, or PL/SQL whose source code has been wrapped by using a WRAP utility or obfuscation. Obfuscating and wrapping PL/SQL source code is covered in a later lesson. Clicking the Execute Statement (F9) icon (instead of the Run Script icon) in the SQL Worksheet toolbar, sometimes displays a better formatted output in the Results tab as shown in the slide examples.

Viewing Packages Using SQL Developer

To view the package spec, click the package name



To view the package body, click the package body



ORACLE®

Viewing Packages Using SQL Developer

To view a package's spec in SQL Developer, use the following steps:

1. Click the **Packages** node in the **Connections** tab.
2. Click the package's name.
3. The package's spec code is displayed in the **Code** tab as shown in the slide.

To view a package's body in SQL Developer, use the following steps:

1. Click the **Packages** node in the **Connections** tab.
2. Click the package's body.
3. The package's body code is displayed in the **Code** tab as shown in the slide.

Guidelines for Writing Packages

- Develop packages for general use.
- Define the package specification before the body.
- The package specification should contain only those constructs that you want to be public.
- Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions.
- The fine-grain dependency management reduces the need to recompile referencing subprograms when a package specification changes.
- The package specification should contain as few constructs as possible.

ORACLE®

3 - 23

Copyright © 2009, Oracle. All rights reserved.

Guidelines for Writing Packages

Keep your packages as general as possible, so that they can be reused in future applications. Also, avoid writing packages that duplicate features provided by the Oracle server.

Package specifications reflect the design of your application, so define them before defining the package bodies. The package specification should contain only those constructs that must be visible to the users of the package. Thus, other developers cannot misuse the package by basing code on irrelevant details.

Place items in the declaration part of the package body when you must maintain them throughout a session or across transactions. For example, declare a variable called NUMBER_EMPLOYED as a private variable if each call to a procedure that uses the variable needs to be maintained. When declared as a global variable in the package specification, the value of that global variable is initialized in a session the first time a construct from the package is invoked.

Before Oracle Database 11g, changes to the package body did not require recompilation of dependent constructs, whereas changes to the package specification required the recompilation of every stored subprogram that references the package. Oracle Database 11g reduces this dependency. Dependencies are now tracked at the level of element within unit. Fine-Grain Dependency Management is covered in a later lesson.

Quiz

The package specification is the interface to your applications. It declares the public types, variables, constants, exceptions, cursors, and subprograms available for use. The package specification may also include PRAGMAAs, which are directives to the compiler.

1. True
2. False



Answer: 1

Summary

In this lesson, you should have learned how to:

- Describe packages and list their components
- Create a package to group related variables, cursors, constants, exceptions, procedures, and functions
- Designate a package construct as either public or private
- Invoke a package construct
- Describe the use of a bodiless package



Summary

You group related procedures and functions in a package. Packages improve organization, management, security, and performance.

A package consists of a package specification and a package body. You can change a package body without affecting its package specification.

Packages enable you to hide source code from users. When you invoke a package for the first time, the entire package is loaded into memory. This reduces the disk access for subsequent calls.

Practice 3 Overview: Creating and Using Packages

This practice covers the following topics:

- Creating packages
- Invoking package program units



Practice 3: Overview

In this practice, you create package specifications and package bodies. You then invoke the constructs in the packages by using sample data.

Note: If you are using SQL Developer, your compile-time errors are displayed in the Message Log tab. If you are using SQL*Plus to create your stored code, use `SHOW ERRORS` to view compile errors.

Working with Packages



ORACLE®

Copyright © 2009, Oracle. All rights reserved.

Objectives

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

- Overload package procedures and functions
- Use forward declarations
- Create an initialization block in a package body
- Manage persistent package data states for the life of a session
- Use associative arrays (index-by tables) and records in packages



Lesson Aim

This lesson introduces the more advanced features of PL/SQL, including overloading, forward referencing, one-time-only procedures, and the persistency of variables, constants, exceptions, and cursors. It also explains the effect of packaging functions that are used in SQL statements.

Lesson Agenda

- Overloading package subprograms, using forward declarations, and creating an initialization block in a package body
- Managing persistent package data states for the life of a session and using associative arrays (index-by tables) and records in packages

ORACLE®

4 - 3

Copyright © 2009, Oracle. All rights reserved.

Overloading Subprograms in PL/SQL

- Enables you to create two or more subprograms with the same name
- Requires that the subprogram's formal parameters differ in number, order, or data type family
- Enables you to build flexible ways for invoking subprograms with different data
- Provides a way to extend functionality without loss of existing code; that is, adding new parameters to existing subprograms
- Provides a way to overload local subprograms, package subprograms, and type methods, but not stand-alone subprograms

ORACLE®

4 - 4

Copyright © 2009, Oracle. All rights reserved.

Overloading Subprograms

The overloading feature in PL/SQL enables you to develop two or more packaged subprograms with the same name. Overloading is useful when you want a subprogram to accept similar sets of parameters that have different data types. For example, the TO_CHAR function has more than one way to be called, enabling you to convert a number or a date to a character string.

PL/SQL allows overloading of package subprogram names and object type methods.

The key rule is that you can use the same name for different subprograms as long as their formal parameters differ in *number, order, or data type family*.

Consider using overloading when:

- Processing rules for two or more subprograms are similar, but the type or number of parameters used varies
- Providing alternative ways for finding different data with varying search criteria. For example, you may want to find employees by their employee ID and also provide a way to find employees by their last name. The logic is intrinsically the same, but the parameters or search criteria differ.
- Extending functionality when you do not want to replace existing code

Note: Stand-alone subprograms cannot be overloaded. Writing local subprograms in object type methods is not discussed in this course.

Overloading Subprograms (continued)

Restrictions

You cannot overload:

- Two subprograms if their formal parameters differ only in data type and the different data types are in the same family (NUMBER and DECIMAL belong to the same family.)
- Two subprograms if their formal parameters differ only in subtype and the different subtypes are based on types in the same family (VARCHAR and STRING are PL/SQL subtypes of VARCHAR2.)
- Two functions that differ only in return type, even if the types are in different families

You get a run-time error when you overload subprograms with the preceding features.

Note: The preceding restrictions apply if the names of the parameters are also the same.

If you use different names for the parameters, you can invoke the subprograms by using named notation for the parameters.

Resolving Calls

The compiler tries to find a declaration that matches the call. It searches first in the current scope and then, if necessary, in successive enclosing scopes. The compiler stops searching if it finds one or more subprogram declarations in which the name matches the name of the called subprogram. For similarly named subprograms at the same level of scope, the compiler needs an exact match in number, order, and data type between the actual and formal parameters.

Overloading Procedures Example: Creating the Package Specification

```
CREATE OR REPLACE PACKAGE dept_pkg IS
    PROCEDURE add_department
        (p_deptno departments.department_id%TYPE,
         p_name  departments.department_name%TYPE := 'unknown',
         p_loc   departments.location_id%TYPE := 1700);

    PROCEDURE add_department
        (p_name  departments.department_name%TYPE := 'unknown',
         p_loc   departments.location_id%TYPE := 1700);
END dept_pkg;
/
```

ORACLE®

4 - 6

Copyright © 2009, Oracle. All rights reserved.

Overloading: Example

The slide shows the dept_pkg package specification with an overloaded procedure called add_department. The first declaration takes three parameters that are used to provide data for a new department record inserted into the department table. The second declaration takes only two parameters because this version internally generates the department ID through an Oracle sequence.

It is better to specify data types using the %TYPE attribute for variables that are used to populate columns in database tables, as shown in the example in the slide; however, you can also specify the data types as follows:

```
CREATE OR REPLACE PACKAGE dept_pkg_method2 IS
    PROCEDURE add_department(p_deptno NUMBER,
                             p_name  VARCHAR2 := 'unknown', p_loc NUMBER := 1700);
    . . .
```

Overloading Procedures Example: Creating the Package Body

```
-- Package body of package defined on previous slide.  
CREATE OR REPLACE PACKAGE BODY dept_pkg IS  
PROCEDURE add_department -- First procedure's declaration  
(p_deptno departments.department_id%TYPE,  
 p_name   departments.department_name%TYPE := 'unknown',  
 p_loc    departments.location_id%TYPE := 1700) IS  
BEGIN  
    INSERT INTO departments(department_id,  
                           department_name, location_id)  
    VALUES (p_deptno, p_name, p_loc);  
END add_department;  
PROCEDURE add_department -- Second procedure's declaration  
(p_name   departments.department_name%TYPE := 'unknown',  
 p_loc    departments.location_id%TYPE := 1700) IS  
BEGIN  
    INSERT INTO departments (department_id,  
                           department_name, location_id)  
    VALUES (departments_seq.NEXTVAL, p_name, p_loc);  
END add_department;  
END dept_pkg; /
```

ORACLE®

4 - 7

Copyright © 2009, Oracle. All rights reserved.

Overloading: Example (continued)

If you call add_department with an explicitly provided department ID, then PL/SQL uses the first version of the procedure. Consider the following example:

```
EXECUTE dept_pkg.add_department(980,'Education',2500)  
SELECT * FROM departments  
WHERE department_id = 980;
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
980	Education		2500
1 rows selected			

If you call add_department with no department ID, PL/SQL uses the second version:

```
EXECUTE dept_pkg.add_department ('Training', 2400)  
SELECT * FROM departments  
WHERE department_name = 'Training';
```

DEPARTMENT_ID	DEPARTMENT_NAME	MANAGER_ID	LOCATION_ID
280	Training		2400
1 rows selected			

Overloading and the STANDARD Package

- A package named STANDARD defines the PL/SQL environment and built-in functions.
- Most built-in functions are overloaded. An example is the TO_CHAR function:

```
FUNCTION TO_CHAR (p1 DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (p2 NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (p1 DATE, p2 VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (p1 NUMBER, p2 VARCHAR2) RETURN
    VARCHAR2;
.
.
```

- A PL/SQL subprogram with the same name as a built-in subprogram overrides the standard declaration in the local context, unless qualified by the package name.

ORACLE®

4 - 8

Copyright © 2009, Oracle. All rights reserved.

Overloading and the STANDARD Package

A package named STANDARD defines the PL/SQL environment and globally declares types, exceptions, and subprograms that are available automatically to PL/SQL programs. Most of the built-in functions that are found in the STANDARD package are overloaded. For example, the TO_CHAR function has four different declarations, as shown in the slide. The TO_CHAR function can take either the DATE or the NUMBER data type and convert it to the character data type. The format to which the date or number has to be converted can also be specified in the function call.

If you re-declare a built-in subprogram in another PL/SQL program, then your local declaration overrides the standard or built-in subprogram. To be able to access the built-in subprogram, you must qualify it with its package name. For example, if you re-declare the TO_CHAR function to access the built-in function, you refer to it as STANDARD.TO_CHAR.

If you re-declare a built-in subprogram as a stand-alone subprogram, then, to access your subprogram, you must qualify it with your schema name: for example, SCOTT.TO_CHAR.

In the example in the slide, PL/SQL resolves a call to TO_CHAR by matching the number and data types of the formal and actual parameters.

Illegal Procedure Reference

- Block-structured languages such as PL/SQL must declare identifiers before referencing them.
- Example of a referencing problem:

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
  PROCEDURE award_bonus(. . .) IS
    BEGIN
      calc_rating(. . .);      --illegal reference
    END;

  PROCEDURE calc_rating(. . .) IS
    BEGIN
      . .
    END;
END forward_pkg;
/
```

ORACLE®

4 - 9

Copyright © 2009, Oracle. All rights reserved.

Using Forward Declarations

In general, PL/SQL is like other block-structured languages and does not allow forward references. You must declare an identifier before using it. For example, a subprogram must be declared before you can call it.

Coding standards often require that subprograms be kept in alphabetical sequence to make them easy to find. In this case, you may encounter problems, as shown in the slide, where the `calc_rating` procedure cannot be referenced because it has not yet been declared.

You can solve the illegal reference problem by reversing the order of the two procedures. However, this easy solution does not work if the coding rules require subprograms to be declared in alphabetical order.

The solution in this case is to use forward declarations provided in PL/SQL. A forward declaration enables you to declare the heading of a subprogram, that is, the subprogram specification terminated by a semicolon.

Note: The compilation error for `calc_rating` occurs only if `calc_rating` is a private packaged procedure. If `calc_rating` is declared in the package specification, it is already declared as if it is a forward declaration, and its reference can be resolved by the compiler.

Using Forward Declarations to Solve Illegal Procedure Reference

In the package body, a forward declaration is a private subprogram specification terminated by a semicolon.

```
CREATE OR REPLACE PACKAGE BODY forward_pkg IS
  PROCEDURE calc_rating (...) -- forward declaration
    -- Subprograms defined in alphabetical order
    PROCEDURE award_bonus (...) IS
      BEGIN
        calc_rating (...); -- reference resolved!
        . . .
      END;
    PROCEDURE calc_rating (...) IS -- implementation
      BEGIN
        . . .
      END;
  END forward_pkg;
```

ORACLE®

4 - 10

Copyright © 2009, Oracle. All rights reserved.

Using Forward Declarations (continued)

As previously mentioned, PL/SQL enables you to create a special subprogram declaration called a forward declaration. A forward declaration may be required for private subprograms in the package body, and consists of the subprogram specification terminated by a semicolon. Forward declarations help to:

- Define subprograms in logical or alphabetical order
- Define mutually recursive subprograms. Mutually recursive programs are programs that call each other directly or indirectly.
- Group and logically organize subprograms in a package body

When creating a forward declaration:

- The formal parameters must appear in both the forward declaration and the subprogram body
- The subprogram body can appear anywhere after the forward declaration, but both must appear in the same program unit

Forward Declarations and Packages

Typically, the subprogram specifications go in the package specification, and the subprogram bodies go in the package body. The public subprogram declarations in the package specification do not require forward declarations.

Initializing Packages

The block at the end of the package body executes once and is used to initialize public and private package variables.

```
CREATE OR REPLACE PACKAGE taxes IS
    v_tax    NUMBER;
    ...
    -- declare all public procedures/functions
END taxes;
/
CREATE OR REPLACE PACKAGE BODY taxes IS
    ...
    -- declare all private variables
    ...
    -- define public/private procedures/functions
BEGIN
    SELECT    rate_value INTO v_tax
    FROM      tax_rates
    WHERE     rate_name = 'TAX';
END taxes;
/
```

ORACLE®

4 - 11

Copyright © 2009, Oracle. All rights reserved.

Package Initialization Block

The first time a component in a package is referenced, the entire package is loaded into memory for the user session. By default, the initial value of variables is NULL (if not explicitly initialized). To initialize package variables, you can:

- Use assignment operations in their declarations for simple initialization tasks
- Add code block to the end of a package body for more complex initialization tasks

Consider the block of code at the end of a package body as a package initialization block that executes once, when the package is first invoked within the user session.

The example in the slide shows the v_tax public variable being initialized to the value in the tax_rates table the first time the taxes package is referenced.

Note: If you initialize the variable in the declaration by using an assignment operation, it is overwritten by the code in the initialization block at the end of the package body. The initialization block is terminated by the END keyword for the package body.

Using Package Functions in SQL

- You use package functions in SQL statements.
- To execute a SQL statement that calls a member function, the Oracle database must know the function's purity level.
- Purity level is the extent to which the function is free of side effects, which refers to accessing database tables, package variables, and so on, for reading or writing.
- It is important to control side effects because they can:
 - Prevent the proper parallelization of a query
 - Produce order-dependent and, therefore, indeterminate results
 - Require impermissible actions such as the maintenance of package state across user sessions

ORACLE®

4 - 12

Copyright © 2009, Oracle. All rights reserved.

Using Package Functions in SQL and Restrictions

To execute a SQL statement that calls a stored function, the Oracle Server must know the purity level of the function, or the extent to which the function is free of side effects. The term *side effect* refers to accessing database tables, package variables, and so forth for reading or writing. It is important to control side effects because they can prevent the proper parallelization of a query, produce order-dependent and therefore indeterminate results, or require impermissible actions such as the maintenance of package state across user sessions.

In general, restrictions are changes to database tables or public package variables (those declared in a package specification). Restrictions can delay the execution of a query, yield order-dependent (therefore indeterminate) results, or require that the package state variables be maintained across user sessions. Various restrictions are not allowed when a function is called from a SQL query or a DML statement.

Controlling Side Effects of PL/SQL Subprograms

To be callable from SQL statements, a stored function must obey the following purity rules to control side effects:

- When called from a SELECT or a parallelized DML statement, the function cannot modify any database tables.
- When called from a DML statement, the function cannot query or modify any database tables modified by that statement.
- When called from a SELECT or DML statement, the function cannot execute SQL transaction control, session control, or system control statements.



Controlling Side Effects of PL/SQL Subprograms

The fewer side effects a function has, the better it can be optimized within a query, particularly when the PARALLEL_ENABLE or DETERMINISTIC hints are used.

To be callable from SQL statements, a stored function (and any subprograms that it calls) must obey the purity rules listed in the slide. The purpose of those rules is to control side effects.

If any SQL statement inside the function body violates a rule, you get an error at run time (when the statement is parsed).

To check for purity rule violations at compile time, use the RESTRICT_REFERENCES pragma to assert that a function does not read or write database tables or package variables.

Note

- In the slide, a DML statement refers to an INSERT, UPDATE, or DELETE statement.
- For information about using the RESTRICT_REFERENCES pragma, refer to the *Oracle Database PL/SQL Language Reference 11g Release 2 (11.2)*.

Package Function in SQL: Example

```
CREATE OR REPLACE PACKAGE taxes_pkg IS
    FUNCTION tax (p_value IN NUMBER) RETURN NUMBER;
END taxes_pkg;
/
CREATE OR REPLACE PACKAGE BODY taxes_pkg IS
    FUNCTION tax (p_value IN NUMBER) RETURN NUMBER IS
        v_rate NUMBER := 0.08;
    BEGIN
        RETURN (p_value * v_rate);
    END tax;
END taxes_pkg;
/
```

```
SELECT taxes_pkg.tax(salary), salary, last_name
FROM employees;
```

4 - 14

Copyright © 2009, Oracle. All rights reserved.

ORACLE®

Package Function in SQL: Example

The first code example in the slide shows how to create the package specification and the body encapsulating the tax function in the `taxes_pkg` package. The second code example shows how to call the packaged tax function in the `SELECT` statement. The results are as follows:

TAXES_PKG.TAX(SALARY)	SALARY	LAST_NAME
1920	24000	King
1360	17000	Kochhar
1360	17000	De Haan
720	9000	Hunold
480	6000	Ernst
384	4800	Austin
384	4800	Pataballa
336	4200	Lorentz
960	12000	Greenberg
...		

107 rows selected

Lesson Agenda

- Overloading package subprograms, using forward declarations, and creating an initialization block in a package body
- Managing persistent package data states for the life of a session and using associative arrays (index-by tables) and records in packages

ORACLE®

4 - 15

Copyright © 2009, Oracle. All rights reserved.

Persistent State of Packages

The collection of package variables and the values define the package state. The package state is:

- Initialized when the package is first loaded
- Persistent (by default) for the life of the session:
 - Stored in the User Global Area (UGA)
 - Unique to each session
 - Subject to change when package subprograms are called or public variables are modified
- Not persistent for the session but persistent for the life of a subprogram call when using PRAGMA SERIALLY_REUSEABLE in the package specification



Persistent State of Packages

The collection of public and private package variables represents the package state for the user session. That is, the package state is the set of values stored in all the package variables at a given point in time. In general, the package state exists for the life of the user session.

Package variables are initialized the first time a package is loaded into memory for a user session. The package variables are, by default, unique to each session and hold their values until the user session is terminated. In other words, the variables are stored in the User Global Area (UGA) memory allocated by the database for each user session. The package state changes when a package subprogram is invoked and its logic modifies the variable state. Public package state can be directly modified by operations appropriate to its type.

PRAGMA signifies that the statement is a compiler directive. PRAGMAS are processed at compile time, not at run time. They do not affect the meaning of a program; they simply convey information to the compiler. If you add PRAGMA SERIALLY_REUSEABLE to the package specification, then the database stores package variables in the System Global Area (SGA) shared across user sessions. In this case, the package state is maintained for the life of a subprogram call or a single reference to a package construct. The SERIALLY_REUSEABLE directive is useful if you want to conserve memory and if the package state does not need to persist for each user session.

Persistent State of Packages (continued)

This PRAGMA is appropriate for packages that declare large temporary work areas that are used once and not needed during subsequent database calls in the same session.

You can mark a bodiless package as serially reusable. If a package has a spec and body, you must mark both. You cannot mark only the body.

The global memory for serially reusable packages is pooled in the System Global Area (SGA), not allocated to individual users in the User Global Area (UGA). That way, the package work area can be reused. When the call to the server ends, the memory is returned to the pool. Each time the package is reused, its public variables are initialized to their default values or to NULL.

Note: Serially reusable packages cannot be accessed from database triggers or other PL/SQL subprograms that are called from SQL statements. If you try, the Oracle server generates an error.

Persistent State of Package Variables: Example

Time	Events	State for Scott		State for Jones	
		v_std_comm [variable]	MAX (commission_pct) [column]	v_std_comm [variable]	MAX (commission_pct) [Column]
9:00	<i>Scott> EXECUTE comm_pkg.reset_comm(0.25)</i>	0.10 0.25	0.4	-	0.4
9:30	<i>Jones> INSERT INTO employees(last_name, commission_pct) VALUES('Madonna', 0.8);</i>	0.25	0.4		0.8
9:35	<i>Jones> EXECUTE comm_pkg.reset_comm (0.5)</i>	0.25	0.4	0.10 0.5	0.8
10:00	<i>Scott> EXECUTE comm_pkg.reset_comm(0.6) Err -20210 'Bad Commission'</i>	0.25	0.4	0.5	0.8
11:00 11:01 12:00	<i>Jones> ROLLBACK; EXIT ... EXEC comm_pkg.reset_comm(0.2)</i>	0.25 0.25 0.25	0.4 0.4 0.4	0.5 - 0.2	0.4 0.4 0.4

ORACLE®

4 - 18

Copyright © 2009, Oracle. All rights reserved.

Persistent State of Package Variables: Example

The slide sequence is based on two different users, Scott and Jones, executing `comm_pkg` (covered in the lesson titled “Creating Packages”), in which the `reset_comm` procedure invokes the `validate` function to check the new commission. The example shows how the persistent state of the `v_std_comm` package variable is maintained in each user session.

At 9:00: Scott calls `reset_comm` with a new commission value of 0.25, the package state for `v_std_comm` is initialized to 0.10 and then set to 0.25, which is validated because it is less than the database maximum value of 0.4.

At 9:30: Jones inserts a new row into the `EMPLOYEES` table with a new maximum `v_commission_pct` value of 0.8. This is not committed, so it is visible to Jones only. Scott’s state is unaffected.

At 9:35: Jones calls `reset_comm` with a new commission value of 0.5. The state for Jones’s `v_std_comm` is first initialized to 0.10 and then set to the new value 0.5 that is valid for his session with the database maximum value of 0.8.

At 10:00: Scott calls `reset_comm` with a new commission value of 0.6, which is greater than the maximum database commission visible to his session, that is, 0.4. (Jones did not commit the 0.8 value.)

Between 11:00 and 12:00: Jones rolls back the transaction (`INSERT` statement) and exits the session. Jones logs in at 11:45 and successfully executes the procedure, setting his state to 0.2.

Persistent State of a Package Cursor: Example

```
CREATE OR REPLACE PACKAGE curs_pkg IS -- Package spec
  PROCEDURE open;
  FUNCTION next(p_n NUMBER := 1) RETURN BOOLEAN;
  PROCEDURE close;
END curs_pkg;

CREATE OR REPLACE PACKAGE BODY curs_pkg IS
  -- Package body
  CURSOR cur_c IS
    SELECT employee_id FROM employees;
  PROCEDURE open IS
  BEGIN
    IF NOT cur_c%ISOPEN THEN
      OPEN cur_c;
    END IF;
  END open;
  . . . -- code continued on next slide
```

ORACLE®

4 - 19

Copyright © 2009, Oracle. All rights reserved.

Persistent State of a Package Cursor: Example

The example in the slide shows the CURS_PKG package specification and body. The body declaration is continued in the next slide.

To use this package, perform the following steps to process the rows:

1. Call the open procedure to open the cursor.

Persistent State of a Package Cursor: Example

```
 . . .
FUNCTION next(p_n NUMBER := 1) RETURN BOOLEAN IS
    v_emp_id employees.employee_id%TYPE;
BEGIN
    FOR count IN 1 .. p_n LOOP
        FETCH cur_c INTO v_emp_id;
        EXIT WHEN cur_c%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Id: ' || (v_emp_id));
    END LOOP;
    RETURN cur_c%FOUND;
END next;
PROCEDURE close IS
BEGIN
    IF cur_c%ISOPEN THEN
        CLOSE cur_c;
    END IF;
END close;
END curs_pkg;
```

ORACLE®

4 - 20

Copyright © 2009, Oracle. All rights reserved.

Persistent State of a Package Cursor: Example (continued)

2. Call the `next` procedure to fetch one or a specified number of rows. If you request more rows than actually exist, the procedure successfully handles termination.
It returns TRUE if more rows need to be processed; otherwise it returns FALSE.
3. Call the `close` procedure to close the cursor, before or at the end of processing the rows.

Note: The cursor declaration is private to the package. Therefore, the cursor state can be influenced by invoking the package procedure and functions listed in the slide.

Executing the CURS_PKG Package

The screenshot shows two windows from Oracle SQL Developer. The top window is a code editor with the following PL/SQL code:

```
1 SET SERVEROUTPUT ON
2
3 EXECUTE curs_pkg.open
4 DECLARE
5   v_more BOOLEAN := curs_pkg.next(3);
6 BEGIN
7   IF NOT v_more THEN
8     curs_pkg.close;
9   END IF;
10 END;
11 /
```

The bottom window is the Script Output tab, displaying the results of the execution:

```
anonymous block completed
anonymous block completed
Id: 100
Id: 101
Id: 102

anonymous block completed
anonymous block completed
Id: 103
Id: 104
Id: 105
```

ORACLE®

4 - 21

Copyright © 2009, Oracle. All rights reserved.

Executing CURS_PKG

Recall that the state of a package variable or cursor persists across transactions within a session. However, the state does not persist across different sessions for the same user. The database tables hold data that persists across sessions and users. The call to `curs_pkg.open` opens the cursor, which remains open until the session is terminated, or the cursor is explicitly closed. The anonymous block executes the `next` function in the Declaration section, initializing the `BOOLEAN` variable `b_more` to `TRUE`, as there are more than three rows in the `EMPLOYEES` table. The block checks for the end of the result set and closes the cursor, if appropriate. When the block executes, it displays the first three rows:

```
Id :100
Id :101
Id :102
```

If you click the Run Script (F5) icon again, the next three rows are displayed:

```
Id :103
Id :104
Id :105
```

To close the cursor, you can issue the following command to close the cursor in the package, or exit the session:

```
EXECUTE curs_pkg.close
```

Using Associative Arrays in Packages

```
CREATE OR REPLACE PACKAGE emp_pkg IS
    TYPE emp_table_type IS TABLE OF employees%ROWTYPE
        INDEX BY BINARY_INTEGER;
    PROCEDURE get_employees(p_emps OUT emp_table_type);
END emp_pkg;
```

```
CREATE OR REPLACE PACKAGE BODY emp_pkg IS
    PROCEDURE get_employees(p_emps OUT emp_table_type) IS
        v_i BINARY_INTEGER := 0;
    BEGIN
        FOR emp_record IN (SELECT * FROM employees)
        LOOP
            emps(v_i) := emp_record;
            v_i:= v_i + 1;
        END LOOP;
    END get_employees;
END emp_pkg;
```

ORACLE®

4 - 22

Copyright © 2009, Oracle. All rights reserved.

Using Associative Arrays in Packages

Associative arrays used to be known as index by tables.

The `emp_pkg` package contains a `get_employees` procedure that reads rows from the `EMPLOYEES` table and returns the rows using the `OUT` parameter, which is an associative array (PL/SQL table of records). The key points include the following:

- `employee_table_type` is declared as a public type.
- `employee_table_type` is used for a formal output parameter in the procedure, and the `employees` variable in the calling block (shown below).

In SQL Developer, you can invoke the `get_employees` procedure in an anonymous PL/SQL block by using the `v_employees` variable, as shown in the following example and output:

```
SET SERVEROUTPUT ON
DECLARE
    v_employees    emp_pkg.emp_table_type;
BEGIN
    emp_pkg.get_employees(v_employees);
    DBMS_OUTPUT.PUT_LINE('Emp 5: ' || v_employees(4).last_name);
END;
anonymous block completed
Emp 5: Ernst
```

Quiz

Overloading subprograms in PL/SQL:

1. Enables you to create two or more subprograms with the same name
2. Requires that the subprogram's formal parameters differ in number, order, or data type family
3. Enables you to build flexible ways for invoking subprograms with different data
4. Provides a way to extend functionality without loss of existing code; that is, adding new parameters to existing subprograms

ORACLE®

4 - 23

Copyright © 2009, Oracle. All rights reserved.

Answers: 1, 2, 3, 4

The overloading feature in PL/SQL enables you to develop two or more packaged subprograms with the same name. Overloading is useful when you want a subprogram to accept similar sets of parameters that have different data types. For example, the TO_CHAR function has more than one way to be called, enabling you to convert a number or a date to a character string.

PL/SQL allows overloading of package subprogram names and object type methods.

The key rule is that you can use the same name for different subprograms as long as their formal parameters differ in *number, order, or data type family*.

Consider using overloading when:

- Processing rules for two or more subprograms are similar, but the type or number of parameters used varies
- Providing alternative ways for finding different data with varying search criteria. For example, you may want to find employees by their employee ID and also provide a way to find employees by their last name. The logic is intrinsically the same, but the parameters or search criteria differ.
- Extending functionality when you do not want to replace existing code

Summary

In this lesson, you should have learned how to:

- Overload package procedures and functions
- Use forward declarations
- Create an initialization block in a package body
- Manage persistent package data states for the life of a session
- Use associative arrays (index-by tables) and records in packages



Summary

Overloading is a feature that enables you to define different subprograms with the same name. It is logical to give two subprograms the same name when the processing in both the subprograms is the same but the parameters passed to them vary.

PL/SQL permits a special subprogram declaration called a forward declaration. A forward declaration enables you to define subprograms in logical or alphabetical order, define mutually recursive subprograms, and group subprograms in a package.

A package initialization block is executed only when the package is first invoked within the other user session. You can use this feature to initialize variables only once per session.

You can keep track of the state of a package variable or cursor, which persists throughout the user session, from the time the user first references the variable or cursor to the time the user disconnects.

Using the PL/SQL wrapper, you can obscure the source code stored in the database to protect your intellectual property.

Practice 4: Overview

This practice covers the following topics:

- Using overloaded subprograms
- Creating a package initialization block
- Using a forward declaration



Practice 4: Overview

In this practice, you modify an existing package to contain overloaded subprograms and you use forward declarations. You also create a package initialization block within a package body to populate a PL/SQL table.

Using Oracle-Supplied Packages in Application Development

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

Objectives

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

- Describe how the DBMS_OUTPUT package works
- Use UTL_FILE to direct output to operating system files
- Describe the main features of UTL_MAIL



Lesson Aim

In this lesson, you learn how to use some of the Oracle-supplied packages and their capabilities.

Lesson Agenda

- Identifying the benefits of using the Oracle-supplied packages and listing some of those packages
- Using the following Oracle-supplied packages:
 - DBMS_OUTPUT
 - UTL_FILE
 - UTL_MAIL



Using Oracle-Supplied Packages

- The 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.

ORACLE®

5 - 4

Copyright © 2009, Oracle. All rights reserved.

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`. The `DBMS_OUTPUT` package is the one that you will be most familiar with during this course. You should already be familiar with this package if you attended the *Oracle Database 11g: PL/SQL Fundamentals* course.

Examples of Some Oracle-Supplied Packages

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

- DBMS_OUTPUT
- UTL_FILE
- UTL_MAIL
- DBMS_ALERT
- DBMS_LOCK
- DBMS_SESSION
- DBMS_APPLICATION_INFO
- HTP
- DBMS_SCHEDULER



List of Some Oracle-Supplied Packages

The list of PL/SQL packages provided with an Oracle database grows with the release of new versions. This lesson covers the first three packages in the slide. For more information, refer to the *Oracle Database PL/SQL Packages and Types Reference 11g Release 2 (11.2)*. The following is a brief description about the remaining listed packages in the slide:

- DBMS_OUTPUT provides debugging and buffering of text data.
- UTL_FILE enables reading and writing of operating system text files.
- UTL_MAIL enables composing and sending of email messages.
- DBMS_ALERT supports asynchronous notification of database events. Messages or alerts are sent on a COMMIT command.
- DBMS_LOCK is used to request, convert, and release locks through Oracle Lock Management services.
- DBMS_SESSION enables programmatic use of the ALTER SESSION SQL statement and other session-level commands.
- DBMS_APPLICATION_INFO can be used with Oracle Trace and the SQL trace facility to record names of executing modules or transactions in the database for later use when tracking the performance of various modules and debugging.
- HTP package writes HTML-tagged data into database buffers.
- DBMS_SCHEDULER enables scheduling and automated execution of PL/SQL blocks, stored procedures, and external procedures and executables (covered in Appendix G).

Lesson Agenda

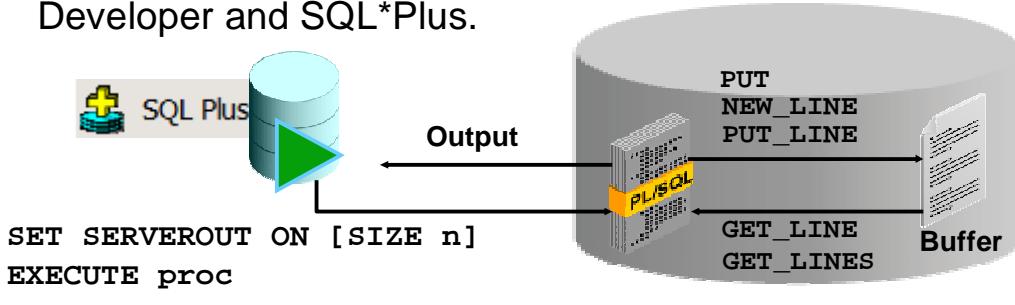
- Identifying the benefits of using the Oracle-supplied packages and listing some of those packages
- Using the following Oracle-supplied packages:
 - DBMS_OUTPUT
 - UTL_FILE
 - UTL_MAIL

ORACLE®

How the DBMS_OUTPUT Package Works

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.
- Messages are not sent until the sending subprogram or trigger completes.
- Use SET SERVEROUTPUT ON to display messages in SQL Developer and SQL*Plus.



5 - 7

Copyright © 2009, Oracle. All rights reserved.

ORACLE®

How the DBMS_OUTPUT Package Works

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

- PUT appends text from the procedure to the current line of the line output buffer.
- NEW_LINE places an end-of-line marker in the output buffer.
- PUT_LINE combines the action of PUT and NEW_LINE (to trim leading spaces).
- GET_LINE retrieves the current line from the buffer into a procedure variable.
- GET_LINES retrieves an array of lines into a procedure-array variable.
- ENABLE/DISABLE enables and disables calls to 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 the number of characters. The minimum is 2,000 and the maximum is unlimited. The default is 20,000.
- An integer parameter between 2,000 and 1,000,000 in the ENABLE procedure

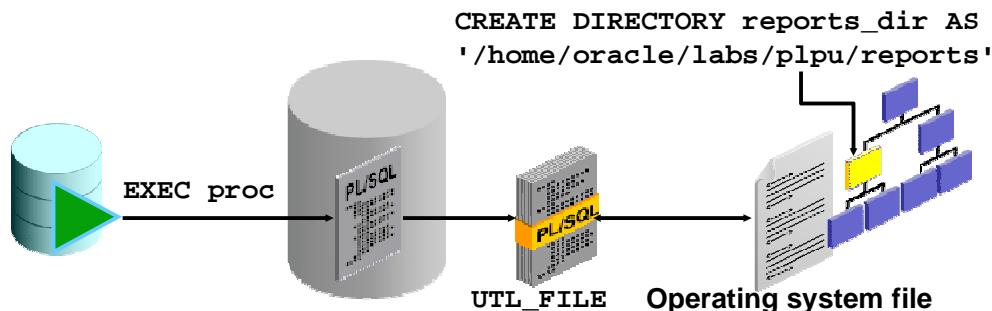
You can output results to the window for debugging purposes. You can trace a code execution path for a function or procedure. You can send messages between subprograms and triggers.

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

Using the UTL_FILE Package to Interact with Operating System Files

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

- Provides a restricted version of operating system stream file I/O for text files
- Can access files in operating system directories defined by a CREATE DIRECTORY statement



Interacting with Operating System Files

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.
```
- The paths specified in the utl_file_dir database initialization parameter

It is recommended that you use the CREATE DIRECTORY feature instead of UTL_FILE_DIR for directory access verification. Directory objects offer more flexibility and granular control to the UTL_FILE application administrator, can be maintained dynamically (that is, without shutting down the database), and are consistent with other Oracle tools. The CREATE DIRECTORY privilege is granted only to SYS and SYSTEM by default.

The operating system directories specified by using either of these techniques 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.

Some of the UTL_FILE Procedures and Functions

Subprogram	Description
ISOPEN function	Determines if a file handle refers to an open file
FOPEN function	Opens a file for input or output
FCLOSE function	Closes all open file handles
FCOPY procedure	Copies a contiguous portion of a file to a newly created file
FGETATTR procedure	Reads and returns the attributes of a disk file
GET_LINE procedure	Reads text from an open file
FREMOVE procedure	Deletes a disk file, if you have sufficient privileges
FRENAME procedure	Renames an existing file to a new name
PUT procedure	Writes a string to a file
PUT_LINE procedure	Writes a line to a file, and so appends an operating system-specific line terminator

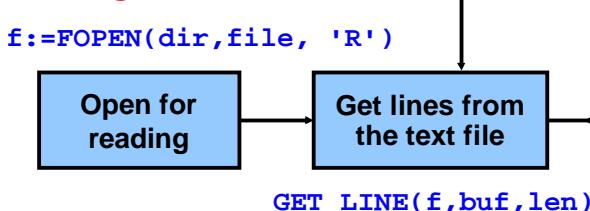
ORACLE®

Some of the UTL_FILE Procedures and Functions

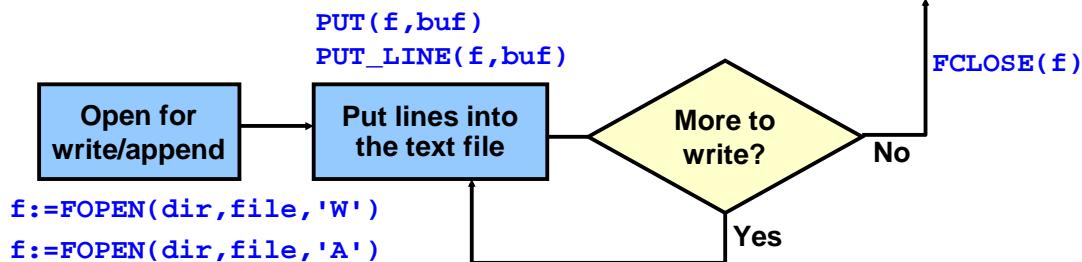
The table in the slide lists some of the UTL_FILE Package subprograms. For a complete list of the package's subprograms, see the *Oracle Database PL/SQL Packages and Types Reference 11g Release 2 (11.2) guide*.

File Processing Using the UTL_FILE Package: Overview

Reading a file



Writing or appending to a file



ORACLE®

File Processing Using the UTL_FILE Package

You can use the procedures and functions in the UTL_FILE package to open files with the FOPEN function. You can then either read from or write or append to the file until processing is done. After you finish processing the file, close the file by using the FCLOSE procedure. The following are the subprograms:

- The FOPEN function opens a file in a specified directory for input/output (I/O) and returns a file handle used in subsequent I/O operations.
- The IS_OPEN function returns a Boolean value whenever a file handle refers to an open file. Use IS_OPEN to check whether the file is already open before opening the file.
- The GET_LINE procedure reads a line of text from the file into an output buffer parameter. (The maximum input record size is 1,023 bytes unless you specify a larger size in the overloaded version of FOPEN.)
- The PUT and PUT_LINE procedures write text to the opened file.
- The PUTF procedure provides formatted output with two format specifiers: %s to substitute a value into the output string and \n for a new line character.
- The NEW_LINE procedure terminates a line in an output file.
- The FFLUSH procedure writes all data buffered in memory to a file.
- The FCLOSE procedure closes an opened file.
- The FCLOSE_ALL procedure closes all opened file handles for the session.

Using the Available Declared Exceptions in the UTL_FILE Package

Exception Name	Description
INVALID_PATH	File location invalid
INVALID_MODE	The <code>open_mode</code> parameter in <code>FOPEN</code> is invalid
INVALID_FILEHANDLE	File handle invalid
INVALID_OPERATION	File could not be opened or operated on as requested
READ_ERROR	Operating system error occurred during the read operation
WRITE_ERROR	Operating system error occurred during the write operation
INTERNAL_ERROR	Unspecified PL/SQL error

ORACLE®

5 - 11

Copyright © 2009, Oracle. All rights reserved.

Exceptions in the UTL_FILE Package

The UTL_FILE package declares fifteen exceptions that indicate an error condition in the operating system file processing. You may have to handle one of these exceptions when using UTL_FILE subprograms.

A subset of the exceptions are displayed in the slide. For additional information about the remaining exceptions, refer to the *Oracle Database PL/SQL Packages and Types Reference 11g Release 2 (11.2)* guide.

Note: These exceptions must always be prefixed with the package name. UTL_FILE procedures can also raise predefined PL/SQL exceptions such as NO_DATA_FOUND or VALUE_ERROR.

The NO_DATA_FOUND exception is raised when reading past the end of a file by using UTL_FILE.GET_LINE or UTL_FILE.GET_LINES.

FOPEN and IS_OPEN Functions: Example

- This FOPEN function opens a file for input or output.

```
FUNCTION FOPEN (p_location  IN VARCHAR2,
                p_filename   IN VARCHAR2,
                p_open_mode  IN VARCHAR2)
RETURN UTL_FILE.FILE_TYPE;
```

- The IS_OPEN function determines whether a file handle refers to an open file.

```
FUNCTION IS_OPEN (p_file IN FILE_TYPE)
RETURN BOOLEAN;
```

ORACLE®

5 - 12

Copyright © 2009, Oracle. All rights reserved.

FOPEN and IS_OPEN Function Parameters: Example

The parameters include the following:

- p_location parameter: Specifies the name of a directory alias defined by a CREATE DIRECTORY statement, or an operating system-specific path specified by using the utl_file_dir database parameter
- p_filename parameter: Specifies the name of the file, including the extension, without any path information
- open_mode string: Specifies how the file is to be opened. Values are:
 - 'R' for reading text (use GET_LINE)
 - 'W' for writing text (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH)
 - 'A' for appending text (PUT, PUT_LINE, NEW_LINE, PUTF, FFLUSH)

The return value from FOPEN is a file handle whose type is UTL_FILE.FILE_TYPE. The handle must be used on subsequent calls to routines that operate on the opened file.

The IS_OPEN function parameter is the file handle. The IS_OPEN function tests a file handle to see whether it identifies an opened file. It returns a Boolean value of TRUE if the file has been opened; otherwise it returns a value of FALSE indicating that the file has not been opened. The example in the slide shows how to combine the use of the two subprograms. For the full syntax, refer to the *Oracle Database PL/SQL Packages and Types Reference 11g Release 2 (11.2)* guide.

FOPEN and IS_OPEN Function Parameters: Example (continued)

```
CREATE OR REPLACE PROCEDURE read_file(p_dir VARCHAR2,
p_filename VARCHAR2) IS
  f_file UTL_FILE.FILE_TYPE;
  v_buffer VARCHAR2(200);
  v_lines  PLS_INTEGER := 0;
BEGIN
  DBMS_OUTPUT.PUT_LINE(' Start ');
  IF NOT UTL_FILE.IS_OPEN(f_file) THEN
    DBMS_OUTPUT.PUT_LINE(' Open ');
    f_file := UTL_FILE.FOPEN (p_dir, p_filename, 'R');
    DBMS_OUTPUT.PUT_LINE(' Opened ');
  BEGIN
    LOOP
      UTL_FILE.GET_LINE(f_file, v_buffer);
      v_lines := v_lines + 1;
      DBMS_OUTPUT.PUT_LINE(TO_CHAR(v_lines, '099') || |
                           ' | ' || buffer);
    END LOOP;
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE(' ** End of File **');
  END; -- ends Begin
  DBMS_OUTPUT.PUT_LINE(v_lines || ' lines read from file');
  UTL_FILE.FCLOSE(f_file);
END IF;
END read_file;
/
SHOW ERRORS
EXECUTE read_file('REPORTS_DIR', 'instructor.txt')
```

The partial output of the above code is as follows:

```
PROCEDURE read_file(dir Compiled.
No Errors.
line 27: SQLPLUS Command Skipped: set serveroutput on
anonymous block completed
Start
Open
Opened
001 SALARY REPORT: GENERATED ON
002                         08-MAR-01
003
004 DEPARTMENT: 10
005   EMPLOYEE: Whalen earns: 4400
```

...

```
120 DEPARTMENT: 110
121   EMPLOYEE: Higgins earns: 12000
122   EMPLOYEE: Gietz earns: 8300
123   EMPLOYEE: Grant earns: 7000
124 *** END OF REPORT ***
** End of File **
124 lines read from file
```

Using UTL_FILE: Example

```
CREATE OR REPLACE PROCEDURE sal_status(
    p_dir IN VARCHAR2, p_filename IN VARCHAR2) IS
    f_file UTL_FILE.FILE_TYPE;
    CURSOR cur_emp IS
        SELECT last_name, salary, department_id
        FROM employees ORDER BY department_id;
    v_newdeptno employees.department_id%TYPE;
    v_olddeptno employees.department_id%TYPE := 0;
BEGIN
    f_file:= UTL_FILE.FOPEN (p_dir, p_filename, 'W');
    UTL_FILE.PUT_LINE(f_file,
        'REPORT: GENERATED ON ' || SYSDATE);
    UTL_FILE.NEW_LINE (f_file);
    ...

```

ORACLE®

5 - 14

Copyright © 2009, Oracle. All rights reserved.

Using UTL_FILE: Example

In the slide example, the `sal_status` procedure creates a report of employees for each department, along with their salaries. The data is written to a text file by using the `UTL_FILE` package. In the code example, the `file` variable is declared as `UTL_FILE.FILE_TYPE`, a package type that is a record with a field called `ID` of the `BINARY_INTEGER` data type. For example:

```
TYPE file_type IS RECORD (id BINARY_INTEGER);
```

The field of `FILE_TYPE` record is private to the `UTL_FILE` package and should never be referenced or changed. The `sal_status` procedure accepts two parameters:

- The `p_dir` parameter for the name of the directory in which to write the text file
- The `p_filename` parameter to specify the name of the file

For example, to call the procedure, use the following:

```
EXECUTE sal_status('REPORTS_DIR', 'salreport2.txt')
```

Note: The directory location used (`REPORTS_DIR`) must be in uppercase characters if it is a directory alias created by a `CREATE DIRECTORY` statement. When reading a file in a loop, the loop should exit when it detects the `NO_DATA_FOUND` exception. The `UTL_FILE` output is sent synchronously. A `DBMS_OUTPUT` procedure does not produce an output until the procedure is completed.

Using UTL_FILE: Example

```
...
FOR emp_rec IN cur_emp LOOP
    IF emp_rec.department_id <> v_olddeptno THEN
        UTL_FILE.PUT_LINE (f_file,
            'DEPARTMENT: ' || emp_rec.department_id);
        UTL_FILE.NEW_LINE (f_file);
    END IF;
    UTL_FILE.PUT_LINE (f_file,
        ' EMPLOYEE: ' || emp_rec.last_name ||
        ' earns: ' || emp_rec.salary);
    v_olddeptno := emp_rec.department_id;
    UTL_FILE.NEW_LINE (f_file);
END LOOP;
UTL_FILE.PUT_LINE(f_file,'*** END OF REPORT ***');
UTL_FILE.FCLOSE (f_file);
EXCEPTION
    WHEN UTL_FILE.INVALID_FILEHANDLE THEN
        RAISE_APPLICATION_ERROR(-20001,'Invalid File.');
    WHEN UTL_FILE.WRITE_ERROR THEN
        RAISE_APPLICATION_ERROR (-20002, 'Unable to write to file');
END sal_status;/
```

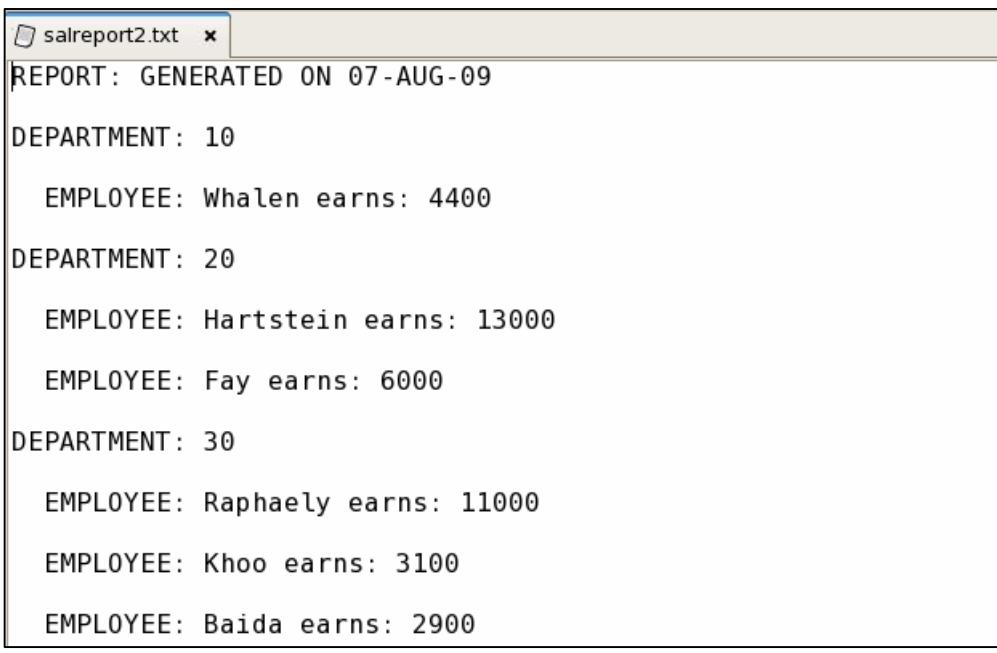
ORACLE®

5 - 15

Copyright © 2009, Oracle. All rights reserved.

Using UTL_FILE: Example (continued)

The following is a sample of the salreport2.txt output file:



The screenshot shows a Windows Notepad window with the title 'salreport2.txt'. The content of the window is as follows:

```
REPORT: GENERATED ON 07-AUG-09

DEPARTMENT: 10

EMPLOYEE: Whalen earns: 4400

DEPARTMENT: 20

EMPLOYEE: Hartstein earns: 13000

EMPLOYEE: Fay earns: 6000

DEPARTMENT: 30

EMPLOYEE: Raphaely earns: 11000

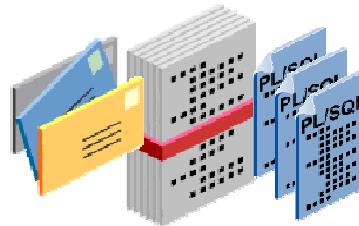
EMPLOYEE: Khoo earns: 3100

EMPLOYEE: Baida earns: 2900

...
```

What Is the UTL_MAIL Package?

- A utility for managing email
- Requires the setting of the SMTP_OUT_SERVER database initialization parameter
- Provides the following procedures:
 - SEND for messages without attachments
 - SEND_ATTACH_RAW for messages with binary attachments
 - SEND_ATTACH_VARCHAR2 for messages with text attachments



ORACLE®

5 - 16

Copyright © 2009, Oracle. All rights reserved.

Using UTL_MAIL

The UTL_MAIL package is a utility for managing email that includes commonly used email features such as attachments, CC, BCC, and return receipt.

The UTL_MAIL package is not installed by default because of the SMTP_OUT_SERVER configuration requirement and the security exposure this involves. When installing UTL_MAIL, you should take steps to prevent the port defined by SMTP_OUT_SERVER being swamped by data transmissions. To install UTL_MAIL, log in as a DBA user in SQL*Plus and execute the following scripts:

```
@$ORACLE_HOME/rdbms/admin/utlmail.sql  
@$ORACLE_HOME/rdbms/admin/prvtmail.plb
```

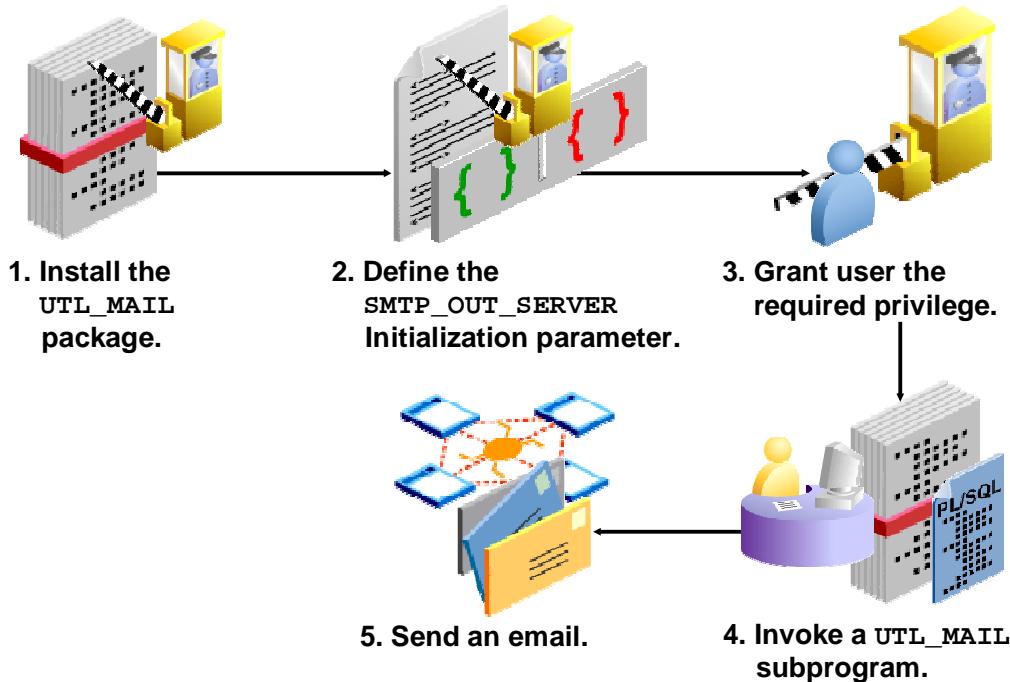
You should define the SMTP_OUT_SERVER parameter in the init.ora file database initialization file:

```
SMTP_OUT_SERVER=mystmpserver.mydomain.com
```

The SMTP_OUT_SERVER parameter specifies the SMTP host and port to which UTL_MAIL delivers outbound email. Multiple servers can be specified, separated by commas. If the first server in the list is unavailable, then UTL_MAIL tries the second server, and so on. If SMTP_OUT_SERVER is not defined, then this invokes a default setting derived from DB_DOMAIN, which is a database initialization parameter specifying the logical location of the database within the network structure. For example:

```
db_domain=mydomain.com
```

Setting Up and Using the UTL_MAIL: Overview



ORACLE®

Setting Up and Using the UTL_MAIL: Overview

In Oracle Database 11g, the UTL_MAIL package is now an invoker's rights package and the invoking user will need the connect privilege granted in the access control list assigned to the remote network host to which he wants to connect. The Security Administrator performs this task.

Note

- For information about how a user with SYSDBA capabilities grants a user the required fine-grained privileges required for using this package, refer to the “Managing Fine-Grained Access to External Network Services” topic in the *Oracle Database Security Guide 11g Release 2 (11.2)* guide and the *Oracle Database 11g Advanced PL/SQL* instructor-led training course.
- Due to firewall restrictions, the UTL_MAIL examples in this lesson cannot be demonstrated; therefore, no labs were designed to use UTL_MAIL.

Summary of UTL_MAIL Subprograms

Subprogram	Description
SEND procedure	Packages an email message, locates SMTP information, and delivers the message to the SMTP server for forwarding to the recipients
SEND_ATTACH_RAW Procedure	Represents the SEND procedure overloaded for RAW attachments
SEND_ATTACH_VARCHAR2 Procedure	Represents the SEND procedure overloaded for VARCHAR2 attachments

ORACLE®

Installing and Using UTL_MAIL

- As SYSDBA, using SQL Developer or SQL*Plus:
 - Install the UTL_MAIL package

```
@?/rdbms/admin/utlmail.sql  
@?/rdbms/admin/prvtmail.plb
```

- Set the SMTP_OUT_SERVER

```
ALTER SYSTEM SET SMTP_OUT_SERVER='smtp.server.com'  
SCOPE=SPFILE
```

- As a developer, invoke a UTL_MAIL procedure:

```
BEGIN  
    UTL_MAIL.SEND('otn@oracle.com', 'user@oracle.com',  
                  message => 'For latest downloads visit OTN',  
                  subject => 'OTN Newsletter');  
END;
```

ORACLE®

5 - 19

Copyright © 2009, Oracle. All rights reserved.

Installing and Using UTL_MAIL

The slide shows how to configure the SMTP_OUT_SERVER parameter to the name of the SMTP host in your network, and how to install the UTL_MAIL package that is not installed by default. Changing the SMTP_OUT_SERVER parameter requires restarting the database instance. These tasks are performed by a user with SYSDBA capabilities.

The last example in the slide shows the simplest way to send a text message by using the UTL_MAIL.SEND procedure with at least a subject and a message. The first two required parameters are the following :

- The sender email address (in this case, otn@oracle.com)
- The recipients email address (for example, user@oracle.com). The value can be a comma-separated list of addresses.

The UTL_MAIL.SEND procedure provides several other parameters, such as cc, bcc, and priority with default values, if not specified. In the example, the message parameter specifies the text for the email, and the subject parameter contains the text for the subject line. To send an HTML message with HTML tags, add the mime_type parameter (for example, mime_type=>'text/html').

Note: For details about all the UTL_MAIL procedure parameters, refer to the *Oracle Database PL/SQL Packages and Types Reference 11g Release 2 (11.2)* guide.

The SEND Procedure Syntax

Packages an email message into the appropriate format, locates SMTP information, and delivers the message to the SMTP server for forwarding to the recipients

```
UTL_MAIL.SEND (
    sender      IN      VARCHAR2 CHARACTER SET ANY_CS,
    recipients   IN      VARCHAR2 CHARACTER SET ANY_CS,
    cc          IN      VARCHAR2 CHARACTER SET ANY_CS
                          DEFAULT NULL,
    bcc         IN      VARCHAR2 CHARACTER SET ANY_CS
                          DEFAULT NULL,
    subject     IN      VARCHAR2 CHARACTER SET ANY_CS
                          DEFAULT NULL,
    message     IN      VARCHAR2 CHARACTER SET ANY_CS,
    mime_type   IN      VARCHAR2
                          DEFAULT 'text/plain; charset=us-ascii',
    priority    IN      PLS_INTEGER DEFAULT NULL);
```

ORACLE®

The SEND Procedure

This procedure packages an email message into the appropriate format, locates SMTP information, and delivers the message to the SMTP server for forwarding to the recipients. It hides the SMTP API and exposes a one-line email facility for ease of use.

The SEND Procedure Parameters

- **sender:** The email address of the sender.
- **recipients:** The email addresses of the recipient(s), separated by commas.
- **cc:** The email addresses of the CC recipient(s), separated by commas. The default is NULL.
- **bcc:** The email addresses of the BCC recipient(s), separated by commas. The default is NULL.
- **subject:** A string to be included as email subject string. The default is NULL.
- **message:** A text message body.
- **mime_type:** The mime type of the message, default is 'text/plain; charset=us-ascii'.
- **priority:** The message priority. The default is NULL.

The SEND_ATTACH_RAW Procedure

This procedure is the SEND procedure overloaded for RAW attachments.

```
UTL_MAIL.SEND_ATTACH_RAW (
    sender          IN      VARCHAR2 CHARACTER SET ANY_CS,
    recipients     IN      VARCHAR2 CHARACTER SET ANY_CS,
    cc              IN      VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    bcc             IN      VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    subject         IN      VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    message         IN      VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    mime_type       IN      VARCHAR2 DEFAULT CHARACTER SET ANY_CS
                            DEFAULT 'text/plain; charset=us-ascii',
    priority        IN      PLS_INTEGER DEFAULT 3,
    attachment      IN      RAW,
    att_inline      IN      BOOLEAN DEFAULT TRUE,
    att_mime_type   IN      VARCHAR2 CHARACTER SET ANY_CS
                            DEFAULT 'text/plain; charset=us-ascii',
    att_filename    IN      VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL);
```

ORACLE®

5 - 21

Copyright © 2009, Oracle. All rights reserved.

The SEND_ATTACH_RAW Procedure Parameters

- **sender:** The email address of the sender
- **recipients:** The email addresses of the recipient(s), separated by commas
- **cc:** The email addresses of the CC recipient(s), separated by commas. The default is NULL.
- **bcc:** The email addresses of the BCC recipient(s), separated by commas. The default is NULL.
- **subject:** A string to be included as email subject string. The default is NULL.
- **message:** A text message body
- **mime_type:** The mime type of the message, default is 'text/plain; charset=us-ascii'
- **priority:** The message priority. The default is NULL.
- **attachment:** A RAW attachment
- **att_inline:** Specifies whether the attachment is viewable inline with the message body. The default is TRUE.
- **att_mime_type:** The mime type of the attachment, default is 'application/octet'
- **att_filename:** The string specifying a file name containing the attachment. The default is NULL.

Sending Email with a Binary Attachment: Example

```
CREATE OR REPLACE PROCEDURE send_mail_logo IS
BEGIN
    UTL_MAIL.SEND_ATTACH_RAW(
        sender => 'me@oracle.com',
        recipients => 'you@somewhere.net',
        message =>
            '<HTML><BODY>See attachment</BODY></HTML>',
        subject => 'Oracle Logo',
        mime_type => 'text/html'
        attachment => get_image('oracle.gif'),
        att_inline => true,
        att_mime_type => 'image/gif',
        att_filename => 'oralogo.gif');
END;
/
```

ORACLE®

5 - 22

Copyright © 2009, Oracle. All rights reserved.

Sending Email with a Binary Attachment: Example

The slide shows a procedure calling the UTL_MAIL.SEND_ATTACH_RAW procedure to send a textual or an HTML message with a binary attachment. In addition to the `sender`, `recipients`, `message`, `subject`, and `mime_type` parameters that provide values for the main part of the email message, the SEND_ATTACH_RAW procedure has the following highlighted parameters:

- The `attachment` parameter (required) accepts a RAW data type, with a maximum size of 32,767 binary characters.
- The `att_inline` parameter (optional) is Boolean (default TRUE) to indicate that the attachment is viewable with the message body.
- The `att_mime_type` parameter (optional) specifies the format of the attachment. If not provided, it is set to `application/octet`.
- The `att_filename` parameter (optional) assigns any file name to the attachment. It is `NULL` by default, in which case, the name is assigned a default name.

The `get_image` function in the example uses a BFILE to read the image data. Using a BFILE requires creating a logical directory name in the database by using the `CREATE DIRECTORY` statement. The code for `get_image` is shown on the following page.

Sending Email with a Binary Attachment: Example (continued)

The `get_image` function uses the DBMS_LOB package to read a binary file from the operating system:

```
CREATE OR REPLACE FUNCTION get_image(
    filename VARCHAR2, dir VARCHAR2 := 'TEMP')
RETURN RAW IS
    image RAW(32767);
    file BFILE := BFILENAME(dir, filename);
BEGIN
    DBMS_LOB.FILEOPEN(file, DBMS_LOB.FILE_READONLY);
    image := DBMS_LOB.SUBSTR(file);
    DBMS_LOB.CLOSE(file);
    RETURN image;
END;
/
```

To create the directory called TEMP, execute the following statement in SQL Developer or SQL*Plus:

```
CREATE DIRECTORY temp AS 'd:\temp';
```

Note

- You need the CREATE ANY DIRECTORY system privilege to execute this statement.
- Due to firewall restrictions at the Oracle Education Center, the examples on this page and the previous page are not available for demonstration.

The SEND_ATTACH_VARCHAR2 Procedure

This procedure is the SEND procedure overloaded for VARCHAR2 attachments.

```
UTL_MAIL.SEND_ATTACH_VARCHAR2 (
    sender          IN  VARCHAR2 CHARACTER SET ANY_CS,
    recipients      IN  VARCHAR2 CHARACTER SET ANY_CS,
    cc              IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    bcc             IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    subject         IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    message         IN  VARCHAR2 CHARACTER SET ANY_CS DEFAULT NULL,
    mime_type       IN  VARCHAR2 CHARACTER SET ANY_CS
                           DEFAULT 'text/plain; charset=us-ascii',
    priority        IN  PLS_INTEGER DEFAULT 3,
    attachment      IN  VARCHAR2 CHARACTER SET ANY_CS,
    att_inline      IN  BOOLEAN DEFAULT TRUE,
    att_mime_type   IN  VARCHAR2 CHARACTER SET ANY_CS
                           DEFAULT 'text/plain; charset=us-ascii',
    att_filename    IN  VARCHAR2CHARACTER SET ANY_CS DEFAULT NULL);
```

ORACLE®

5 - 24

Copyright © 2009, Oracle. All rights reserved.

The SEND_ATTACH_VARCHAR2 Procedure Parameters

- **sender:** The email address of the sender
- **recipients:** The email addresses of the recipient(s), separated by commas
- **cc:** The email addresses of the CC recipient(s), separated by commas. The default is NULL.
- **bcc:** The email addresses of the BCC recipient(s), separated by commas. The default is NULL.
- **subject:** A string to be included as email subject string. The default is NULL.
- **Message:** A text message body
- **mime_type:** The mime type of the message, default is 'text/plain; charset=us-ascii'
- **priority:** The message priority. The default is NULL.
- **attachment:** A text attachment
- **att_inline:** Specifies whether the attachment is inline. The default is TRUE.
- **att_mime_type:** The mime type of the attachment, default is 'text/plain; charset=us-ascii'
- **att_filename:** The string specifying a file name containing the attachment. The default is NULL.

Sending Email with a Text Attachment: Example

```
CREATE OR REPLACE PROCEDURE send_mail_file IS
BEGIN
    UTL_MAIL.SEND_ATTACH_VARCHAR2(
        sender => 'me@oracle.com',
        recipients => 'you@somewhere.net',
        message =>
            '<HTML><BODY>See attachment</BODY></HTML>',
        subject => 'Oracle Notes',
        mime_type => 'text/html'
        attachment => get_file('notes.txt'),
        att_inline => false,
        att_mime_type => 'text/plain',
        att_filename => 'notes.txt');
END;
/
```

ORACLE®

5 - 25

Copyright © 2009, Oracle. All rights reserved.

Sending Email with a Text Attachment

The slide shows a procedure that calls the UTL_MAIL.SEND_ATTACH_VARCHAR2 procedure to send a textual or an HTML message with a text attachment. In addition to the `sender`, `recipients`, `message`, `subject`, and `mime_type` parameters that provide values for the main part of the e-mail message, the `SEND_ATTACH_VARCHAR2` procedure has the following parameters highlighted:

- The `attachment` parameter (required) accepts a `VARCHAR2` data type with a maximum size of 32,767 binary characters.
- The `att_inline` parameter (optional) is a Boolean (default TRUE) to indicate that the attachment is viewable with the message body.
- The `att_mime_type` parameter (optional) specifies the format of the attachment. If not provided, it is set to `application/octet`.
- The `att_filename` parameter (optional) assigns any file name to the attachment. It is `NULL` by default, in which case, the name is assigned a default name.

The `get_file` function in the example uses a `BFILE` to read a text file from the operating system directories for the value of the `attachment` parameter, which could simply be populated from a `VARCHAR2` variable. The code for `get_file` is shown on the following page.

Sending Email with a Text Attachment (continued)

The `get_file` function uses the DBMS_LOB package to read a binary file from the operating system, and uses the UTL_RAW package to convert the RAW binary data into readable text data in the form of a VARCHAR2 data type:

```
CREATE OR REPLACE FUNCTION get_file(
    filename VARCHAR2, dir VARCHAR2 := 'TEMP')
RETURN VARCHAR2 IS
    contents VARCHAR2(32767);
    file BFILE := BFILENAME(dir, filename);
BEGIN
    DBMS_LOB.FILEOPEN(file, DBMS_LOB.FILE_READONLY);
    contents := UTL_RAW.CAST_TO_VARCHAR2(
        DBMS_LOB.SUBSTR(file));
    DBMS_LOB.CLOSE(file);
    RETURN contents;
END;
/
```

Note: Alternatively, you could read the contents of the text file into a VARCHAR2 variable by using the UTL_FILE package functionality.

The preceding example requires the TEMP directory to be created similar to the following statement in SQL*Plus:

```
CREATE DIRECTORY temp AS '/temp';
```

Note

- The CREATE ANY DIRECTORY system privilege is required to execute this statement.
- Due to firewall restrictions at the Oracle Education Center, the examples on this page and the previous page are not available for demonstration.

Quiz

The Oracle-supplied UTL_FILE package is used to access text files in the operating system of the database server.

The database provides functionality through directory objects to allow access to specific operating system directories.

1. True
2. False



Answer: 1

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.
- The paths specified in the utl_file_dir database initialization parameter

Summary

In this lesson, you should have learned:

- How the DBMS_OUTPUT package works
- How to use UTL_FILE to direct output to operating system files
- About the main features of UTL_MAIL



Summary

This lesson covers a small subset of packages provided with the Oracle database. You have extensively used DBMS_OUTPUT for debugging purposes and displaying procedurally generated information on the screen in SQL*Plus.

In this lesson, you should have learned how to use the power features provided by the database to create text files in the operating system by using UTL_FILE. You also learned how to send email with or without binary or text attachments by using the UTL_MAIL package.

Note: For more information about all PL/SQL packages and types, refer to *PL/SQL Packages and Types Reference*.

Practice 5: Overview

This practice covers how to use UTL_FILE to generate a text report.



Practice 5: Overview

In this practice, you use UTL_FILE to generate a text file report of employees in each department.

Using Dynamic SQL

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

Objectives

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

- Describe the execution flow of SQL statements
- Build and execute SQL statements dynamically using Native Dynamic SQL (NDS)
- Identify situations when you must use the DBMS_SQL package instead of NDS to build and execute SQL statements dynamically



Lesson Aim

In this lesson, you learn to construct and execute SQL statements dynamically—that is, at run time using the Native Dynamic SQL statements in PL/SQL.

Lesson Agenda

- Using Native Dynamic SQL (NDS)
- Using the DBMS_SQL package



Execution Flow of SQL

- All SQL statements go through some or all of the following stages:
 - Parse
 - Bind
 - Execute
 - Fetch
- Some stages may not be relevant for all statements:
 - The fetch phase is applicable to queries.
 - For embedded SQL statements such as SELECT, DML, MERGE, COMMIT, SAVEPOINT, and ROLLBACK, the parse and bind phases are done at compile time.
 - For dynamic SQL statements, all phases are performed at run time.

ORACLE®

6 - 4

Copyright © 2009, Oracle. All rights reserved.

Steps to Process SQL Statements

All SQL statements have to go through various stages. However, some stages may not be relevant for all statements. The following are the key stages:

- **Parse:** Every SQL statement must be parsed. Parsing the statement includes checking the statement's syntax and validating the statement, ensuring that all references to objects are correct and that the relevant privileges to those objects exist.
- **Bind:** After parsing, the Oracle server may need values from or for any bind variable in the statement. The process of obtaining these values is called binding variables. This stage may be skipped if the statement does not contain bind variables.
- **Execute:** At this point, the Oracle server has all necessary information and resources, and the statement is executed. For non-query statements, this is the last phase.
- **Fetch:** In the fetch stage, which is applicable to queries, the rows are selected and ordered (if requested by the query), and each successive fetch retrieves another row of the result, until the last row has been fetched.

Working with Dynamic SQL

Use dynamic SQL to create a SQL statement whose structure may change during run time. Dynamic SQL:

- Is constructed and stored as a character string, string variable, or string expression within the application
- Is a SQL statement with varying column data, or different conditions with or without placeholders (bind variables)
- Enables DDL, DCL, or session-control statements to be written and executed from PL/SQL
- Is executed with Native Dynamic SQL statements or the DBMS_SQL package



Dynamic SQL

The embedded SQL statements available in PL/SQL are limited to SELECT, INSERT, UPDATE, DELETE, MERGE, COMMIT, and ROLLBACK, all of which are parsed at compile time—that is, they have a fixed structure. You need to use dynamic SQL functionality if you require:

- The structure of a SQL statement to be altered at run time
- Access to data definition language (DDL) statements and other SQL functionality in PL/SQL

To perform these kinds of tasks in PL/SQL, you must construct SQL statements dynamically in character strings and execute them using either of the following:

- Native Dynamic SQL statements with EXECUTE IMMEDIATE
- The DBMS_SQL package

The process of using SQL statements that are not embedded in your source program and are constructed in strings and executed at run time is known as “dynamic SQL.” The SQL statements are created dynamically at run time and can access and use PL/SQL variables. For example, you create a procedure that uses dynamic SQL to operate on a table whose name is not known until run time, or execute a DDL statement (such as CREATE TABLE), a data control statement (such as GRANT), or a session control statement (such as ALTER SESSION).

Using Dynamic SQL

- Use dynamic SQL when the full text of the dynamic SQL statement is unknown until run time; therefore, its syntax is checked at *run time* rather than at *compile time*.
- Use dynamic SQL when one of the following items is unknown at precompile time:
 - Text of the SQL statement such as commands, clauses, and so on
 - The number and data types of host variables
 - References to database objects such as tables, columns, indexes, sequences, usernames, and views
- Use dynamic SQL to make your PL/SQL programs more general and flexible.

ORACLE®

6 - 6

Copyright © 2009, Oracle. All rights reserved.

Using Dynamic SQL

In PL/SQL, you need dynamic SQL to execute the following SQL statements where the full text is unknown at compile time such as:

- A SELECT statement that includes an identifier that is unknown at compile time (such as a table name)
- A WHERE clause in which the column name is unknown at compile time

Note

For additional information about dynamic SQL, see the following resources:

- *Pro*C/C++ Programmer's Guide 11g Release 2 (11.2)* documentation guide
 - *Lesson 13, Oracle Dynamic SQL*, covers the four available methods that you can use to define dynamic SQL statements. It briefly describes the capabilities and limitations of each method, and then offers guidelines for choosing the right method. Later sections in the same guide show you how to use the methods, and include example programs that you can study.
 - *Lesson 15, Oracle Dynamic SQL: Method 4*, contains very detailed information about Method 4 when defining dynamic SQL statements.
- *Oracle PL/SQL Programming* book by Steven Feuerstein and Bill Pribyl. *Lesson 16, Dynamic SQL and Dynamic PL/SQL*, contains additional information about dynamic SQL.

Native Dynamic SQL (NDS)

- Provides native support for dynamic SQL directly in the PL/SQL language.
- Provides the ability to execute SQL statements whose structure is unknown until execution time.
- If the dynamic SQL statement is a `SELECT` statement that returns multiple rows, NDS gives you the following choices:
 - Use the `EXECUTE IMMEDIATE` statement with the `BULK COLLECT INTO` clause
 - Use the `OPEN-FOR`, `FETCH`, and `CLOSE` statements
- In Oracle Database 11g, NDS supports statements larger than 32 KB by accepting a `CLOB` argument.

ORACLE®

6 - 7

Copyright © 2009, Oracle. All rights reserved.

Native Dynamic SQL

Native Dynamic SQL provides the ability to dynamically execute SQL statements whose structure is constructed at execution time. The following statements have been added or extended in PL/SQL to support Native Dynamic SQL:

- **EXECUTE IMMEDIATE:** Prepares a statement, executes it, returns variables, and then deallocates resources
- **OPEN-FOR:** Prepares and executes a statement using a cursor variable
- **FETCH:** Retrieves the results of an opened statement by using the cursor variable
- **CLOSE:** Closes the cursor used by the cursor variable and deallocates resources

You can use bind variables in the dynamic parameters in the `EXECUTE IMMEDIATE` and `OPEN` statements. Native Dynamic SQL includes the following capabilities:

- Define a dynamic SQL statement.
- Handle `IN`, `IN OUT`, and `OUT` bind variables that are bound by position, not by name.

Using the EXECUTE IMMEDIATE Statement

Use the EXECUTE IMMEDIATE statement for NDS or PL/SQL anonymous blocks:

```
EXECUTE IMMEDIATE dynamic_string
[INTO {define_variable
[, define_variable] ... | record}]
[USING [IN|OUT|IN OUT] bind_argument
[, [IN|OUT|IN OUT] bind_argument] ... ];
```

- INTO is used for single-row queries and specifies the variables or records into which column values are retrieved.
- USING is used to hold all bind arguments. The default parameter mode is IN.

ORACLE®

6 - 8

Copyright © 2009, Oracle. All rights reserved.

Using the EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement can be used to execute SQL statements or PL/SQL anonymous blocks. The syntactical elements include the following:

- dynamic_string is a string expression that represents a dynamic SQL statement (without terminator) or a PL/SQL block (with terminator).
- define_variable is a PL/SQL variable that stores the selected column value.
- record is a user-defined or %ROWTYPE record that stores a selected row.
- bind_argument is an expression whose value is passed to the dynamic SQL statement or PL/SQL block.
- The INTO clause specifies the variables or record into which column values are retrieved. It is used only for single-row queries. For each value retrieved by the query, there must be a corresponding, type-compatible variable or field in the INTO clause.
- The USING clause holds all bind arguments. The default parameter mode is IN.

You can use numeric, character, and string literals as bind arguments, but you cannot use Boolean literals (TRUE, FALSE, and NULL).

Note: Use OPEN-FOR, FETCH, and CLOSE for a multirow query. The syntax shown in the slide is not complete because support exists for bulk-processing operations (which is a topic that is not covered in this course).

Available Methods for Using NDS

Method #	SQL Statement Type	NDS SQL Statements Used
Method 1	Non-query without host variables	EXECUTE IMMEDIATE without the USING and INTO clauses
Method 2	Non-query with known number of input host variables	EXECUTE IMMEDIATE with a USING clause
Method 3	Query with known number of select-list items and input host variables	EXECUTE IMMEDIATE with the USING and INTO clauses
Method 4	Query with unknown number of select-list items or input host variables	Use the DBMS_SQL package

ORACLE®

6 - 9

Copyright © 2009, Oracle. All rights reserved.

Available Methods for Using NDS

The four available methods for NDS that are listed in the slide are increasingly general. That is, Method 2 encompasses Method 1, Method 3 encompasses Methods 1 and 2, and Method 4 encompasses Methods 1, 2, and 3. However, each method is most useful for handling a certain kind of SQL statement, as follows:

Method 1:

This method lets your program accept or build a dynamic SQL statement, and then immediately execute it using the EXECUTE IMMEDIATE command. The SQL statement must not be a query (SELECT statement) and must not contain any placeholders for input host variables. For example, the following host strings qualify:

- DELETE FROM EMPLOYEES WHERE DEPTNO = 20
- GRANT SELECT ON EMPLOYEES TO scott

With Method 1, the SQL statement is parsed every time it is executed.

Note

- Examples of non-queries include data definition language (DDLS) statements, UPDATES, INSERTS, or DELETES.
- The term *select-list item* includes column names and expressions such as SAL * 1.10 and MAX(SAL).

Available Methods for Using NDS (continued)

Method 2:

This method lets your program accept or build a dynamic SQL statement, and then process it using the PREPARE and EXECUTE commands. The SQL statement must not be a query. The number of placeholders for input host variables and the data types of the input host variables must be known at precompile time. For example, the following host strings fall into this category:

- ```
INSERT INTO EMPLOYEES (FIRST_NAME, LAST_NAME, JOB_ID) VALUES
(:emp_first_name, :emp_last_name, :job_id)
```
- ```
DELETE FROM EMPLOYEES WHERE EMPLOYEE_ID = :emp_number
```

With Method 2, the SQL statement is parsed just once, but can be executed many times with different values for the host variables. SQL data definition statements such as CREATE and GRANT are executed when they are prepared.

Method 3:

This method lets your program accept or build a dynamic query, and then process it using the PREPARE command with the DECLARE, OPEN, FETCH, and CLOSE cursor commands. The number of select-list items, the number of placeholders for input host variables, and the data types of the input host variables must be known at precompile time. For example, the following host strings qualify:

- ```
SELECT DEPARTMENT_ID, MIN(SALARY), MAX(SALARY)
FROM EMPLOYEES
GROUP BY DEPARTMENT_ID
```
- ```
SELECT LAST_NAME, EMPLOYEE_ID
FROM EMPLOYEES
WHERE DEPARTMENT_ID = :dept_number
```

Method 4:

This method lets your program accept or build a dynamic SQL statement, and then process it using descriptors. A descriptor is an area of memory used by your program and Oracle to hold a complete description of the variables in a dynamic SQL statement. The number of select-list items, the number of placeholders for input host variables, and the data types of the input host variables can be unknown until run time. For example, the following host strings fall into this category:

- ```
INSERT INTO EMPLOYEES (<unknown>) VALUES (<unknown>)
```
- ```
SELECT <unknown> FROM EMPLOYEES WHERE DEPARTMENT_ID = 20
```

Method 4 is required for dynamic SQL statements that contain an unknown number of select-list items or input host variables. With this method, you use the DBMS_SQL package, which is covered later in this lesson. Situations that require using Method 4 are rare.

Note

For additional information about the four dynamic SQL methods, see the following lessons in the *Pro*C/C++ Programmer's Guide 11g Release 2 (11.2)* documentation guide.

- *Lesson 13, Oracle Dynamic SQL*
- *Lesson 15, Oracle Dynamic SQL: Method 4*

Dynamic SQL with a DDL Statement: Examples

```
-- Create a table using dynamic SQL

CREATE OR REPLACE PROCEDURE create_table(
    p_table_name VARCHAR2, p_col_specs VARCHAR2) IS
BEGIN
    EXECUTE IMMEDIATE 'CREATE TABLE ' || p_table_name ||
        ' (' || p_col_specs || ')';
END;
/
```

```
-- Call the procedure

BEGIN
    create_table('EMPLOYEE_NAMES',
        'id NUMBER(4) PRIMARY KEY, name VARCHAR2(40)');
END;
/
```

ORACLE®

6 - 11

Copyright © 2009, Oracle. All rights reserved.

Dynamic SQL with a DDL Statement

The code examples show the creation of a `create_table` procedure that accepts the table name and column definitions (specifications) as parameters.

The procedure call shows the creation of a table called `EMPLOYEE_NAMES` with two columns:

- An ID column with a `NUMBER` data type used as a primary key
- A name column of up to 40 characters for the employee name

Any DDL statement can be executed by using the syntax shown in the slide, whether the statement is dynamically constructed or specified as a literal string. You can create and execute a statement that is stored in a PL/SQL string variable, as in the following example:

```
CREATE OR REPLACE PROCEDURE add_col(p_table_name VARCHAR2,
                                    p_col_spec    VARCHAR2) IS
    v_stmt VARCHAR2(100) := 'ALTER TABLE ' || p_table_name ||
                           ' ADD ' || p_col_spec;
BEGIN
    EXECUTE IMMEDIATE v_stmt;
END;
/
```

To add a new column to a table, enter the following:

```
EXECUTE add_col('employee_names', 'salary number(8,2)')
```

Dynamic SQL with DML Statements

```
-- Delete rows from any table:  
CREATE FUNCTION del_rows(p_table_name VARCHAR2)  
RETURN NUMBER IS  
BEGIN  
    EXECUTE IMMEDIATE 'DELETE FROM '|| p_table_name;  
    RETURN SQL%ROWCOUNT;  
END;  
/  
BEGIN DBMS_OUTPUT.PUT_LINE(  
    del_rows('EMPLOYEE_NAMES')|| ' rows deleted.');//  
END;  
/
```

```
-- Insert a row into a table with two columns:  
CREATE PROCEDURE add_row(p_table_name VARCHAR2,  
    p_id NUMBER, p_name VARCHAR2) IS  
BEGIN  
    EXECUTE IMMEDIATE 'INSERT INTO '|| p_table_name ||  
        ' VALUES (:1, :2)' USING p_id, p_name;  
END;
```

ORACLE®

6 - 12

Copyright © 2009, Oracle. All rights reserved.

Dynamic SQL with DML Statements

The first code example in the slide defines a dynamic SQL statement using Method 1—that is, nonquery without host variables. The examples in the slide demonstrate the following:

- The `del_rows` function deletes rows from a specified table and returns the number of rows deleted by using the implicit SQL cursor `%ROWCOUNT` attribute. Executing the function is shown below the example for creating a function.
- The `add_row` procedure shows how to provide input values to a dynamic SQL statement with the `USING` clause. The bind variable names `:1` and `:2` are not important; however, the order of the parameter names (`p_id` and `p_name`) in the `USING` clause is associated with the bind variables by position, in the order of their respective appearance. Therefore, the PL/SQL parameter `p_id` is assigned to the `:1` placeholder, and the `p_name` parameter is assigned to the `:2` placeholder. Placeholder or bind variable names can be alphanumeric but must be preceded with a colon.

Note: The `EXECUTE IMMEDIATE` statement prepares (parses) and immediately executes the dynamic SQL statement. Dynamic SQL statements are always parsed.

Also, note that a `COMMIT` operation is not performed in either of the examples. Therefore, the operations can be undone with a `ROLLBACK` statement.

Dynamic SQL with a Single-Row Query: Example

```
CREATE FUNCTION get_emp( p_emp_id NUMBER )
  RETURN employees%ROWTYPE IS
    v_stmt VARCHAR2(200);
    v_emprec employees%ROWTYPE;
BEGIN
  v_stmt := 'SELECT * FROM employees ' ||
            'WHERE employee_id = :p_emp_id';
  EXECUTE IMMEDIATE v_stmt INTO v_emprec USING p_emp_id;
  RETURN v_emprec;
END;
/
DECLARE
  v_emprec employees%ROWTYPE := get_emp(100);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Emp: ' || v_emprec.last_name);
END;
/
```

```
FUNCTION get_emp(p_emp_id Compiled.
anonymous block completed
Emp: King
```

ORACLE®

6 - 13

Copyright © 2009, Oracle. All rights reserved.

Dynamic SQL with a Single-Row Query

The code example in the slide is an example of defining a dynamic SQL statement using Method 3 with a single row queried—that is, query with a known number of select-list items and input host variables.

The single-row query example demonstrates the `get_emp` function that retrieves an `EMPLOYEES` record into a variable specified in the `INTO` clause. It also shows how to provide input values for the `WHERE` clause.

The anonymous block is used to execute the `get_emp` function and return the result into a local `EMPLOYEES` record variable.

The example could be enhanced to provide alternative `WHERE` clauses depending on input parameter values, making it more suitable for dynamic SQL processing.

Note:

- For an example of “Dynamic SQL with a Multirow Query: Example” using `REF CURSORS`, see the `demo_06_13_a` in the `/home/oracle/labs/plpu/demo` folder.
- For an example on using `REF CURSORS`, see the `demo_06_13_b` in the `/home/oracle/labs/plpu/demo` folder.
- `REF CURSORS` are covered in the *Oracle Database 11g: Advanced PL/SQL* 3-day class.

Executing a PL/SQL Anonymous Block Dynamically

```
CREATE FUNCTION annual_sal( p_emp_id NUMBER )
RETURN NUMBER IS
    v_plsql varchar2(200) := 
        'DECLARE ' ||
        '    rec_emp employees%ROWTYPE; ' ||
        'BEGIN ' ||
        '    rec_emp := get_emp(:empid); ' ||
        '    :res := rec_emp.salary * 12; ' ||
        'END;';
    v_result NUMBER;
BEGIN
    EXECUTE IMMEDIATE v_plsql
        USING IN p_emp_id, OUT v_result;
    RETURN v_result;
END;
/
EXECUTE DBMS_OUTPUT.PUT_LINE(annual_sal(100))
```

6 - 14

Copyright © 2009, Oracle. All rights reserved.

ORACLE®

Dynamically Executing a PL/SQL Block

The annual_sal function dynamically constructs an anonymous PL/SQL block. The PL/SQL block contains bind variables for:

- The input of the employee ID using the :empid placeholder
- The output result computing the annual employees' salary using the placeholder called :res

Note: This example demonstrates how to use the OUT result syntax (in the USING clause of the EXECUTE IMMEDIATE statement) to obtain the result calculated by the PL/SQL block. The procedure output variables and function return values can be obtained in a similar way from a dynamically executed anonymous PL/SQL block.

The output of the slide example is as follows:

```
FUNCTION annual_sal(emp_id Compiled.
anonymous block completed
316800
```

Using Native Dynamic SQL to Compile PL/SQL Code

Compile PL/SQL code with the ALTER statement:

- ALTER PROCEDURE name COMPILE
- ALTER FUNCTION name COMPILE
- ALTER PACKAGE name COMPILE SPECIFICATION
- ALTER PACKAGE name COMPILE BODY

```
CREATE PROCEDURE compile_plsql(p_name VARCHAR2,
                                p_plsql_type VARCHAR2, p_options VARCHAR2 := NULL) IS
    v_stmt varchar2(200) := 'ALTER ' || p_plsql_type || 
                           ' ' || p_name || ' COMPILE';
BEGIN
    IF p_options IS NOT NULL THEN
        v_stmt := v_stmt || ' ' || p_options;
    END IF;
    EXECUTE IMMEDIATE v_stmt;
END;
/
```

ORACLE®

6 - 15

Copyright © 2009, Oracle. All rights reserved.

Using Native Dynamic SQL to Compile PL/SQL Code

The `compile_plsql` procedure in the example can be used to compile different PL/SQL code using the ALTER DDL statement. Four basic forms of the ALTER statement are shown to compile:

- A procedure
- A function
- A package specification
- A package body

Note: If you leave out the keyword SPECIFICATION or BODY with the ALTER PACKAGE statement, then the specification and body are both compiled.

Here are examples of calling the procedure in the slide for each of the four cases, respectively:

```
EXEC compile_plsql ('list_employees', 'procedure')
EXEC compile_plsql ('get_emp', 'function')
EXEC compile_plsql ('mypack', 'package', 'specification')
EXEC compile_plsql ('mypack', 'package', 'body')
```

Here is an example of compiling with debug enabled for the `get_emp` function:

```
EXEC compile_plsql ('get_emp', 'function', 'debug')
```

Lesson Agenda

- Using Native Dynamic SQL (NDS)
- Using the DBMS_SQL package



Using the DBMS_SQL Package

- The DBMS_SQL package is used to write dynamic SQL in stored procedures and to parse DDL statements.
- You must use the DBMS_SQL package to execute a dynamic SQL statement that has an unknown number of input or output variables, also known as Method 4.
- In most cases, NDS is easier to use and performs better than DBMS_SQL except when dealing with Method 4.
- For example, you must use the DBMS_SQL package in the following situations:
 - You do not know the SELECT list at compile time
 - You do not know how many columns a SELECT statement will return, or what their data types will be

ORACLE®

6 - 17

Copyright © 2009, Oracle. All rights reserved.

Using the DBMS_SQL Package

Using DBMS_SQL, you can write stored procedures and anonymous PL/SQL blocks that use dynamic SQL, such as executing DDL statements in PL/SQL—for example, executing a DROP TABLE statement. The operations provided by this package are performed under the current user, not under the package owner SYS.

Method 4: Method 4 refers to situations where, in a dynamic SQL statement, the number of columns selected for a query or the number of bind variables set is not known until run time. In this case, you should use the DBMS_SQL package.

When generating dynamic SQL, you can either use the DBMS_SQL supplied package when dealing with Method 4 situations, or you can use native dynamic SQL. Before Oracle Database 11g, each of these methods had functional limitations. In Oracle Database 11g, functionality is added to both methods to make them more complete.

The features for executing dynamic SQL from PL/SQL had some restrictions in Oracle Database 10g. DBMS_SQL was needed for Method 4 scenarios but it could not handle the full range of data types and its cursor representation was not usable by a client to the database. Native dynamic SQL was more convenient for non-Method 4 scenarios, but it did not support statements bigger than 32 KB. Oracle Database 11g removes these and other restrictions to make the support of dynamic SQL from PL/SQL functionally complete.

Using the DBMS_SQL Package Subprograms

Examples of the package procedures and functions:

- OPEN_CURSOR
- PARSE
- BIND_VARIABLE
- EXECUTE
- FETCH_ROWS
- CLOSE_CURSOR



Using the DBMS_SQL Package Subprograms

The DBMS_SQL package provides the following subprograms to execute dynamic SQL:

- OPEN_CURSOR to open a new cursor and return a cursor ID number
- PARSE to parse the SQL statement. Every SQL statement must be parsed by calling the PARSE procedures. Parsing the statement checks the statement's syntax and associates it with the cursor in your program. You can parse any DML or DDL statement. DDL statements are immediately executed when parsed.
- BIND_VARIABLE to bind a given value to a bind variable identified by its name in the statement being parsed. This is not needed if the statement does not have bind variables.
- EXECUTE to execute the SQL statement and return the number of rows processed
- FETCH_ROWS to retrieve the next row for a query (use in a loop for multiple rows)
- CLOSE_CURSOR to close the specified cursor

Note: Using the DBMS_SQL package to execute DDL statements can result in a deadlock. For example, the most likely reason is that the package is being used to drop a procedure that you are still using.

Using the DBMS_SQL Package Subprograms (continued)

The PARSE Procedure Parameters

The LANGUAGE_FLAG parameter of the PARSE procedure determines how Oracle handles the SQL statement—that is, using behavior associated with a specific Oracle database version. Using NATIVE (or 1) for this parameter specifies using the normal behavior associated with the database to which the program is connected.

If the LANGUAGE_FLAG parameter is set to V6 (or 0), that specifies version 6 behavior. If the LANGUAGE_FLAG parameter is set to V7 (or 2), that specifies Oracle database version 7 behavior.

Note: For additional information, see the *Oracle Database PL/SQL Packages and Types Reference 11g Release 2 (11.2)* guide.

Using DBMS_SQL with a DML Statement: Deleting Rows

```
CREATE OR REPLACE FUNCTION delete_all_rows
  (p_table_name  VARCHAR2) RETURN NUMBER IS
    v_cur_id      INTEGER;
    v_rows_del    NUMBER;
BEGIN
  v_cur_id := DBMS_SQL.OPEN_CURSOR;
  DBMS_SQLPARSE(v_cur_id,
    'DELETE FROM ' || p_table_name, DBMS_SQL.NATIVE);
  v_rows_del := DBMS_SQL.EXECUTE (v_cur_id);
  DBMS_SQL.CLOSE_CURSOR(v_cur_id);
  RETURN v_rows_del;
END;
/
```

```
CREATE TABLE temp_emp AS SELECT * FROM employees;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Rows Deleted: ' ||
delete_all_rows('temp_emp'));
END;
/
```

ORACLE®

6 - 20

Copyright © 2009, Oracle. All rights reserved.

Using DBMS_SQL with a DML Statement

In the slide, the table name is passed into the `delete_all_rows` function. The function uses dynamic SQL to delete rows from the specified table, and returns a count representing the number of rows that are deleted after successful execution of the statement.

To process a DML statement dynamically, perform the following steps:

1. Use `OPEN_CURSOR` to establish an area in memory to process a SQL statement.
2. Use `PARSE` to establish the validity of the SQL statement.
3. Use the `EXECUTE` function to run the SQL statement. This function returns the number of rows processed.
4. Use `CLOSE_CURSOR` to close the cursor.

The steps to execute a DDL statement are similar; but step 3 is optional because a DDL statement is immediately executed when the `PARSE` is successfully done—that is, the statement syntax and semantics are correct. If you use the `EXECUTE` function with a DDL statement, then it does not do anything and returns a value of 0 for the number of rows processed because DDL statements do not process rows.

```
CREATE TABLE succeeded.
anonymous block completed
Rows Deleted: 107

DROP TABLE temp_emp succeeded.
```

Using DBMS_SQL with a Parameterized DML Statement

```
CREATE PROCEDURE insert_row (p_table_name VARCHAR2,
    p_id VARCHAR2, p_name VARCHAR2, p_region NUMBER) IS
    v_cur_id      INTEGER;
    v_stmt        VARCHAR2(200);
    v_rows_added  NUMBER;
BEGIN
    v_stmt := 'INSERT INTO '|| p_table_name ||
              ' VALUES (:cid, :cname, :rid)';
    v_cur_id := DBMS_SQL.OPEN_CURSOR;
    DBMS_SQLPARSE(v_cur_id, v_stmt, DBMS_SQL.NATIVE);
    DBMS_SQL.BIND_VARIABLE(v_cur_id, ':cid', p_id);
    DBMS_SQL.BIND_VARIABLE(v_cur_id, ':cname', p_name);
    DBMS_SQL.BIND_VARIABLE(v_cur_id, ':rid', p_region);
    v_rows_added := DBMS_SQL.EXECUTE(v_cur_id);
    DBMS_SQL.CLOSE_CURSOR(v_cur_id);
    DBMS_OUTPUT.PUT_LINE(v_rows_added|| ' row added');
END;
/
```



Using DBMS_SQL with a Parameterized DML Statement

The example in the slide performs the DML operation to insert a row into a specified table. The example demonstrates the extra step required to associate values to bind variables that exist in the SQL statement. For example, a call to the procedure shown in the slide is:

```
EXECUTE insert_row('countries', 'LB', 'Lebanon', 4)
```

After the statement is parsed, you must call the DBMS_SQL.BIND_VARIABLE procedure to assign values for each bind variable that exists in the statement. The binding of values must be done before executing the code. To process a SELECT statement dynamically, perform the following steps after opening and before closing the cursor:

1. Execute DBMS_SQL.DEFINE_COLUMN for each column selected.
2. Execute DBMS_SQL.BIND_VARIABLE for each bind variable in the query.
3. For each row, perform the following steps:
 - a. Execute DBMS_SQL.FETCH_ROWS to retrieve a row and return the number of rows fetched. Stop additional processing when a zero value is returned.
 - b. Execute DBMS_SQL.COLUMN_VALUE to retrieve each selected column value into each PL/SQL variable for processing.

Although this coding process is not complex, it is more time consuming to write and is prone to error compared with using the Native Dynamic SQL approach.

Quiz

The full text of the dynamic SQL statement might be unknown until run time; therefore, its syntax is checked at *run time* rather than at *compile time*.

1. True
2. False



Answer: 1

Summary

In this lesson, you should have learned how to:

- Describe the execution flow of SQL statements
- Build and execute SQL statements dynamically using Native Dynamic SQL (NDS)
- Identify situations when you must use the DBMS_SQL package instead of NDS to build and execute SQL statements dynamically



Summary

In this lesson, you discovered how to dynamically create any SQL statement and execute it using the Native Dynamic SQL statements. Dynamically executing SQL and PL/SQL code extends the capabilities of PL/SQL beyond query and transactional operations. For earlier releases of the database, you could achieve similar results with the DBMS_SQL package.

Practice 6 Overview: Using Native Dynamic SQL

This practice covers the following topics:

- Creating a package that uses Native Dynamic SQL to create or drop a table and to populate, modify, and delete rows from a table
- Creating a package that compiles the PL/SQL code in your schema



Practice 6: Overview

In this practice, you write code to perform the following tasks:

- Create a package that uses Native Dynamic SQL to create or drop a table, and to populate, modify, and delete rows from the table.
- Create a package that compiles the PL/SQL code in your schema, either all the PL/SQL code or only code that has an INVALID status in the USER_OBJECTS table.



Design Considerations for PL/SQL Code

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

Objectives

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

- Create standard constants and exceptions
- Write and call local subprograms
- Control the run-time privileges of a subprogram
- Perform autonomous transactions
- Pass parameters by reference using a NOCOPY hint
- Use the PARALLEL ENABLE hint for optimization
- Use the cross-session PL/SQL function result cache
- Use the DETERMINISTIC clause with functions
- Use the RETURNING clause and bulk binding with DML



Lesson Aim

In this lesson, you learn to use package specifications to standardize names for constant values and exceptions. You learn how to create subprograms in the Declaration section of any PL/SQL block for using locally in the block. The AUTHID compiler directive is discussed to show how you can manage run-time privileges of the PL/SQL code, and create independent transactions by using the AUTONOMOUS TRANSACTION directive for subprograms.

This lesson also covers some performance considerations that can be applied to PL/SQL applications, such as bulk binding operations with a single SQL statement, the RETURNING clause, and the NOCOPY and PARALLEL ENABLE hints.

Lesson Agenda

- Standardizing constants and exceptions, using local subprograms, controlling the run-time privileges of a subprogram, and performing autonomous transactions
- Using the NOCOPY and the PARALLEL_ENABLE hints, the cross-session PL/SQL function result cache, and the DETERMINISTIC clause
- Using the RETURNING clause and bulk binding with DML

ORACLE®

Standardizing Constants and Exceptions

Constants and exceptions are typically implemented using a bodiless package (that is, a package specification).

- Standardizing helps to:
 - Develop programs that are consistent
 - Promote a higher degree of code reuse
 - Ease code maintenance
 - Implement company standards across entire applications
- Start with standardization of:
 - Exception names
 - Constant definitions

ORACLE®

7 - 4

Copyright © 2009, Oracle. All rights reserved.

Standardizing Constants and Exceptions

When several developers are writing their own exception handlers in an application, there could be inconsistencies in the handling of error situations. Unless certain standards are adhered to, the situation can become confusing because of the different approaches followed in handling the same error or because of the display of conflicting error messages that confuse users. To overcome these, you can:

- Implement company standards that use a consistent approach to error handling across the entire application
- Create predefined, generic exception handlers that produce consistency in the application
- Write and call programs that produce consistent error messages

All good programming environments promote naming and coding standards. In PL/SQL, a good place to start implementing naming and coding standards is with commonly used constants and exceptions that occur in the application domain.

The PL/SQL package specification construct is an excellent component to support standardization because all identifiers declared in the package specification are public. They are visible to the subprograms that are developed by the owner of the package and all code with EXECUTE rights to the package specification.

Standardizing Exceptions

Create a standardized error-handling package that includes all named and programmer-defined exceptions to be used in the application.

```
CREATE OR REPLACE PACKAGE error_pkg IS
    e_fk_err      EXCEPTION;
    e_seq_nbr_err EXCEPTION;
    PRAGMA EXCEPTION_INIT (e_fk_err, -2292);
    PRAGMA EXCEPTION_INIT (e_seq_nbr_err, -2277);
    ...
END error_pkg;
/
```

ORACLE®

7 - 5

Copyright © 2009, Oracle. All rights reserved.

Standardizing Exceptions

In the example in the slide, the `error_pkg` package is a standardized exception package. It declares a set of programmer-defined exception identifiers. Because many of the Oracle database predefined exceptions do not have identifying names, the example package shown in the slide uses the `PRAGMA EXCEPTION_INIT` directive to associate selected exception names with an Oracle database error number. This enables you to refer to any of the exceptions in a standard way in your applications, as in the following example:

```
BEGIN
    DELETE FROM departments
    WHERE department_id = deptno;
    ...
EXCEPTION
    WHEN error_pkg.e_fk_err THEN
        ...
    WHEN OTHERS THEN
        ...
END;
/
```

Standardizing Exception Handling

Consider writing a subprogram for common exception handling to:

- Display errors based on SQLCODE and SQLERRM values for exceptions
- Track run-time errors easily by using parameters in your code to identify:
 - The procedure in which the error occurred
 - The location (line number) of the error
 - RAISE_APPLICATION_ERROR using stack trace capabilities, with the third argument set to TRUE

ORACLE®

7 - 6

Copyright © 2009, Oracle. All rights reserved.

Standardizing Exception Handling

Standardized exception handling can be implemented either as a stand-alone subprogram or a subprogram added to the package that defines the standard exceptions. Consider creating a package with:

- Every named exception that is to be used in the application
- All unnamed, programmer-defined exceptions that are used in the application. These are error numbers –20000 through –20999.
- A program to call RAISE_APPLICATION_ERROR based on package exceptions
- A program to display an error based on the values of SQLCODE and SQLERRM
- Additional objects, such as error log tables, and programs to access the tables

A common practice is to use parameters that identify the name of the procedure and the location in which the error has occurred. This enables you to keep track of run-time errors more easily. An alternative is to use the RAISE_APPLICATION_ERROR built-in procedure to keep a stack trace of exceptions that can be used to track the call sequence leading to the error. To do this, set the third optional argument to TRUE. For example:

```
RAISE_APPLICATION_ERROR(-20001, 'My first error', TRUE);
```

This is meaningful when more than one exception is raised in this manner.

Standardizing Constants

For programs that use local variables whose values should not change:

- Convert the variables to constants to reduce maintenance and debugging
- Create one central package specification and place all constants in it

```
CREATE OR REPLACE PACKAGE constant_pkg IS  
    c_order_received CONSTANT VARCHAR(2) := 'OR';  
    c_order_shipped   CONSTANT VARCHAR(2) := 'OS';  
    c_min_sal         CONSTANT NUMBER(3)   := 900;  
END constant_pkg;
```

ORACLE®

7 - 7

Copyright © 2009, Oracle. All rights reserved.

Standardizing Constants

By definition, a variable's value changes, whereas a constant's value cannot be changed. If you have programs that use local variables whose values should not change, then convert the variables to constants. This can help with the maintenance and debugging of your code.

Consider creating a single shared package with all your constants in it. This makes maintenance and change of the constants much easier. This procedure or package can be loaded on system startup for better performance.

The example in the slide shows the `constant_pkg` package containing a few constants. Refer to any of the package constants in your application as required. Here is an example:

```
BEGIN  
    UPDATE employees  
        SET salary = salary + 200  
        WHERE salary <= constant_pkg.c_min_sal;  
END;  
/
```

Local Subprograms

A local subprogram is a PROCEDURE or FUNCTION defined at the end of the declarative section in a subprogram.

```
CREATE PROCEDURE employee_sal(p_id NUMBER) IS
    v_emp employees%ROWTYPE;
    FUNCTION tax(p_salary VARCHAR2) RETURN NUMBER IS
    BEGIN
        RETURN p_salary * 0.825;
    END tax;
BEGIN
    SELECT * INTO v_emp
    FROM EMPLOYEES WHERE employee_id = p_id;
    DBMS_OUTPUT.PUT_LINE('Tax: ' || tax(v_emp.salary));
END;
/
EXECUTE employee_sal(100)
```

```
PROCEDURE employee_sal(p_id Compiled.
anonymous block completed
Tax: 19800
```

ORACLE®

Local Subprograms

Local subprograms can drive top-down design. They reduce the size of a module by removing redundant code. This is one of the main reasons for creating a local subprogram. If a module needs the same routine several times, but only this module needs the routine, then define it as a local subprogram.

You can define a named PL/SQL block in the declarative section of any PL/SQL program, procedure, function, or anonymous block *provided that it is declared at the end of the Declaration section*. Local subprograms have the following characteristics:

- They are only accessible to the block in which they are defined.
- They are compiled as part of their enclosing blocks.

The benefits of local subprograms are:

- Reduction of repetitive code
- Improved code readability and ease of maintenance
- Less administration because there is one program to maintain instead of two

The concept is simple. The example shown in the slide illustrates this with a basic example of an income tax calculation of an employee's salary.

Definer's Rights Versus Invoker's Rights

Definer's rights:

- Programs execute with the privileges of the creating user.
- User does not require privileges on underlying objects that the procedure accesses. User requires privilege only to execute a procedure.

Invoker's rights:

- Programs execute with the privileges of the calling user.
- User requires privileges on the underlying objects that the procedure accesses.

ORACLE®

7 - 9

Copyright © 2009, Oracle. All rights reserved.

Definer's Rights Versus Invoker's Rights

Definer's Rights Model

By default, all programs executed with the privileges of the user who created the subprogram. This is known as the definer's rights model, which:

- Allows a caller of the program the privilege to execute the procedure, but no privileges on the underlying objects that the procedure accesses
- Requires the owner to have all the necessary object privileges for the objects that the procedure references

For example, if user Scott creates a PL/SQL subprogram `get_employees` that is subsequently invoked by Sarah, then the `get_employees` procedure runs with the privileges of the definer Scott.

Invoker's Rights Model

In the invoker's rights model, which was introduced in Oracle8i, programs are executed with the privileges of the calling user. A user of a procedure running with invoker's rights requires privileges on the underlying objects that the procedure references.

For example, if Scott's PL/SQL subprogram `get_employees` is invoked by Sarah, then the `get_employees` procedure runs with the privileges of the invoker Sarah.

Specifying Invoker's Rights: Setting AUTHID to CURRENT_USER

```
CREATE OR REPLACE PROCEDURE add_dept(
    p_id NUMBER, p_name VARCHAR2) AUTHID CURRENT_USER IS
BEGIN
    INSERT INTO departments
    VALUES (p_id, p_name, NULL, NULL);
END;
```

When used with stand-alone functions, procedures, or packages:

- Names used in queries, DML, Native Dynamic SQL, and DBMS_SQL package are resolved in the invoker's schema
- Calls to other packages, functions, and procedures are resolved in the definer's schema

ORACLE®

7 - 10

Copyright © 2009, Oracle. All rights reserved.

Specifying Invoker's Rights

You can set the invoker's rights for different PL/SQL subprogram constructs as follows:

```
CREATE FUNCTION name RETURN type AUTHID CURRENT_USER IS...
CREATE PROCEDURE name AUTHID CURRENT_USER IS...
CREATE PACKAGE name AUTHID CURRENT_USER IS...
CREATE TYPE name AUTHID CURRENT_USER IS OBJECT...
```

The default is AUTHID DEFINER, which specifies that the subprogram executes with the privileges of its owner. Most supplied PL/SQL packages such as DBMS_LOB, DBMS_ROWID, and so on, are invoker-rights packages.

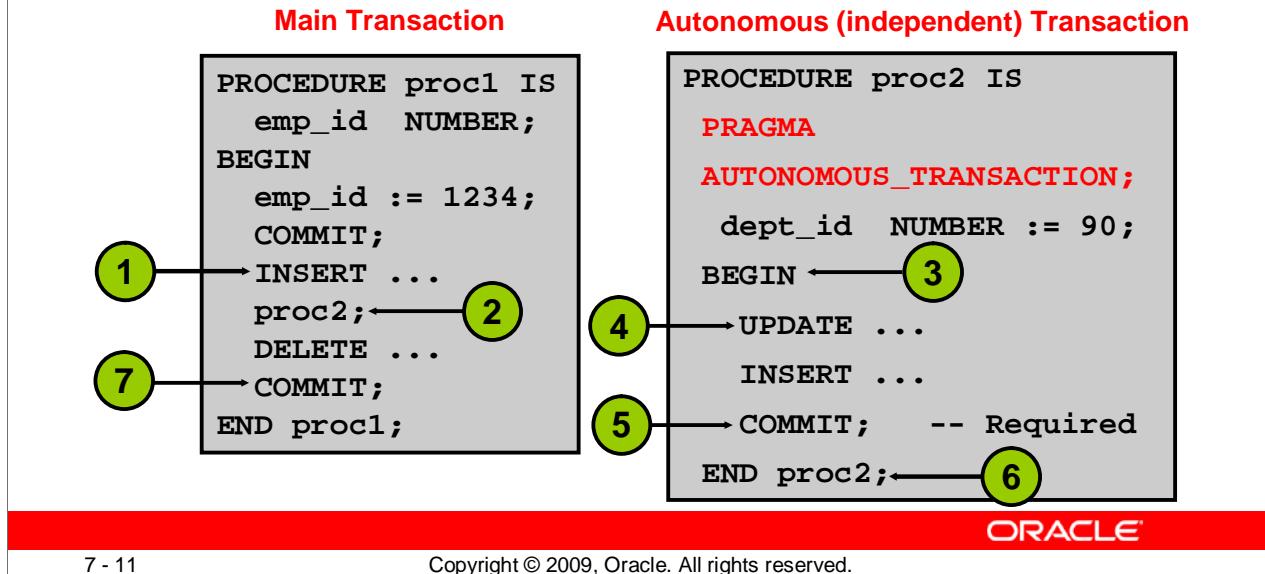
Name Resolution

For a definer's rights procedure, all external references are resolved in the definer's schema. For an invoker's rights procedure, the resolution of external references depends on the kind of statement in which they appear:

- Names used in queries, data manipulation language (DML) statements, dynamic SQL, and DBMS_SQL are resolved in the invoker's schema.
- All other statements, such as calls to packages, functions, and procedures, are resolved in the definer's schema.

Autonomous Transactions

- Are independent transactions started by another main transaction
- Are specified with PRAGMA AUTONOMOUS_TRANSACTION



7 - 11

Copyright © 2009, Oracle. All rights reserved.

ORACLE®

Autonomous Transactions

A transaction is a series of statements doing a logical unit of work that completes or fails as an integrated unit. Often, one transaction starts another that may need to operate outside the scope of the transaction that started it. That is, in an existing transaction, a required independent transaction may need to commit or roll back changes without affecting the outcome of the starting transaction. For example, in a stock purchase transaction, the customer's information must be committed regardless of whether the overall stock purchase completes. Or, while running that same transaction, you want to log messages to a table even if the overall transaction rolls back.

Since Oracle8i, autonomous transactions were added to make it possible to create an independent transaction. An autonomous transaction (AT) is an independent transaction started by another main transaction (MT). The slide depicts the behavior of an AT:

1. The main transaction begins.
2. A proc2 procedure is called to start the autonomous transaction.
3. The main transaction is suspended.
4. The autonomous transactional operation begins.
5. The autonomous transaction ends with a commit or roll back operation.
6. The main transaction is resumed.
7. The main transaction ends.

Features of Autonomous Transactions

- Are independent of the main transaction
- Suspend the calling transaction until the autonomous transactions are completed
- Are not nested transactions
- Do not roll back if the main transaction rolls back
- Enable the changes to become visible to other transactions upon a commit
- Are started and ended by individual subprograms and not by nested or anonymous PL/SQL blocks

ORACLE®

7 - 12

Copyright © 2009, Oracle. All rights reserved.

Features of Autonomous Transactions

Autonomous transactions exhibit the following features:

- Although called within a transaction, autonomous transactions are independent of that transaction. That is, they are not nested transactions.
- If the main transaction rolls back, autonomous transactions do not.
- Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits.
- With their stack-like functionality, only the “top” transaction is accessible at any given time. After completion, the autonomous transaction is popped, and the calling transaction is resumed.
- There are no limits other than resource limits on how many autonomous transactions can be recursively called.
- Autonomous transactions must be explicitly committed or rolled back; otherwise, an error is returned when attempting to return from the autonomous block.
- You cannot use PRAGMA to mark all subprograms in a package as autonomous. Only individual routines can be marked autonomous.
- You cannot mark a nested or anonymous PL/SQL block as autonomous.

Using Autonomous Transactions: Example

```
CREATE TABLE usage (card_id NUMBER, loc NUMBER)
/
CREATE TABLE txn (acc_id NUMBER, amount NUMBER)
/
CREATE OR REPLACE PROCEDURE log_usage (p_card_id NUMBER, p_loc NUMBER)
IS
    PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
    INSERT INTO usage
    VALUES (p_card_id, p_loc);
    COMMIT;
END log_usage;
/
CREATE OR REPLACE PROCEDURE bank_trans(p_cardnbr NUMBER,p_loc NUMBER) IS
BEGIN
    INSERT INTO txn VALUES (9001, 1000);
    log_usage (p_cardnbr, p_loc);
END bank_trans;
/
EXECUTE bank_trans(50, 2000)
```

ORACLE®

7 - 13

Copyright © 2009, Oracle. All rights reserved.

Using Autonomous Transactions

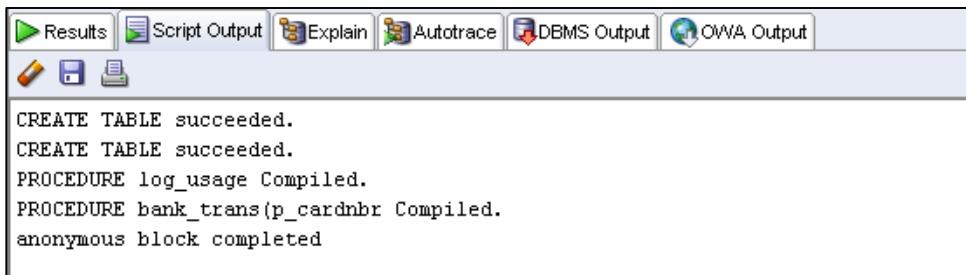
To define autonomous transactions, you use PRAGMA AUTONOMOUS_TRANSACTION. PRAGMA instructs the PL/SQL compiler to mark a routine as autonomous (independent). In this context, the term “routine” includes top-level (not nested) anonymous PL/SQL blocks; local, stand-alone, and packaged functions and procedures; methods of a SQL object type; and database triggers. You can code PRAGMA anywhere in the declarative section of a routine. However, for readability, it is best placed at the top of the Declaration section.

In the example in the slide, you track where the bankcard is used, regardless of whether the transaction is successful. The following are the benefits of autonomous transactions:

- After starting, an autonomous transaction is fully independent. It shares no locks, resources, or commit dependencies with the main transaction, so you can log events, increment retry counters, and so on even if the main transaction rolls back.
- More importantly, autonomous transactions help you build modular, reusable software components. For example, stored procedures can start and finish autonomous transactions on their own. A calling application need not know about a procedure’s autonomous operations, and the procedure need not know about the application’s transaction context. That makes autonomous transactions less error-prone than regular transactions and easier to use.

Using Autonomous Transactions (continued)

The output of the previous slide examples, the TXN and USAGE tables are as follows:



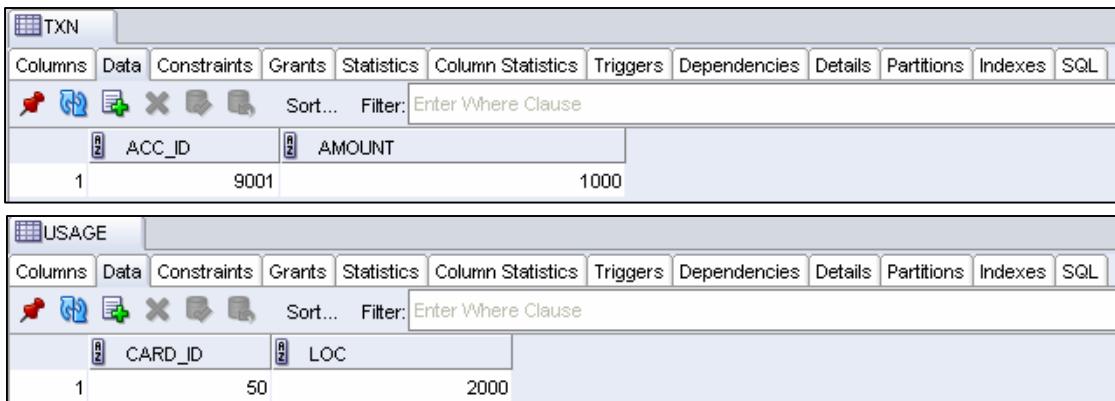
Results | Script Output | Explain | Autotrace | DBMS Output | OWA Output

CREATE TABLE succeeded.
CREATE TABLE succeeded.
PROCEDURE log_usage Compiled.
PROCEDURE bank_trans(p_cardnbr Compiled.
anonymous block completed

To run the code on the previous page, enter the following code:

```
EXECUTE bank_trans(50, 2000)
```

Use the Data tab in the Tables node of the Object Navigator tree to display the values in the TXN and USAGE tables as follows:



TXN

	ACC_ID	AMOUNT
1	9001	1000

USAGE

	CARD_ID	LOC
1	50	2000

Lesson Agenda

- Standardizing constants and exceptions, using local subprograms, controlling the run-time privileges of a subprogram, and performing autonomous transactions
- Using the NOCOPY and the PARALLEL_ENABLE hints, the cross-session PL/SQL function result cache, and the DETERMINISTIC clause
- Using the RETURNING clause and bulk binding with DML

ORACLE®

7 - 15

Copyright © 2009, Oracle. All rights reserved.

Using the NOCOPY Hint

- Allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference rather than by value
- Enhances performance by reducing overhead when passing parameters

```
DECLARE
  TYPE      rec_emp_type IS TABLE OF employees%ROWTYPE;
  rec_emp  rec_emp_type;
  PROCEDURE populate(p_tab IN OUT NOCOPY emptabtype) IS
    BEGIN
      . . .
    END;
BEGIN
  populate(rec_emp);
END;
/
```

ORACLE®

7 - 16

Copyright © 2009, Oracle. All rights reserved.

Using the NOCOPY Hint

Note that PL/SQL subprograms support three parameter-passing modes: IN, OUT, and IN OUT. By default:

- The IN parameter is passed by reference. A pointer to the IN actual parameter is passed to the corresponding formal parameter. So, both the parameters reference the same memory location, which holds the value of the actual parameter.
- The OUT and IN OUT parameters are passed by value. The value of the OUT or IN OUT actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to the OUT and IN OUT formal parameters are copied into the corresponding actual parameters.

Copying parameters that represent large data structures (such as collections, records, and instances of object types) with OUT and IN OUT parameters slows down execution and uses up memory. To prevent this overhead, you can specify the NOCOPY hint, which enables the PL/SQL compiler to pass the OUT and IN OUT parameters by reference.

The slide shows an example of declaring an IN OUT parameter with the NOCOPY hint.

Effects of the NOCOPY Hint

- If the subprogram exits with an exception that is not handled:
 - You cannot rely on the values of the actual parameters passed to a NOCOPY parameter
 - Any incomplete modifications are not “rolled back”
- The remote procedure call (RPC) protocol enables you to pass parameters only by value.

ORACLE®

7 - 17

Copyright © 2009, Oracle. All rights reserved.

Effects of the NOCOPY Hint

As a trade-off for better performance, the NOCOPY hint enables you to trade well-defined exception semantics for better performance. Its use affects exception handling in the following ways:

- Because NOCOPY is a hint and not a directive, the compiler can pass NOCOPY parameters to a subprogram by value or by reference. So, if the subprogram exits with an unhandled exception, you cannot rely on the values of the NOCOPY actual parameters.
- By default, if a subprogram exits with an unhandled exception, the values assigned to its OUT and IN OUT formal parameters are not copied to the corresponding actual parameters, and the changes appear to roll back. However, when you specify NOCOPY, assignments to the formal parameters immediately affect the actual parameters as well. So, if the subprogram exits with an unhandled exception, the (possibly unfinished) changes are not “rolled back.”
- Currently, the RPC protocol enables you to pass parameters only by value. So, exception semantics can change without notification when you partition applications. For example, if you move a local procedure with NOCOPY parameters to a remote site, those parameters are no longer passed by reference.

When Does the PL/SQL Compiler Ignore the NOCOPY Hint?

The NOCOPY hint has no effect if:

- The actual parameter:
 - Is an element of associative arrays (index-by tables)
 - Is constrained (for example, by scale or NOT NULL)
 - And formal parameter are records, where one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ
 - Requires an implicit data type conversion
- The subprogram is involved in an external or remote procedure call

ORACLE®

7 - 18

Copyright © 2009, Oracle. All rights reserved.

When Does the PL/SQL Compiler Ignore the NOCOPY Hint?

In the following cases, the PL/SQL compiler ignores the NOCOPY hint and uses the by-value parameter-passing method (with no error generated):

- The actual parameter is an element of associative arrays (index-by tables). This restriction does not apply to entire associative arrays.
- The actual parameter is constrained (by scale or NOT NULL). This restriction does not extend to constrained elements or attributes. Also, it does not apply to size-constrained character strings.
- The actual and formal parameters are records; one or both records were declared by using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ.
- The actual and formal parameters are records; the actual parameter was declared (implicitly) as the index of a cursor FOR loop, and constraints on corresponding fields in the records differ.
- Passing the actual parameter requires an implicit data type conversion.
- The subprogram is involved in an external or remote procedure call.

Using the PARALLEL_ENABLE Hint

- Can be used in functions as an optimization hint
- Indicates that a function can be used in a parallelized query or parallelized DML statement

```
CREATE OR REPLACE FUNCTION f2 (p_p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p_p1 * 2;
END f2;
```

```
FUNCTION f2 Compiled.
```

ORACLE®

7 - 19

Copyright © 2009, Oracle. All rights reserved.

Using the PARALLEL_ENABLE Hint

The PARALLEL_ENABLE keyword can be used in the syntax for declaring a function. It is an optimization hint that indicates that the function can be used in a parallelized query or parallelized DML statement. Oracle's parallel execution feature divides the work of executing a SQL statement across multiple processes. Functions called from a SQL statement that is run in parallel can have a separate copy run in each of these processes, with each copy called for only the subset of rows that are handled by that process.

For DML statements, before Oracle8i, the parallelization optimization looked to see whether a function was noted as having all four of RNDS, WNDS, RNPS, and WNPS specified in a PRAGMA RESTRICT_REFERENCES declaration; those functions that were marked as neither reading nor writing to either the database or package variables could run in parallel. Again, those functions defined with a CREATE FUNCTION statement had their code implicitly examined to determine whether they were actually pure enough; parallelized execution might occur even though a PRAGMA cannot be specified on these functions.

The PARALLEL_ENABLE keyword is placed after the return value type in the declaration of the function, as shown in the example in the slide.

Note: The function should not use session state, such as package variables, because those variables may not be shared among the parallel execution servers.

Using the Cross-Session PL/SQL Function Result Cache

- Each time a result-cached PL/SQL function is called with different parameter values, those parameters and their results are stored in cache.
- The function result cache is stored in a shared global area (SGA), making it available to any session that runs your application.
- Subsequent calls to the same function with the same parameters uses the result from cache.
- Performance and scalability are improved.
- This feature is used with functions that are called frequently and dependent on information that changes infrequently.

ORACLE®

7 - 20

Copyright © 2009, Oracle. All rights reserved.

Cross-Session PL/SQL Function Result Cache

Starting in Oracle Database 11g, you can use the PL/SQL cross-session 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 a shared global area (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.

Each time a result-cached PL/SQL function is called with different parameter values, those parameters and their results are stored in the cache. Subsequently, when the same function is called with the same parameter values, the result is retrieved from the cache, instead of being recomputed. If a database object that was used to compute a cached result is updated, the cached result becomes invalid and must be recomputed.

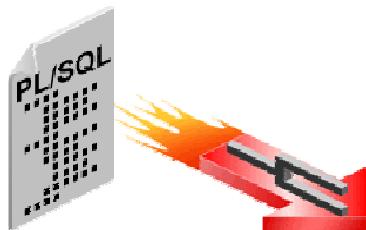
Use the result-caching feature with functions that are called frequently and are dependent on information that never changes or changes infrequently.

Note: For additional information about *Cross-Session PL/SQL Function Result Cache*, refer to the *Oracle Database 11g Advanced PL/SQL* course, the *Oracle Database 11g SQL and PL/SQL New Features* course, or the *Oracle Database PL/SQL Language Reference 11g Release 2 (11.2)* guide.

Enabling Result-Caching for a Function

You can make a function result-cached as follows:

- Include the RESULT_CACHE clause in the following:
 - The function declaration
 - The function definition
- Include an optional RELIES_ON clause to specify any tables or views on which the function results depend.



ORACLE®

7 - 21

Copyright © 2009, Oracle. All rights reserved.

Enabling Result-Caching for a Function

To enable result-caching for a PL/SQL function, use the RESULT_CACHE clause. When a result-cached function is called, the system checks the function result 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 reexecute 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 has been called. If the system needs more memory, it ages out (deletes) one or more cached results.

Note: If function execution results in an unhandled exception, the exception result is not stored in the cache.

Declaring and Defining a Result-Cached Function: Example

```
CREATE OR REPLACE FUNCTION emp_hire_date (p_emp_id
    NUMBER) RETURN VARCHAR
RESULT_CACHE RELIES_ON (employees) IS
    v_date_hired DATE;
BEGIN
    SELECT hire_date INTO v_date_hired
    FROM HR.Employees
    WHERE Employee_ID = p_emp_ID;
    RETURN to_char(v_date_hired);
END;
```

FUNCTION emp_hire_date Compiled.

ORACLE®

7 - 22

Copyright © 2009, Oracle. All rights reserved.

Declaring and Defining a Result-Cached Function: Example

If a function depends on settings that might vary from session to session such as NLS_DATE_FORMAT and TIME_ZONE, make the function result-cached only if you can modify it to handle the various settings.

In the example in the slide, the emp_hire_date function uses the to_char function to convert a DATE item to a VARCHAR item. emp_hire_date does not specify a format mask, so the format mask defaults to the one that NLS_DATE_FORMAT specifies. If sessions that call emp_hire_date have different NLS_DATE_FORMAT settings, cached results can have different formats. If a cached result computed by one session ages out, and another session recomputes it, the format might vary even for the same parameter value. If a session gets a cached result whose format differs from its own format, that result will probably be incorrect.

Some possible solutions to this problem are:

- Change the return type of emp_hire_date to DATE and have each session call the to_char function.
- If a common format is acceptable to all sessions, specify a format mask, removing the dependency on NLS_DATE_FORMAT—for example, to_char(date_hired, 'mm/dd/yy') ;.

Declaring and Defining a Result-Cached Function: Example (continued)

- Add a format mask parameter to HireDate as follows:

```
CREATE OR REPLACE FUNCTION emp_hire_date (p_emp_id NUMBER,  
                                         fmt VARCHAR) RETURN VARCHAR  
RESULT_CACHE RELIES_ON (employees) IS  
    v_date_hired DATE;  
BEGIN  
    SELECT hire_date INTO v_date_hired  
    FROM employees  
    WHERE employee_id = p_emp_id;  
    RETURN to_char(v_date_hired, fmt);  
END;
```

Using the DETERMINISTIC Clause with Functions

- Specify DETERMINISTIC to indicate that the function returns the same result value whenever it is called with the same values for its arguments.
- This helps the optimizer avoid redundant function calls.
- If a function was called previously with the same arguments, the optimizer can elect to use the previous result.
- Do not specify DETERMINISTIC for a function whose result depends on the state of session variables or schema objects.

ORACLE®

7 - 24

Copyright © 2009, Oracle. All rights reserved.

Using the DETERMINISTIC Clause with Functions

You can use the DETERMINISTIC function clause to indicate that the function returns the same result value whenever it is called with the same values for its arguments.

You must specify this keyword if you intend to call the function in the expression of a function-based index or from the query of a materialized view that is marked REFRESH FAST or ENABLE QUERY REWRITE. When Oracle Database encounters a deterministic function in one of these contexts, it attempts to use previously calculated results when possible rather than reexecuting the function. If you subsequently change the semantics of the function, you must manually rebuild all dependent function-based indexes and materialized views.

Do not specify this clause to define a function that uses package variables or that accesses the database in any way that might affect the return result of the function. The results of doing so will not be captured if Oracle Database chooses not to reexecute the function.

Note

- Do not specify DETERMINISTIC for a function whose result depends on the state of session variables or schema objects because results might vary across calls. Instead, consider making the function result-cached.
- For more information about the DETERMINISTIC clause, refer to the *Oracle Database SQL Language Reference 11g Release 1 (11.1)* guide.

Lesson Agenda

- Standardizing constants and exceptions, using local subprograms, controlling the run-time privileges of a subprogram, and performing autonomous transactions
- Using the NOCOPY and the PARALLEL_ENABLE hints, the cross-session PL/SQL function result cache, and the DETERMINISTIC clause
- Using the RETURNING clause and bulk binding with DML

ORACLE®

7 - 25

Copyright © 2009, Oracle. All rights reserved.

Using the RETURNING Clause

- Improves performance by returning column values with INSERT, UPDATE, and DELETE statements
- Eliminates the need for a SELECT statement

```
CREATE OR REPLACE PROCEDURE update_salary(p_emp_id
    NUMBER) IS
    v_name      employees.last_name%TYPE;
    v_new_sal   employees.salary%TYPE;
BEGIN
    UPDATE employees
        SET salary = salary * 1.1
    WHERE employee_id = p_emp_id
    RETURNING last_name, salary INTO v_name, v_new_sal;
    DBMS_OUTPUT.PUT_LINE(v_name || ' new salary is ' ||
        v_new_sal);
END update_salary;
/
```

ORACLE®

7 - 26

Copyright © 2009, Oracle. All rights reserved.

Using the RETURNING Clause

Often, applications need information about the row affected by a SQL operation—for example, to generate a report or to take a subsequent action. The INSERT, UPDATE, and DELETE statements can include a RETURNING clause, which returns column values from the affected row into PL/SQL variables or host variables. This eliminates the need to SELECT the row after an INSERT or UPDATE, or before a DELETE. As a result, fewer network round trips, less server CPU time, fewer cursors, and less server memory are required.

The example in the slide shows how to update the salary of an employee and, at the same time, retrieve the employee's last name and new salary into a local PL/SQL variable. To test the code example, issue the following commands that check the salary of employee_id 108 before updating it. Next, the procedure is invoked passing the employee_id 108 as the parameter.

```
1 SET SERVEROUTPUT ON
2 /
3 SELECT last_name, salary
4 FROM employees
5 WHERE employee_id = 108;
6 /
7 EXECUTE update_salary(108)
```

LAST_NAME	SALARY
Greenberg	12000

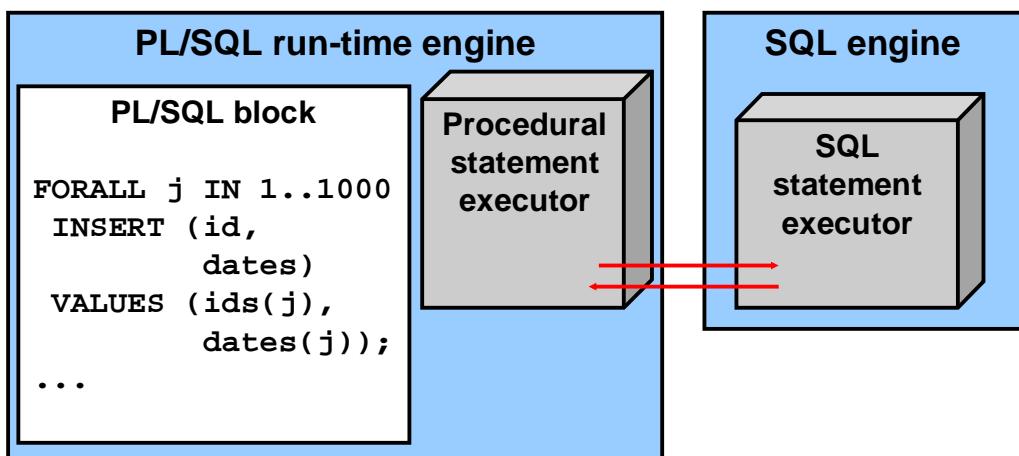
1 rows selected

anonymous block completed

Greenberg new salary is 13200

Using Bulk Binding

Binds whole arrays of values in a single operation, rather than using a loop to perform a `FETCH`, `INSERT`, `UPDATE`, and `DELETE` operation multiple times



Using Bulk Binding

The Oracle server uses two engines to run PL/SQL blocks and subprograms:

- The PL/SQL run-time engine, which runs procedural statements but passes the SQL statements to the SQL engine
- The SQL engine, which parses and executes the SQL statement and, in some cases, returns data to the PL/SQL engine

During execution, every SQL statement causes a context switch between the two engines, which results in a performance penalty for excessive amounts of SQL processing. This is typical of applications that have a SQL statement in a loop that uses values from indexed collections. Collections include nested tables, varying arrays, index-by tables, and host arrays.

Performance can be substantially improved by minimizing the number of context switches through the use of bulk binding. Bulk binding causes an entire collection to be bound in one call, a context switch, to the SQL engine. That is, a bulk bind process passes the entire collection of values back and forth between the two engines in a single context switch, compared with incurring a context switch for each collection element in an iteration of a loop. The more rows affected by a SQL statement, the greater the performance gain with bulk binding.

Bulk Binding: Syntax and Keywords

- The `FORALL` keyword instructs the *PL/SQL engine* to bulk bind input collections before sending them to the SQL engine.

```
FORALL index IN lower_bound .. upper_bound
  [SAVE EXCEPTIONS]
  sql_statement;
```

- The `BULK COLLECT` keyword instructs the *SQL engine* to bulk bind output collections before returning them to the PL/SQL engine.

```
... BULK COLLECT INTO
  collection_name[,collection_name] ...
```



Bulk Binding: Syntax and Keywords

Use bulk binds to improve the performance of:

- DML statements that reference collection elements
- SELECT statements that reference collection elements
- Cursor FOR loops that reference collections and the `RETURNING INTO` clause

The `FORALL` keyword instructs the PL/SQL engine to bulk bind input collections before sending them to the SQL engine. Although the `FORALL` statement contains an iteration scheme, it is not a `FOR` loop.

The `BULK COLLECT` keyword instructs the SQL engine to bulk bind output collections, before returning them to the PL/SQL engine. This enables you to bind locations into which SQL can return the retrieved values in bulk. Thus, you can use these keywords in the `SELECT INTO`, `FETCH INTO`, and `RETURNING INTO` clauses.

The `SAVE EXCEPTIONS` keyword is optional. However, if some of the DML operations succeed and some fail, you would want to track or report on those that fail. Using the `SAVE EXCEPTIONS` keyword causes failed operations to be stored in a cursor attribute called `%BULK_EXCEPTIONS`, which is a collection of records indicating the bulk DML iteration number and corresponding error code.

Bulk Binding: Syntax and Keywords (continued)

Handling FORALL Exceptions with the %BULK_EXCEPTIONS Attribute

To manage exceptions and have the bulk bind complete despite errors, add the keywords SAVE EXCEPTIONS to your FORALL statement after the bounds, before the DML statement.

All exceptions raised during the execution are saved in the cursor attribute %BULK_EXCEPTIONS, which stores a collection of records. Each record has two fields:

%BULK_EXCEPTIONS(i).ERROR_INDEX holds the “iteration” of the FORALL statement during which the exception was raised and %BULK_EXCEPTIONS(i).ERROR_CODE holds the corresponding Oracle error code.

Values stored in %BULK_EXCEPTIONS refer to the most recently executed FORALL statement. Its subscripts range from 1 to %BULK_EXCEPTIONS.COUNT.

Note: For additional information about bulk binding and handling bulk-binding exceptions, refer to the *Oracle Database PL/SQL User's Guide and Reference 11g Release 2 (11.2)*.

Bulk Binding FORALL: Example

```
CREATE PROCEDURE raise_salary(p_percent NUMBER) IS
  TYPE numlist_type IS TABLE OF NUMBER
    INDEX BY BINARY_INTEGER;
  v_id  numlist_type; -- collection
BEGIN
  v_id(1):= 100; v_id(2):= 102; v_id(3):= 104; v_id(4) := 110;
  -- bulk-bind the PL/SQL table
  FORALL i IN v_id.FIRST .. v_id.LAST
    UPDATE employees
      SET salary = (1 + p_percent/100) * salary
      WHERE employee_id = v_id(i);
END;
/
```

```
EXECUTE raise_salary(10)
```

anonymous block completed

ORACLE®

7 - 30

Copyright © 2009, Oracle. All rights reserved.

Bulk Binding FORALL: Example

Note: Before you can run the example in the slide, you must disable the update_job_history trigger as follows:

```
ALTER TRIGGER update_job_history DISABLE;
```

In the example in the slide, the PL/SQL block increases the salary for employees with IDs 100, 102, 104, or 110. It uses the FORALL keyword to bulk bind the collection. Without bulk binding, the PL/SQL block would have sent a SQL statement to the SQL engine for each employee record that is updated. If there are many employee records to update, then the large number of context switches between the PL/SQL engine and the SQL engine can affect performance. However, the FORALL keyword bulk binds the collection to improve performance.

Note: A looping construct is no longer required when using this feature.

Bulk Binding FORALL: Example (continued)

An Additional Cursor Attribute for DML Operations

Another cursor attribute added to support bulk operations is %BULK_ROWCOUNT. The %BULK_ROWCOUNT attribute is a composite structure designed for use with the FORALL statement. This attribute acts like an index-by table. Its i th element stores the number of rows processed by the i th execution of an UPDATE or DELETE statement. If the i th execution affects no rows, then %BULK_ROWCOUNT(i) returns zero.

Here is an example:

```
CREATE TABLE num_table (n NUMBER);
DECLARE
    TYPE num_list_type IS TABLE OF NUMBER
        INDEX BY BINARY_INTEGER;
    v_nums num_list_type;
BEGIN
    v_nums(1) := 1;
    v_nums(2) := 3;
    v_nums(3) := 5;
    v_nums(4) := 7;
    v_nums(5) := 11;
    FORALL i IN v_nums.FIRST .. v_nums.LAST
        INSERT INTO v_num_table (n) VALUES (v_nums(i));
    FOR i IN v_nums.FIRST .. v_nums.LAST
    LOOP
        dbms_output.put_line('Inserted ' ||
            SQL%BULK_ROWCOUNT(i) || ' row(s)' ||
            ' on iteration ' || i);
    END LOOP;
END;
/
DROP TABLE num_table;
```

The following results are produced by this example:

```
CREATE TABLE succeeded.
anonymous block completed
Inserted 1 row(s) on iteration 1
Inserted 1 row(s) on iteration 2
Inserted 1 row(s) on iteration 3
Inserted 1 row(s) on iteration 4
Inserted 1 row(s) on iteration 5

DROP TABLE num_table succeeded.
```

Using BULK COLLECT INTO with Queries

The SELECT statement supports the BULK COLLECT INTO syntax.

```
CREATE PROCEDURE get_departments(p_loc NUMBER) IS
    TYPE dept_tab_type IS
        TABLE OF departments%ROWTYPE;
    v_depts dept_tab_type;
BEGIN
    SELECT * BULK COLLECT INTO v_depts
    FROM departments
    WHERE location_id = p_loc;
    FOR i IN 1 .. v_depts.COUNT LOOP
        DBMS_OUTPUT.PUT_LINE(v_depts(i).department_id
            || ' ' || v_depts(i).department_name);
    END LOOP;
END;
```

ORACLE®

7 - 32

Copyright © 2009, Oracle. All rights reserved.

Using BULK COLLECT INTO with Queries

Starting with Oracle Database 10g, when using a SELECT statement in PL/SQL, you can use the bulk collection syntax shown in the example in the slide. Thus, you can quickly obtain a set of rows without using a cursor mechanism.

The example reads all the department rows for a specified region into a PL/SQL table, whose contents are displayed with the FOR loop that follows the SELECT statement.

Using BULK COLLECT INTO with Cursors

The FETCH statement has been enhanced to support the BULK COLLECT INTO syntax.

```
CREATE OR REPLACE PROCEDURE get_departments(p_loc NUMBER) IS
  CURSOR cur_dept IS
    SELECT * FROM departments
    WHERE location_id = p_loc;
  TYPE dept_tab_type IS TABLE OF cur_dept%ROWTYPE;
  v_depts dept_tab_type;
BEGIN
  OPEN cur_dept;
  FETCH cur_dept BULK COLLECT INTO v_depts;
  CLOSE cur_dept;
  FOR i IN 1 .. v_depts.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_depts(i).department_id
      ||' '|| v_depts(i).department_name);
  END LOOP;
END;
```

ORACLE®

7 - 33

Copyright © 2009, Oracle. All rights reserved.

Using BULK COLLECT INTO with Cursors

In Oracle Database 10g, when using cursors in PL/SQL, you can use a form of the FETCH statement that supports the bulk collection syntax shown in the example in the slide.

This example shows how BULK COLLECT INTO can be used with cursors.

You can also add a LIMIT clause to control the number of rows fetched in each operation. The code example in the slide could be modified as follows:

```
CREATE OR REPLACE PROCEDURE get_departments(p_loc NUMBER,
  p_nrows NUMBER) IS
  CURSOR dept_csr IS SELECT *
    FROM departments
    WHERE location_id = p_loc;
  TYPE dept_tabtype IS TABLE OF dept_csr%ROWTYPE;
  depts dept_tabtype;
BEGIN
  OPEN dept_csr;
  FETCH dept_csr BULK COLLECT INTO depts LIMIT nrows;
  CLOSE dept_csr;
  DBMS_OUTPUT.PUT_LINE(depts.COUNT||' rows read');
END;
```

Using BULK COLLECT INTO with a RETURNING Clause

```
CREATE OR REPLACE PROCEDURE raise_salary(p_rate NUMBER)
IS
  TYPE emplist_type IS TABLE OF NUMBER;
  TYPE numlist_type IS TABLE OF employees.salary%TYPE
    INDEX BY BINARY_INTEGER;
  v_emp_ids emplist_type :=
    emplist_type(100,101,102,104);
  v_new_sals numlist_type;
BEGIN
  FORALL i IN v_emp_ids.FIRST .. v_emp_ids.LAST
    UPDATE employees
      SET commission_pct = p_rate * salary
      WHERE employee_id = v_emp_ids(i)
    RETURNING salary BULK COLLECT INTO v_new_sals;
  FOR i IN 1 .. v_new_sals.COUNT LOOP
    DBMS_OUTPUT.PUT_LINE(v_new_sals(i));
  END LOOP;
END;
```

ORACLE®

7 - 34

Copyright © 2009, Oracle. All rights reserved.

Using BULK COLLECT INTO with a RETURNING Clause

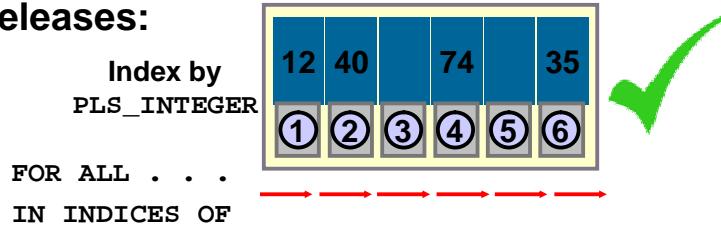
Bulk binds can be used to improve the performance of FOR loops that reference collections and return DML. If you have, or plan to have, PL/SQL code that does this, then you can use the FORALL keyword along with the RETURNING and BULK COLLECT INTO keywords to improve performance.

In the example shown in the slide, the salary information is retrieved from the EMPLOYEES table and collected into the new_sals array. The new_sals collection is returned in bulk to the PL/SQL engine.

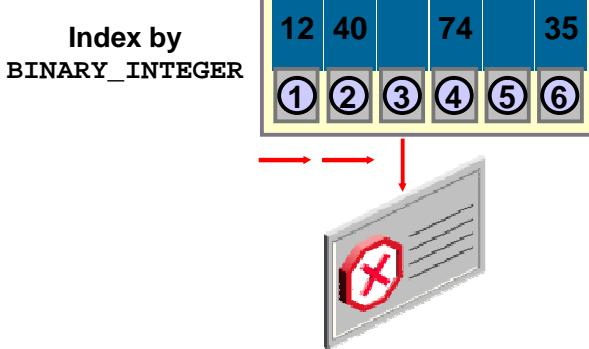
The example in the slide shows an incomplete FOR loop that is used to iterate through the new salary data received from the UPDATE operation and then process the results.

Using Bulk Binds in Sparse Collections

Current releases:



Prior releases:



ORACLE®

FORALL Support for Sparse Collections

- In earlier releases, PL/SQL did not allow sparse collections to be used with the FORALL statement.
- If the SAVE EXCEPTIONS keyword was not specified, the statement was terminated when the first deleted element was encountered. Even when SAVE EXCEPTION was used, the PL/SQL engine tried to iterate over all elements (existing and non-existing). This substantially degraded the performance of the DML operation if the relative percentage of the deleted elements was high.
- When you use the keyword INDICES (available with Oracle Database 10g and later), you can successfully loop over a sparse collection with the FORALL statement. This syntax binds sparse collections more efficiently and it also supports a more general approach where an index array can be specified to iterate over the collections.
- Using sparse collection and index arrays in bulk operations improves performance.

Using Bulk Binds in Sparse Collections

```
-- The INDICES OF syntax allows the bound arrays
-- themselves to be sparse.

FORALL index_name IN INDICES OF sparse_array_name
    BETWEEN LOWER_BOUND AND UPPER_BOUND -- optional
    SAVE EXCEPTIONS -- optional, but recommended
    INSERT INTO table_name VALUES
    sparse_array(index_name);
. . .
```

```
-- The VALUES OF syntax lets you indicate a subset
-- of the binding arrays.

FORALL index_name IN VALUES OF index_array_name
    SAVE EXCEPTIONS -- optional, but recommended
    INSERT INTO table_name VALUES
    binding_array_name(index_name);
. . .
```

ORACLE®

7 - 36

Copyright © 2009, Oracle. All rights reserved.

FORALL Support for Sparse Collections (continued)

- You can use the INDICES OF and VALUES OF syntax with the FORALL statement.
- The bulk bind for sparse array syntax can be used in all DML syntaxes.
- In the syntax, the index array must be dense, and the binding arrays may be dense or sparse and the indicated elements must exist.

Using Bulk Binds in Sparse Collections

The typical application for this feature is an order entry and order processing system where:

- Users enter orders through the Web
- Orders are placed in a staging table before validation
- Data is later validated based on complex business rules (usually implemented programmatically using PL/SQL)
- Invalid orders are separated from valid ones
- Valid orders are inserted into a permanent table for processing

ORACLE®

7 - 37

Copyright © 2009, Oracle. All rights reserved.

Using Bulk Binds in Sparse Collections

This feature can be used in any application that starts with a dense PL/SQL table of records or table of scalar that are populated using a bulk collect. This is used as the binding array. A dense array (pointer), whose elements denote the indices of the binding array, is made sparse based on the application logic. This pointer array is then used in the FORALL statement to perform bulk DML with the binding arrays. Any exceptions encountered can be saved and further processed in the exception-handling section, perhaps by using another FORALL statement.

Using Bulk Bind with Index Array

```
CREATE OR REPLACE PROCEDURE ins_emp2 AS
  TYPE emptab_type IS TABLE OF employees%ROWTYPE;
  v_emp emptab_type;
  TYPE values_of_tab_type IS TABLE OF PLS_INTEGER
    INDEX BY PLS_INTEGER;
  v_num   values_of_tab_type;
  . . .
BEGIN
  . . .
    FORALL k IN VALUES OF v_num
      INSERT INTO new_employees VALUES v_emp(k);
END;
```

ORACLE®

7 - 38

Copyright © 2009, Oracle. All rights reserved.

Using Bulk Bind with Index Array

You can use an index collection of PLS_INTEGER or BINARY_INTEGER (or one of its subtypes) whose values are the indexes of the collections involved in the bulk-bind DML operation using FORALL. These index collections can then be used in a FORALL statement to process bulk DML using the VALUES_OF clause.

In the example shown above, V_NUM is a collection whose type is PLS_INTEGER. In the example, you are creating a procedure INS_EMP2, which identifies only one employee for each occurrence of the first letter of the last name. This procedure then inserts into the NEW_EMPLOYEES table created earlier using the FORALL..IN VALUES_OF syntax.

Quiz

The NOCOPY hint allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference rather than by value. This enhances performance by reducing overhead when passing parameters.

1. True
2. False



Answer: 1

PL/SQL subprograms support three parameter-passing modes: IN, OUT, and IN OUT.

By default:

- The IN parameter is passed by reference. A pointer to the IN actual parameter is passed to the corresponding formal parameter. So, both the parameters reference the same memory location, which holds the value of the actual parameter.
- The OUT and IN OUT parameters are passed by value. The value of the OUT or IN OUT actual parameter is copied into the corresponding formal parameter. Then, if the subprogram exits normally, the values assigned to the OUT and IN OUT formal parameters are copied into the corresponding actual parameters.

Copying parameters that represent large data structures (such as collections, records, and instances of object types) with OUT and IN OUT parameters slows down execution and uses up memory. To prevent this overhead, you can specify the NOCOPY hint, which enables the PL/SQL compiler to pass OUT and IN OUT parameters by reference.

Summary

In this lesson, you should have learned how to:

- Create standard constants and exceptions
- Write and call local subprograms
- Control the run-time privileges of a subprogram
- Perform autonomous transactions
- Pass parameters by reference using a NOCOPY hint
- Use the PARALLEL_ENABLE hint for optimization
- Use the cross-session PL/SQL function result cache
- Use the DETERMINISTIC clause with functions
- Use the RETURNING clause and bulk binding with DML



Summary

The lesson provides insights into managing your PL/SQL code by defining constants and exceptions in a package specification. This enables a high degree of reuse and standardization of code.

Local subprograms can be used to simplify and modularize a block of code where the subprogram functionality is repeatedly used in the local block.

The run-time security privileges of a subprogram can be altered by using definer's or invoker's rights.

Autonomous transactions can be executed without affecting an existing main transaction.

You should understand how to obtain performance gains by using the NOCOPY hint, bulk binding and the RETURNING clauses in SQL statements, and the PARALLEL_ENABLE hint for optimization of functions.

Practice 7: Overview

This practice covers the following topics:

- Creating a package that uses bulk fetch operations
- Creating a local subprogram to perform an autonomous transaction to audit a business operation



Practice 7: Overview

In this practice, you create a package that performs a bulk fetch of employees in a specified department. The data is stored in a PL/SQL table in the package. You also provide a procedure to display the contents of the table.

You add an `add_employee` procedure to the package that inserts new employees. The procedure uses a local autonomous subprogram to write a log record each time the `add_employee` procedure is called, whether it successfully adds a record or not.

8

Creating Triggers

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

Objectives

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

- Describe database triggers and their uses
- Describe the different types of triggers
- Create database triggers
- Describe database trigger-firing rules
- Remove database triggers
- Display trigger information

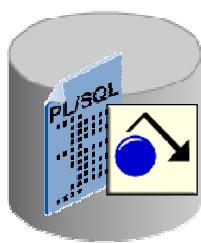


Lesson Aim

In this lesson, you learn how to create and use database triggers.

What Are Triggers?

- A trigger is a PL/SQL block that is stored in the database and fired (executed) in response to a specified event.
- The Oracle database automatically executes a trigger when specified conditions occur.



ORACLE®

8 - 3

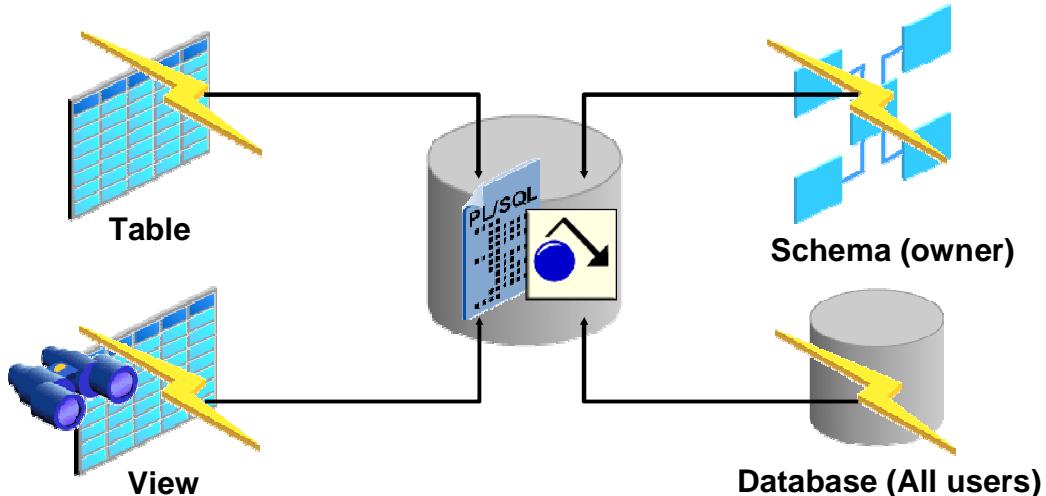
Copyright © 2009, Oracle. All rights reserved.

Working with Triggers: Overview

Triggers are similar to stored procedures. A trigger stored in the database contains PL/SQL in the form of an anonymous block, a call statement, or a compound trigger block. However, procedures and triggers differ in the way that they are invoked. A procedure is explicitly run by a user, application, or trigger. Triggers are implicitly fired by the Oracle database when a triggering event occurs, no matter which user is connected or which application is being used.

Defining Triggers

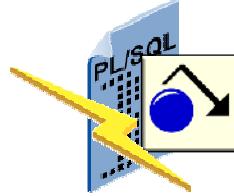
A trigger can be defined on the table, view, schema (schema owner), or database (all users).



Trigger Event Types

You can write triggers that fire whenever one of the following operations occurs in the database:

- A database manipulation (DML) statement (`DELETE`, `INSERT`, or `UPDATE`).
- A database definition (DDL) statement (`CREATE`, `ALTER`, or `DROP`).
- A database operation such as `SERVERERROR`, `LOGON`, `LOGOFF`, `STARTUP`, or `SHUTDOWN`.



ORACLE®

8 - 5

Copyright © 2009, Oracle. All rights reserved.

Triggering Event or Statement

A triggering event or statement is the SQL statement, database event, or user event that causes a trigger to fire. A triggering event can be one or more of the following:

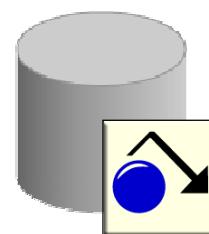
- An `INSERT`, `UPDATE`, or `DELETE` statement on a specific table (or view, in some cases)
- A `CREATE`, `ALTER`, or `DROP` statement on any schema object
- A database startup or instance shutdown
- A specific error message or any error message
- A user logon or logoff

Application and Database Triggers

- Database trigger (covered in this course):
 - Fires whenever a DML, a DLL, or system event occurs on a schema or database
- Application trigger:
 - Fires whenever an event occurs within a particular application



Application Trigger



Database Trigger

ORACLE®

Types of Triggers

Application triggers execute implicitly whenever a particular data manipulation language (DML) event occurs within an application. An example of an application that uses triggers extensively is an application developed with Oracle Forms Developer.

Database triggers execute implicitly when any of the following events occur:

- DML operations on a table
- DML operations on a view, with an INSTEAD OF trigger
- DDL statements, such as CREATE and ALTER

This is the case no matter which user is connected or which application is used. Database triggers also execute implicitly when some user actions or database system actions occur (for example, when a user logs on or the DBA shuts down the database).

Database triggers can be system triggers on a database or a schema (covered in the next lesson). For databases, triggers fire for each event for all users; for a schema, they fire for each event for that specific user. Oracle Forms can define, store, and run triggers of a different sort. However, do not confuse Oracle Forms triggers with the triggers discussed in this lesson.

Business Application Scenarios for Implementing Triggers

You can use triggers for:

- Security
- Auditing
- Data integrity
- Referential integrity
- Table replication
- Computing derived data automatically
- Event logging



Business Application Scenarios for Implementing Triggers

Develop database triggers in order to enhance features that cannot otherwise be implemented by the Oracle server or as alternatives to those provided by the Oracle server.

- **Security:** The Oracle server allows table access to users or roles. Triggers allow table access according to data values.
- **Auditing:** The Oracle server tracks data operations on tables. Triggers track values for data operations on tables.
- **Data integrity:** The Oracle server enforces integrity constraints. Triggers implement complex integrity rules.
- **Referential integrity:** The Oracle server enforces standard referential integrity rules. Triggers implement nonstandard functionality.
- **Table replication:** The Oracle server copies tables asynchronously into snapshots. Triggers copy tables synchronously into replicas.
- **Derived data:** The Oracle server computes derived data values manually. Triggers compute derived data values automatically.
- **Event logging:** The Oracle server logs events explicitly. Triggers log events transparently.

Available Trigger Types

- Simple DML triggers
 - BEFORE
 - AFTER
 - INSTEAD OF
- Compound triggers
- Non-DML triggers
 - DDL event triggers
 - Database event triggers

ORACLE®

8 - 8

Copyright © 2009, Oracle. All rights reserved.

Note

In this lesson, we will discuss the BEFORE, AFTER, and INSTEAD OF triggers. The other trigger types are discussed in the lesson titled “Creating Compound, DDL, and Event Database Triggers.”

Trigger Event Types and Body

- A trigger event type determines which DML statement causes the trigger to execute. The possible events are:
 - INSERT
 - UPDATE [OF column]
 - DELETE
- A trigger body determines what action is performed and is a PL/SQL block or a CALL to a procedure.

ORACLE®

8 - 9

Copyright © 2009, Oracle. All rights reserved.

Triggering Event Types

The triggering event or statement can be an INSERT, UPDATE, or DELETE statement on a table.

- When the triggering event is an UPDATE statement, you can include a column list to identify which columns must be changed to fire the trigger. You cannot specify a column list for an INSERT or for a DELETE statement because it always affects entire rows.
 . . . UPDATE OF salary . . .
- The triggering event can contain one, two, or all three of these DML operations.
 . . . INSERT or UPDATE or DELETE
 . . . INSERT or UPDATE OF job_id . . .

The trigger body defines the action—that is, what needs to be done when the triggering event is issued. The PL/SQL block can contain SQL and PL/SQL statements, and can define PL/SQL constructs such as variables, cursors, exceptions, and so on. You can also call a PL/SQL procedure or a Java procedure.

Note: The size of a trigger cannot be greater than 32 KB.

Creating DML Triggers Using the CREATE TRIGGER Statement

```
CREATE [OR REPLACE] TRIGGER trigger_name  
timing -- when to fire the trigger  
event1 [OR event2 OR event3]  
ON object_name  
[REFERENCING OLD AS old / NEW AS new]  
FOR EACH ROW -- default is statement level trigger  
WHEN (condition)])  
DECLARE ]  
BEGIN  
... trigger_body -- executable statements  
[EXCEPTION . . .]  
END [trigger_name];
```

timing = BEFORE | AFTER | INSTEAD OF

event = INSERT | DELETE | UPDATE | UPDATE OF column_list

ORACLE®

8 - 10

Copyright © 2009, Oracle. All rights reserved.

Creating DML Triggers

The components of the trigger syntax are:

- *trigger_name* uniquely identifies the trigger.
- *timing* indicates when the trigger fires in relation to the triggering event. Values are BEFORE, AFTER, and INSTEAD OF.
- *event* identifies the DML operation causing the trigger to fire. Values are INSERT, UPDATE [OF column], and DELETE.
- *object_name* indicates the table or view associated with the trigger.
- For row triggers, you can specify:
 - A REFERENCING clause to choose correlation names for referencing the old and new values of the current row (default values are OLD and NEW)
 - FOR EACH ROW to designate that the trigger is a row trigger
 - A WHEN clause to apply a conditional predicate, in parentheses, which is evaluated for each row to determine whether or not to execute the trigger body
- The *trigger_body* is the action performed by the trigger, implemented as either of the following:
 - An anonymous block with a DECLARE or BEGIN, and an END
 - A CALL clause to invoke a stand-alone or packaged stored procedure, such as:
CALL my_procedure;

Specifying the Trigger Firing (Timing)

You can specify the trigger timing as to whether to run the trigger's action before or after the triggering statement:

- **BEFORE**: Executes the trigger body before the triggering DML event on a table.
- **AFTER**: Execute the trigger body after the triggering DML event on a table.
- **INSTEAD OF**: Execute the trigger body instead of the triggering statement. This is used for views that are not otherwise modifiable.

ORACLE®

8 - 11

Copyright © 2009, Oracle. All rights reserved.

Trigger Timing

The **BEFORE** trigger timing is frequently used in the following situations:

- To determine whether the triggering statement should be allowed to complete (This eliminates unnecessary processing and enables a rollback in cases where an exception is raised in the triggering action.)
- To derive column values before completing an `INSERT` or `UPDATE` statement
- To initialize global variables or flags, and to validate complex business rules

The **AFTER** triggers are frequently used in the following situations:

- To complete the triggering statement before executing the triggering action
- To perform different actions on the same triggering statement if a BEFORE trigger is already present

The **INSTEAD OF** triggers provide a transparent way of modifying views that cannot be modified directly through SQL DML statements because a view is not always modifiable. You can write appropriate DML statements inside the body of an `INSTEAD OF` trigger to perform actions directly on the underlying tables of views.

If it is practical, replace the set of individual triggers with different timing points with a single compound trigger that explicitly codes the actions in the order you intend. If two or more triggers are defined with the same timing point, and the order in which they fire is important, then you can control the firing order using the `FOLLOWS` and `PRECEDES` clauses.

Statement-Level Triggers Versus Row-Level Triggers

Statement-Level Triggers	Row-Level Triggers
Is the default when creating a trigger	Use the FOR EACH ROW clause when creating a trigger.
Fires once for the triggering event	Fires once for each row affected by the triggering event
Fires once even if no rows are affected	Does not fire if the triggering event does not affect any rows

ORACLE®

8 - 12

Copyright © 2009, Oracle. All rights reserved.

Types of DML Triggers

You can specify that the trigger will be executed once for every row affected by the triggering statement (such as a multiple row UPDATE) or once for the triggering statement, no matter how many rows it affects.

Statement Trigger

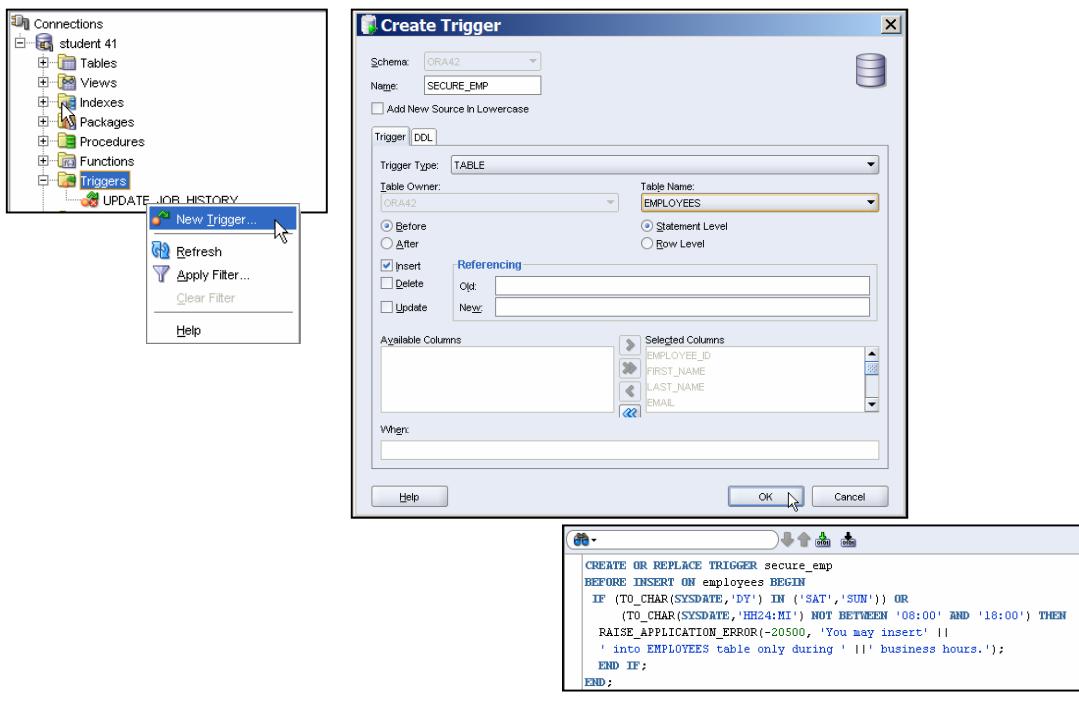
A statement trigger is fired once on behalf of the triggering event, even if no rows are affected at all. Statement triggers are useful if the trigger action does not depend on the data from rows that are affected or on data provided by the triggering event itself (for example, a trigger that performs a complex security check on the current user).

Row Trigger

A row trigger fires each time the table is affected by the triggering event. If the triggering event affects no rows, a row trigger is not executed. Row triggers are useful if the trigger action depends on data of the rows that are affected or on data provided by the triggering event itself.

Note: Row triggers use correlation names to access the old and new column values of the row being processed by the trigger.

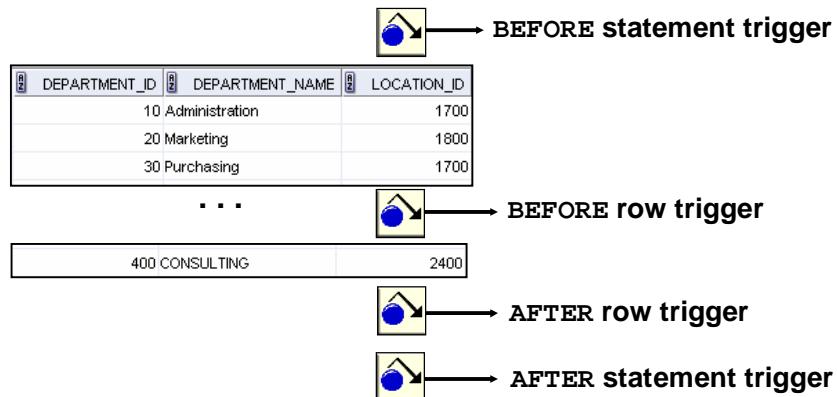
Creating DML Triggers Using SQL Developer



Trigger-Firing Sequence: Single-Row Manipulation

Use the following firing sequence for a trigger on a table when a single row is manipulated:

```
INSERT INTO departments
  (department_id, department_name, location_id)
VALUES (400, 'CONSULTING', 2400);
```



ORACLE®

Trigger-Firing Sequence: Single-Row Manipulation

Create a statement trigger or a row trigger based on the requirement that the trigger must fire once for each row affected by the triggering statement, or just once for the triggering statement, regardless of the number of rows affected.

When the triggering DML statement affects a single row, both the statement trigger and the row trigger fire exactly once.

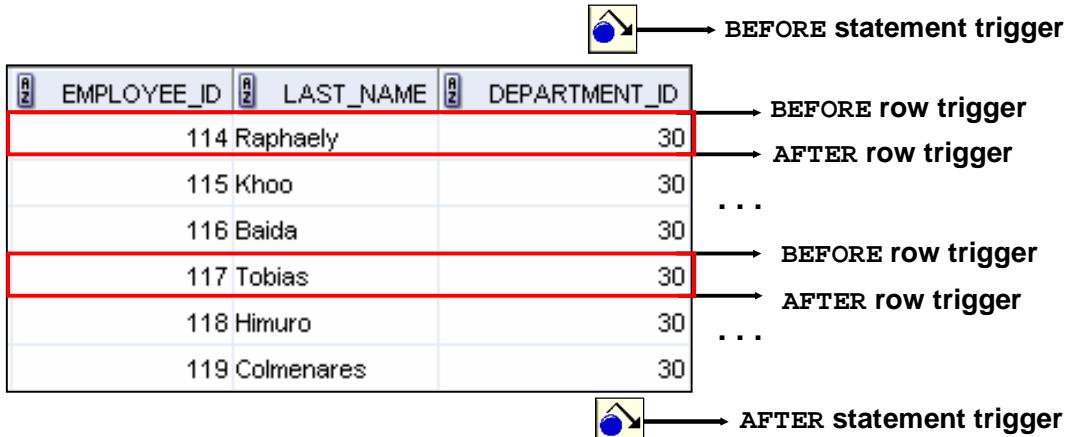
Example

The SQL statement in the slide does not differentiate statement triggers from row triggers because exactly one row is inserted into the table using the syntax for the `INSERT` statement shown in the slide.

Trigger-Firing Sequence: Multirow Manipulation

Use the following firing sequence for a trigger on a table when many rows are manipulated:

```
UPDATE employees
    SET salary = salary * 1.1
  WHERE department_id = 30;
```



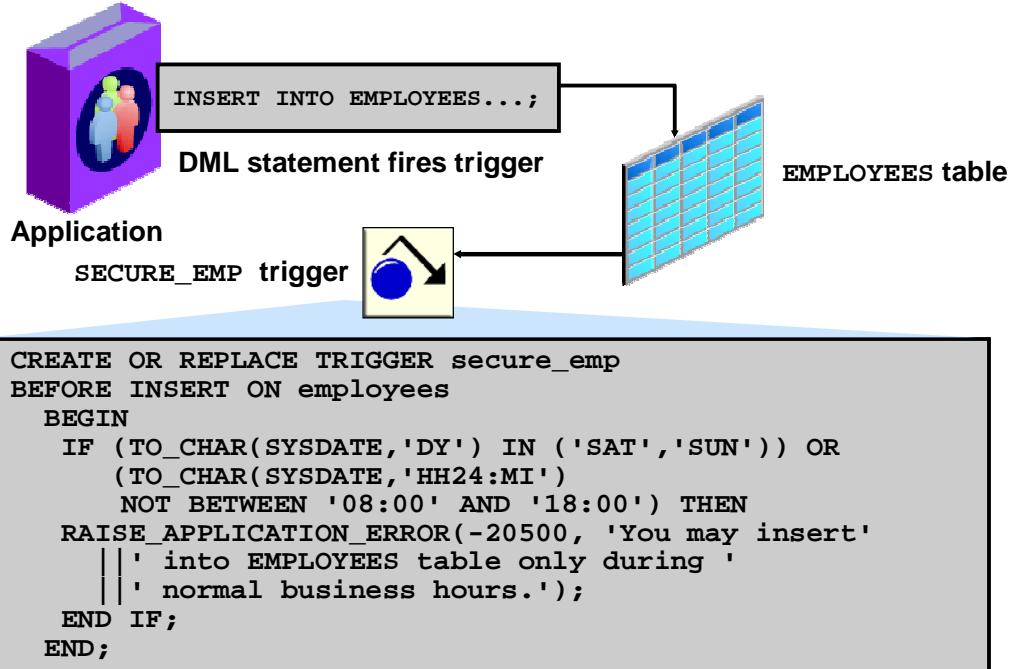
Trigger-Firing Sequence: Multirow Manipulation

When the triggering DML statement affects many rows, the statement trigger fires exactly once, and the row trigger fires once for every row affected by the statement.

Example

The SQL statement in the slide causes a row-level trigger to fire a number of times equal to the number of rows that satisfy the WHERE clause (that is, the number of employees reporting to department 30).

Creating a DML Statement Trigger Example: SECURE_EMP



Creating a DML Statement Trigger

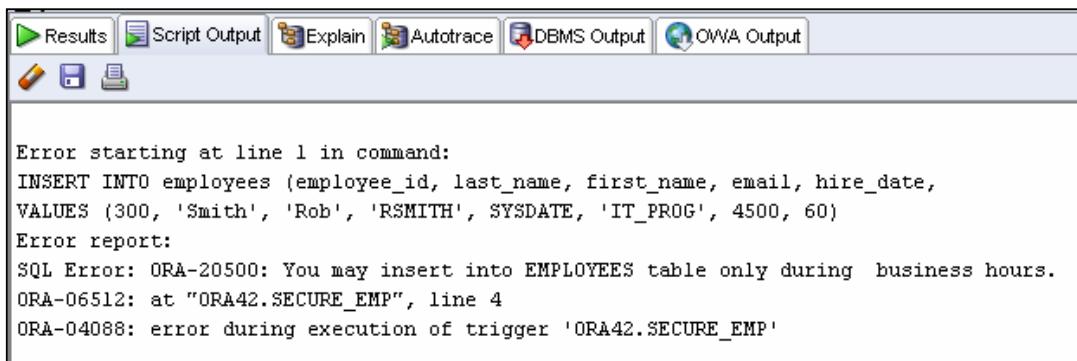
In the example in the slide, the SECURE_EMP database trigger is a BEFORE statement trigger that prevents the INSERT operation from succeeding if the business condition is violated. In this case, the trigger restricts inserts into the EMPLOYEES table during certain business hours, Monday through Friday.

If a user attempts to insert a row into the EMPLOYEES table on Saturday, then the user sees an error message, the trigger fails, and the triggering statement is rolled back. Remember that the RAISE_APPLICATION_ERROR is a server-side built-in procedure that returns an error to the user and causes the PL/SQL block to fail.

When a database trigger fails, the triggering statement is automatically rolled back by the Oracle server.

Testing Trigger SECURE_EMP

```
INSERT INTO employees (employee_id, last_name,
    first_name, email, hire_date, job_id, salary,
    department_id)
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE,
    'IT_PROG', 4500, 60);
```



The screenshot shows the Oracle SQL Developer interface. At the top, there is a toolbar with several tabs: Results, Script Output, Explain, Autotrace, DBMS Output, and OWA Output. Below the toolbar, there are three small icons: a pencil, a square, and a document. The main area displays an error message:

```
Error starting at line 1 in command:  
INSERT INTO employees (employee_id, last_name, first_name, email, hire_date,  
VALUES (300, 'Smith', 'Rob', 'RSMITH', SYSDATE, 'IT_PROG', 4500, 60)  
Error report:  
SQL Error: ORA-20500: You may insert into EMPLOYEES table only during business hours.  
ORA-06512: at "ORA42.SECURE_EMP", line 4  
ORA-04088: error during execution of trigger 'ORA42.SECURE_EMP'
```

ORACLE®

8 - 17

Copyright © 2009, Oracle. All rights reserved.

Testing SECURE_EMP

To test the trigger, insert a row into the EMPLOYEES table during nonbusiness hours. When the date and time are out of the business hours specified in the trigger, you receive the error message shown in the slide.

Using Conditional Predicates

```
CREATE OR REPLACE TRIGGER secure_emp BEFORE
INSERT OR UPDATE OR DELETE ON employees
BEGIN
    IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
       (TO_CHAR(SYSDATE,'HH24') NOT BETWEEN '08' AND '18') THEN
        IF DELETING THEN RAISE_APPLICATION_ERROR(
            -20502,'You may delete from EMPLOYEES table'|||
            'only during normal business hours.');
        ELSIF INSERTING THEN RAISE_APPLICATION_ERROR(
            -20500,'You may insert into EMPLOYEES table'|||
            'only during normal business hours.');
        ELSIF UPDATING ('SALARY') THEN
            RAISE_APPLICATION_ERROR(-20503, 'You may '|||
            'update SALARY only normal during business hours.');
        ELSE RAISE_APPLICATION_ERROR(-20504,'You may' |||
            ' update EMPLOYEES table only during'|||
            ' normal business hours.');
        END IF;
    END IF;
END;
```

ORACLE®

8 - 18

Copyright © 2009, Oracle. All rights reserved.

Detecting the DML Operation That Fired a Trigger

If more than one type of DML operation can fire a trigger (for example, ON INSERT OR DELETE OR UPDATE OF Emp_tab), the trigger body can use the conditional predicates INSERTING, DELETING, and UPDATING to check which type of statement fired the trigger.

You can combine several triggering events into one by taking advantage of the special conditional predicates INSERTING, UPDATING, and DELETING within the trigger body.

Example

Create one trigger to restrict all data manipulation events on the EMPLOYEES table to certain business hours, 8 AM to 6 PM, Monday through Friday.

Creating a DML Row Trigger

```
CREATE OR REPLACE TRIGGER restrict_salary
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
BEGIN
    IF NOT (:NEW.job_id IN ('AD_PRES', 'AD_VP'))
        AND :NEW.salary > 15000 THEN
        RAISE_APPLICATION_ERROR (-20202,
            'Employee cannot earn more than $15,000.');
    END IF;
END;
```

```
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell';
```

```
Error starting at line 1 in command:
UPDATE employees
SET salary = 15500
WHERE last_name = 'Russell'
Error report:
SQL Error: ORA-20202: Employee cannot earn more than $15,000.
ORA-06512: at "ORA62.RESTRICT_SALARY", line 4
ORA-04088: error during execution of trigger 'ORA62.RESTRICT_SALARY'
```

ORACLE®

8 - 19

Copyright © 2009, Oracle. All rights reserved.

Creating a DML Row Trigger

You can create a BEFORE row trigger in order to prevent the triggering operation from succeeding if a certain condition is violated.

In the first example in the slide, a trigger is created to allow only employees whose job IDs are either AD_PRES or AD_VP to earn a salary of more than 15,000. If you try to update the salary of employee Russell whose employee ID is SA_MAN, the trigger raises the exception displayed in the slide.

Note: Before executing the first code example in the slide, make sure you disable the secure_emp and secure_employees triggers.

Using OLD and NEW Qualifiers

- When a row-level trigger fires, the PL/SQL run-time engine creates and populates two data structures:
 - OLD: Stores the original values of the record processed by the trigger
 - NEW: Contains the new values
- NEW and OLD have the same structure as a record declared using the %ROWTYPE on the table to which the trigger is attached.

Data Operations	Old Value	New Value
INSERT	NULL	Inserted value
UPDATE	Value before update	Value after update
DELETE	Value before delete	NULL

ORACLE®

8 - 20

Copyright © 2009, Oracle. All rights reserved.

Using OLD and NEW Qualifiers

Within a ROW trigger, you can reference the value of a column before and after the data change by prefixing it with the OLD and NEW qualifiers.

Note

- The OLD and NEW qualifiers are available only in ROW triggers.
- Prefix these qualifiers with a colon (:) in every SQL and PL/SQL statement.
- There is no colon (:) prefix if the qualifiers are referenced in the WHEN restricting condition.
- Row triggers can decrease the performance if you perform many updates on larger tables.

Using OLD and NEW Qualifiers: Example

```
CREATE TABLE audit_emp (
    user_name      VARCHAR2(30),
    time_stamp     date,
    id             NUMBER(6),
    old_last_name VARCHAR2(25),
    new_last_name VARCHAR2(25),
    old_title      VARCHAR2(10),
    new_title      VARCHAR2(10),
    old_salary     NUMBER(8,2),
    new_salary     NUMBER(8,2) )

/
CREATE OR REPLACE TRIGGER audit_emp_values
AFTER DELETE OR INSERT OR UPDATE ON employees
FOR EACH ROW
BEGIN
    INSERT INTO audit_emp(user_name, time_stamp, id,
        old_last_name, new_last_name, old_title,
        new_title, old_salary, new_salary)
    VALUES (USER, SYSDATE, :OLD.employee_id,
        :OLD.last_name, :NEW.last_name, :OLD.job_id,
        :NEW.job_id, :OLD.salary, :NEW.salary);
END;
```

ORACLE®

8 - 21

Copyright © 2009, Oracle. All rights reserved.

Using OLD and NEW Qualifiers: Example

In the example in the slide, the AUDIT_EMP_VALUES trigger is created on the EMPLOYEES table. The trigger adds rows to a user table, AUDIT_EMP, logging a user's activity against the EMPLOYEES table. The trigger records the values of several columns both before and after the data changes by using the OLD and NEW qualifiers with the respective column name.

Using OLD and NEW Qualifiers: Example

```
INSERT INTO employees (employee_id, last_name, job_id,
salary, email, hire_date)
VALUES (999, 'Temp emp', 'SA_REP', 6000, 'TEMPEMP',
TRUNC(SYSDATE))
/
UPDATE employees
SET salary = 7000, last_name = 'Smith'
WHERE employee_id = 999
/
SELECT *
FROM audit_emp;
```

USER_NAME	TIME_STAMP	ID	OLD_LAST_NAME	NEW_LAST_NAME	OLD_TITLE	NEW_TITLE	OLD_SALARY	NEW_SALARY
ORA61	04-JUN-09	(null)	Temp emp	(null)	SA_REP	(null)	6000	6000
ORA61	04-JUN-09	999	Temp emp	Smith	SA_REP	SA_REP	6000	7000

ORACLE®

8 - 22

Copyright © 2009, Oracle. All rights reserved.

Using OLD and NEW Qualifiers: Example the Using AUDIT_EMP Table

Create a trigger on the EMPLOYEES table to add rows to a user table, AUDIT_EMP, logging a user's activity against the EMPLOYEES table. The trigger records the values of several columns both before and after the data changes by using the OLD and NEW qualifiers with the respective column name.

The following is the result of inserting the employee record into the EMPLOYEES table:

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
999 (null)	Smith	TEMPEMP	(null)	04-JUN-09	SA_REP	7000	(null)	(null)	(null)	(null)
300	Rob	RSMITH	(null)	04-JUN-09	IT_PROG	4500	(null)	(null)	(null)	60
206	William	Gietz	WGIETZ	515.123.8181	07-JUN-94	AC_ACCOUNT	8300	(null)	205	110

The following is the result of updating the salary for employee "Smith":

EMPLOYEE_ID	FIRST_NAME	LAST_NAME	EMAIL	PHONE_NUMBER	HIRE_DATE	JOB_ID	SALARY	COMMISSION_PCT	MANAGER_ID	DEPARTMENT_ID
999 (null)	Smith	TEMPEMP	(null)	04-JUN-09	SA_REP	7000	(null)	(null)	(null)	(null)

Using the WHEN Clause to Fire a Row Trigger Based on a Condition

```
CREATE OR REPLACE TRIGGER derive_commission_pct
BEFORE INSERT OR UPDATE OF salary ON employees
FOR EACH ROW
WHEN (NEW.job_id = 'SA_REP')
BEGIN
  IF INSERTING THEN
    :NEW.commission_pct := 0;
  ELSIF :OLD.commission_pct IS NULL THEN
    :NEW.commission_pct := 0;
  ELSE
    :NEW.commission_pct := :OLD.commission_pct+0.05;
  END IF;
END;
/
```

ORACLE®

8 - 23

Copyright © 2009, Oracle. All rights reserved.

Restricting a Row Trigger: Example

Optionally, you can include a trigger restriction in the definition of a row trigger by specifying a Boolean SQL expression in a WHEN clause. If you include a WHEN clause in the trigger, then the expression in the WHEN clause is evaluated for each row that the trigger affects.

If the expression evaluates to TRUE for a row, then the trigger body executes on behalf of that row. However, if the expression evaluates to FALSE or NOT TRUE for a row (unknown, as with nulls), then the trigger body does not execute for that row. The evaluation of the WHEN clause does not have an effect on the execution of the triggering SQL statement (in other words, the triggering statement is not rolled back if the expression in a WHEN clause evaluates to FALSE).

Note: A WHEN clause cannot be included in the definition of a statement trigger.

In the example in the slide, a trigger is created on the EMPLOYEES table to calculate an employee's commission when a row is added to the EMPLOYEES table, or when an employee's salary is modified.

The NEW qualifier cannot be prefixed with a colon in the WHEN clause because the WHEN clause is outside the PL/SQL blocks.

Summary of the Trigger Execution Model

1. Execute all BEFORE STATEMENT triggers.
2. Loop *for each row affected by the SQL statement*:
 - a. Execute all BEFORE ROW triggers *for that row*.
 - b. Execute the DML statement and perform integrity constraint checking *for that row*.
 - c. Execute all AFTER ROW triggers *for that row*.
3. Execute all AFTER STATEMENT triggers.

ORACLE®

8 - 24

Copyright © 2009, Oracle. All rights reserved.

Trigger Execution Model

A single DML statement can potentially fire up to four types of triggers:

- BEFORE and AFTER statement triggers
- BEFORE and AFTER row triggers

A triggering event or a statement within the trigger can cause one or more integrity constraints to be checked. However, you can defer constraint checking until a COMMIT operation is performed.

Triggers can also cause other triggers—known as cascading triggers—to fire.

All actions and checks performed as a result of a SQL statement must succeed. If an exception is raised within a trigger and the exception is not explicitly handled, then all actions performed because of the original SQL statement are rolled back (including actions performed by firing triggers). This guarantees that integrity constraints can never be compromised by triggers.

When a trigger fires, the tables referenced in the trigger action may undergo changes by other users' transactions. In all cases, a read-consistent image is guaranteed for the modified values that the trigger needs to read (query) or write (update).

Note: Integrity checking can be deferred until the COMMIT operation is performed.

Implementing an Integrity Constraint with an After Trigger

```
-- Integrity constraint violation error -2991 raised.  
UPDATE employees SET department_id = 999  
WHERE employee_id = 170;
```

```
CREATE OR REPLACE TRIGGER employee_dept_fk_trg  
AFTER UPDATE OF department_id ON employees  
FOR EACH ROW  
BEGIN  
    INSERT INTO departments VALUES(:new.department_id,  
        'Dept '||:new.department_id, NULL, NULL);  
EXCEPTION  
    WHEN DUP_VAL_ON_INDEX THEN  
        NULL; -- mask exception if department exists  
END;  
/
```

```
-- Successful after trigger is fired  
UPDATE employees SET department_id = 999  
WHERE employee_id = 170;
```

1 rows updated

ORACLE®

8 - 25

Copyright © 2009, Oracle. All rights reserved.

Implementing an Integrity Constraint with an After Trigger

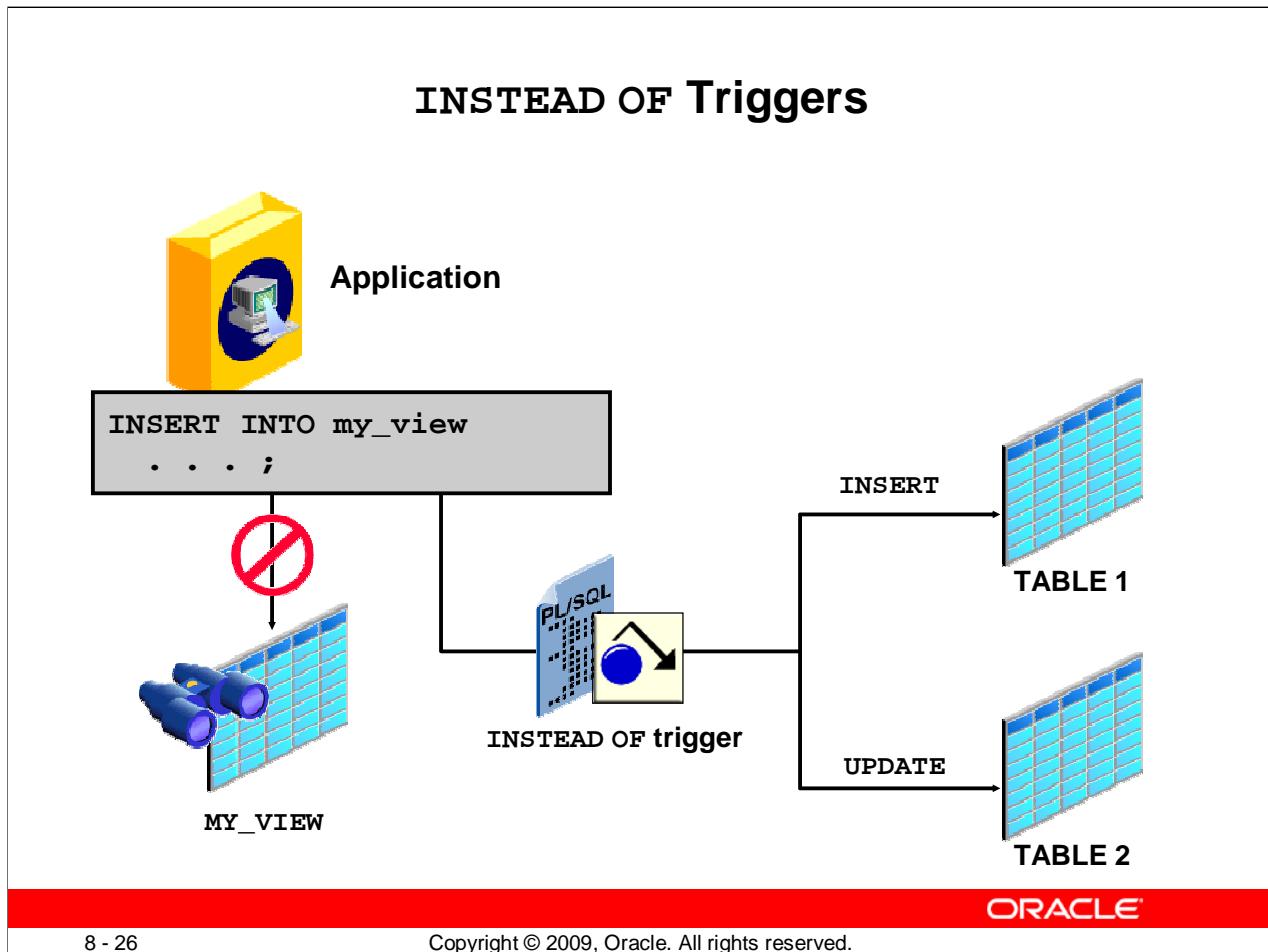
The example in the slide explains a situation in which the integrity constraint can be taken care of by using an AFTER trigger. The EMPLOYEES table has a foreign key constraint on the DEPARTMENT_ID column of the DEPARTMENTS table.

In the first SQL statement, the DEPARTMENT_ID of the employee 170 is modified to 999. Because department 999 does not exist in the DEPARTMENTS table, the statement raises exception –2291 for the integrity constraint violation.

The EMPLOYEE_DEPT_FK_TRG trigger is created and it inserts a new row into the DEPARTMENTS table by using :NEW.DEPARTMENT_ID for the value of the new department's DEPARTMENT_ID. The trigger fires when the UPDATE statement modifies the DEPARTMENT_ID of employee 170 to 999. When the foreign key constraint is checked, it is successful because the trigger inserted the department 999 into the DEPARTMENTS table. Therefore, no exception occurs unless the department already exists when the trigger attempts to insert the new row. However, the EXCEPTION handler traps and masks the exception allowing the operation to succeed.

Note: Although the example shown in the slide is somewhat contrived due to the limited data in the HR schema, the point is that if you defer the constraint check until the commit, you then have the ability to engineer a trigger to detect that constraint failure and repair it prior to the commit action.

INSTEAD OF Triggers



8 - 26

Copyright © 2009, Oracle. All rights reserved.

ORACLE®

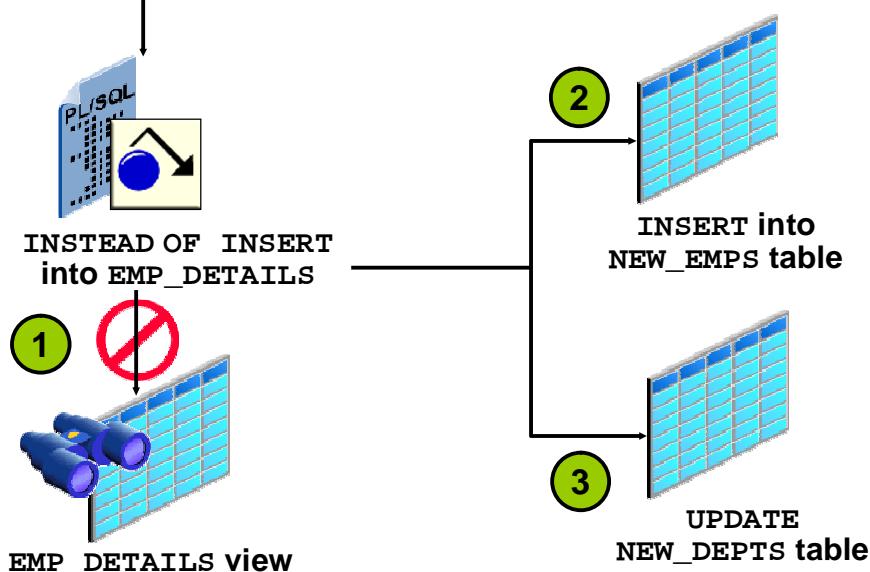
INSTEAD OF Triggers

Use INSTEAD OF triggers to modify data in which the DML statement has been issued against an inherently un-updatable view. These triggers are called INSTEAD OF triggers because, unlike other triggers, the Oracle server fires the trigger instead of executing the triggering statement. These triggers are used to perform INSERT, UPDATE, and DELETE operations directly on the underlying tables. You can write INSERT, UPDATE, and DELETE statements against a view, and the INSTEAD OF trigger works invisibly in the background to make the right actions take place. A view cannot be modified by normal DML statements if the view query contains set operators, group functions, clauses such as GROUP BY, CONNECT BY, START, the DISTINCT operator, or joins. For example, if a view consists of more than one table, an insert to the view may entail an insertion into one table and an update to another. So you write an INSTEAD OF trigger that fires when you write an insert against the view. Instead of the original insertion, the trigger body executes, which results in an insertion of data into one table and an update to another table.

Note: If a view is inherently updatable and has INSTEAD OF triggers, then the triggers take precedence. INSTEAD OF triggers are row triggers. The CHECK option for views is not enforced when insertions or updates to the view are performed by using INSTEAD OF triggers. The INSTEAD OF trigger body must enforce the check.

Creating an INSTEAD OF Trigger: Example

```
INSERT INTO emp_details  
VALUES (9001,'ABBOTT',3000, 10, 'Administration');
```



Creating an INSTEAD OF Trigger

You can create an INSTEAD OF trigger in order to maintain the base tables on which a view is based.

The example in the slide illustrates an employee being inserted into the view `EMP_DETAILS`, whose query is based on the `EMPLOYEES` and `DEPARTMENTS` tables. The `NEW_EMP_DEPT` (INSTEAD OF) trigger executes in place of the `INSERT` operation that causes the trigger to fire. The INSTEAD OF trigger then issues the appropriate `INSERT` and `UPDATE` to the base tables used by the `EMP_DETAILS` view. Therefore, instead of inserting the new employee record into the `EMPLOYEES` table, the following actions take place:

1. The `NEW_EMP_DEPT` INSTEAD OF trigger fires.
2. A row is inserted into the `NEW_EMPS` table.
3. The `DEPT_SAL` column of the `NEW_DEPTS` table is updated. The salary value supplied for the new employee is added to the existing total salary of the department to which the new employee has been assigned.

Note: Before you run the example in the slide, you must create the required structures shown on the next two pages.

Creating an INSTEAD OF Trigger to Perform DML on Complex Views

```
CREATE TABLE new_emps AS
  SELECT employee_id, last_name, salary, department_id
    FROM employees;

CREATE TABLE new_depts AS
  SELECT d.department_id, d.department_name,
         sum(e.salary) dept_sal
    FROM employees e, departments d
   WHERE e.department_id = d.department_id;

CREATE VIEW emp_details AS
  SELECT e.employee_id, e.last_name, e.salary,
         e.department_id, d.department_name
    FROM employees e, departments d
   WHERE e.department_id = d.department_id
  GROUP BY d.department_id, d.department_name;
```

ORACLE®

8 - 28

Copyright © 2009, Oracle. All rights reserved.

Creating an INSTEAD OF Trigger (continued)

The example in the slide creates two new tables, NEW_EMPS and NEW_DEPTS, that are based on the EMPLOYEES and DEPARTMENTS tables, respectively. It also creates an EMP_DETAILS view from the EMPLOYEES and DEPARTMENTS tables.

If a view has a complex query structure, then it is not always possible to perform DML directly on the view to affect the underlying tables. The example requires creation of an INSTEAD OF trigger, called NEW_EMP_DEPT, shown on the next page. The NEW_DEPT_EMP trigger handles DML in the following way:

- When a row is inserted into the EMP_DETAILS view, instead of inserting the row directly into the view, rows are added into the NEW_EMPS and NEW_DEPTS tables, using the data values supplied with the INSERT statement.
- When a row is modified or deleted through the EMP_DETAILS view, corresponding rows in the NEW_EMPS and NEW_DEPTS tables are affected.

Note: INSTEAD OF triggers can be written only for views, and the BEFORE and AFTER timing options are not valid.

Creating an INSTEAD OF Trigger (continued)

```
CREATE OR REPLACE TRIGGER new_emp_dept
INSTEAD OF INSERT OR UPDATE OR DELETE ON emp_details
FOR EACH ROW
BEGIN
    IF INSERTING THEN
        INSERT INTO new_emps
        VALUES (:NEW.employee_id, :NEW.last_name,
                :NEW.salary, :NEW.department_id);
        UPDATE new_depts
        SET dept_sal = dept_sal + :NEW.salary
        WHERE department_id = :NEW.department_id;
    ELSIF DELETING THEN
        DELETE FROM new_emps
        WHERE employee_id = :OLD.employee_id;
        UPDATE new_depts
        SET dept_sal = dept_sal - :OLD.salary
        WHERE department_id = :OLD.department_id;
    ELSIF UPDATING ('salary') THEN
        UPDATE new_emps
        SET salary = :NEW.salary
        WHERE employee_id = :OLD.employee_id;
        UPDATE new_depts
        SET dept_sal = dept_sal +
                      (:NEW.salary - :OLD.salary)
        WHERE department_id = :OLD.department_id;
    ELSIF UPDATING ('department_id') THEN
        UPDATE new_emps
        SET department_id = :NEW.department_id
        WHERE employee_id = :OLD.employee_id;
        UPDATE new_depts
        SET dept_sal = dept_sal - :OLD.salary
        WHERE department_id = :OLD.department_id;
        UPDATE new_depts
        SET dept_sal = dept_sal + :NEW.salary
        WHERE department_id = :NEW.department_id;
    END IF;
END;
/
```

DEPARTMENT_ID	DEPARTMENT_NAME	DEPT_SAL
10	Administration	7400

1 rows selected

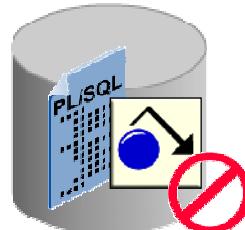
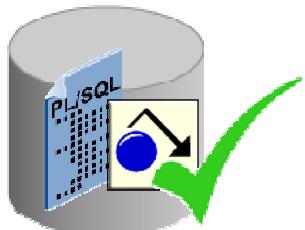
EMPLOYEE_ID	LAST_NAME	SALARY	DEPARTMENT_ID
200	Whalen	4400	10
9001	ABBOTT	3000	10

2 rows selected

The Status of a Trigger

A trigger is in either of two distinct modes:

- Enabled: The trigger runs its trigger action if a triggering statement is issued and the trigger restriction (if any) evaluates to true (default).
- Disabled: The trigger does not run its trigger action, even if a triggering statement is issued and the trigger restriction (if any) would evaluate to true.



ORACLE®

Creating a Disabled Trigger

- Before Oracle Database 11g, if you created a trigger whose body had a PL/SQL compilation error, then DML to the table failed.
- In Oracle Database 11g, you can create a disabled trigger and then enable it only when you know it will be compiled successfully.

```
CREATE OR REPLACE TRIGGER mytrg
  BEFORE INSERT ON mytable FOR EACH ROW
  DISABLE
BEGIN
  :New.ID := my_seq.Nextval;
  . . .
END;
/
```

ORACLE®

8 - 31

Copyright © 2009, Oracle. All rights reserved.

Creating a Disabled Trigger

Before Oracle Database 11g, if you created a trigger whose body had a PL/SQL compilation error, then DML to the table failed. The following error message was displayed:

ORA-04098: trigger 'TRG' is invalid and failed re-validation

In Oracle Database 11g, you can create a disabled trigger, and then enable it only when you know it will be compiled successfully.

You can also temporarily disable a trigger in the following situations:

- An object it references is not available.
- You need to perform a large data load, and you want it to proceed quickly without firing triggers.
- You are reloading data.

Note: The code example in the slide assumes that you have an existing sequence named my_seq.

Managing Triggers Using the ALTER and DROP SQL Statements

```
-- Disable or reenable a database trigger:
```

```
ALTER TRIGGER trigger_name DISABLE | ENABLE;
```

```
-- Disable or reenable all triggers for a table:
```

```
ALTER TABLE table_name DISABLE | ENABLE ALL TRIGGERS;
```

```
-- Recompile a trigger for a table:
```

```
ALTER TRIGGER trigger_name COMPILE;
```

```
-- Remove a trigger from the database:
```

```
DROP TRIGGER trigger_name;
```

ORACLE®

8 - 32

Copyright © 2009, Oracle. All rights reserved.

Managing Triggers

A trigger has two modes or states: ENABLED and DISABLED. When a trigger is first created, it is enabled by default. The Oracle server checks integrity constraints for enabled triggers and guarantees that triggers cannot compromise them. In addition, the Oracle server provides read-consistent views for queries and constraints, manages the dependencies, and provides a two-phase commit process if a trigger updates remote tables in a distributed database.

Disabling a Trigger

Use the ALTER TRIGGER command to disable a trigger. You can also disable all triggers on a table by using the ALTER TABLE command. You can disable triggers to improve performance or to avoid data integrity checks when loading massive amounts of data with utilities such as SQL*Loader. You might also disable a trigger when it references a database object that is currently unavailable, due to a failed network connection, disk crash, offline data file, or offline tablespace.

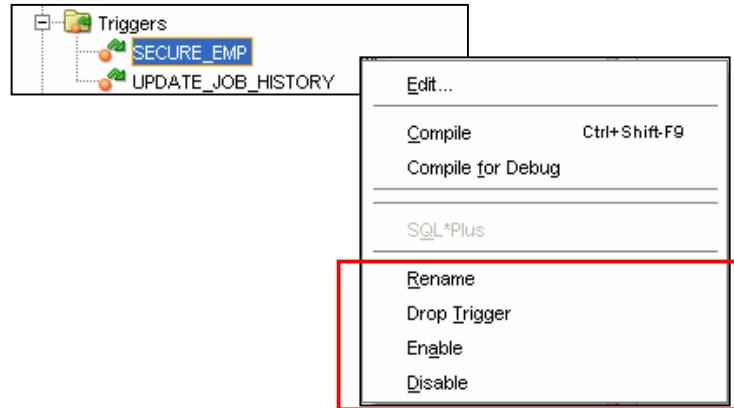
Recompiling a Trigger

Use the ALTER TRIGGER command to explicitly recompile a trigger that is invalid.

Removing Triggers

When a trigger is no longer required, use a SQL statement in SQL Developer or SQL*Plus to remove it. When you remove a table, all triggers on that table are also removed.

Managing Triggers Using SQL Developer



ORACLE®

Managing Triggers Using SQL Developer

You can use the Triggers node in the Connections navigation tree to manage triggers. Right-click a trigger name, and then select one of the following options:

- Edit
- Compile
- Compile for Debug
- Rename
- Drop Trigger
- Enable
- Disable

Testing Triggers

- Test each triggering data operation, as well as non-triggering data operations.
- Test each case of the WHEN clause.
- Cause the trigger to fire directly from a basic data operation, as well as indirectly from a procedure.
- Test the effect of the trigger on other triggers.
- Test the effect of other triggers on the trigger.

ORACLE®

8 - 34

Copyright © 2009, Oracle. All rights reserved.

Testing Triggers

Testing code can be a time-consuming process. Do the following when testing triggers:

- Ensure that the trigger works properly by testing a number of cases separately:
 - Test the most common success scenarios first.
 - Test the most common failure conditions to see that they are properly managed.
- The more complex the trigger, the more detailed your testing is likely to be. For example, if you have a row trigger with a WHEN clause specified, then you should ensure that the trigger fires when the conditions are satisfied. Or, if you have cascading triggers, you need to test the effect of one trigger on the other and ensure that you end up with the desired results.
- Use the DBMS_OUTPUT package to debug triggers.

Viewing Trigger Information

You can view the following trigger information:

Data Dictionary View	Description
USER_OBJECTS	Displays object information
USER/ALL/DBA_TRIGGERS	Displays trigger information
USER_ERRORS	Displays PL/SQL syntax errors for a trigger



Viewing Trigger Information

The slide shows the data dictionary views that you can access to get information regarding the triggers.

The USER_OBJECTS view contains the name and status of the trigger and the date and time when the trigger was created.

The USER_ERRORS view contains the details about the compilation errors that occurred while a trigger was compiling. The contents of these views are similar to those for subprograms.

The USER_TRIGGERS view contains details such as name, type, triggering event, the table on which the trigger is created, and the body of the trigger.

The `SELECT Username FROM USER_USERS;` statement gives the name of the owner of the trigger, not the name of the user who is updating the table.

Using USER_TRIGGERS

DESCRIBE user_triggers

Name	Null	Type
TRIGGER_NAME		VARCHAR2(30)
TRIGGER_TYPE		VARCHAR2(16)
TRIGGERING_EVENT		VARCHAR2(227)
TABLE_OWNER		VARCHAR2(30)
BASE_OBJECT_TYPE		VARCHAR2(16)
TABLE_NAME		VARCHAR2(30)
COLUMN_NAME		VARCHAR2(4000)
REFERENCING_NAMES		VARCHAR2(128)
WHEN_CLAUSE		VARCHAR2(4000)
STATUS		VARCHAR2(8)
DESCRIPTION		VARCHAR2(4000)
ACTION_TYPE		VARCHAR2(11)
TRIGGER_BODY		LONG
CROSSEDITON		VARCHAR2(7)
BEFORE_STATEMENT		VARCHAR2(3)
BEFORE_ROW		VARCHAR2(3)
AFTER_ROW		VARCHAR2(3)
AFTER_STATEMENT		VARCHAR2(3)
INSTEAD_OF_ROW		VARCHAR2(3)
FIRE_ONCE		VARCHAR2(3)
APPLY_SERVER_ONLY		VARCHAR2(3)

21 rows selected

```
SELECT trigger_type, trigger_body
FROM user_triggers
WHERE trigger_name = 'SECURE_EMP';
```

ORACLE®

8 - 36

Copyright © 2009, Oracle. All rights reserved.

Using USER_TRIGGERS

If the source file is unavailable, then you can use the SQL Worksheet in SQL Developer or SQL*Plus to regenerate it from USER_TRIGGERS. You can also examine the ALL_TRIGGERS and DBA_TRIGGERS views, each of which contains the additional column OWNER, for the owner of the object. The result for the second example in the slide is as follows:

```
TRIGGER_TYPE      TRIGGER_BODY
-----
BEFORE STATEMENT BEGIN
    IF (TO_CHAR(SYSDATE,'DY') IN ('SAT','SUN')) OR
       (TO_CHAR(SYSDATE,'HH24') NOT BETWEEN '08' AND '18') THEN
        IF DELETING THEN
            RAISE_APPLICATION_ERROR(-20502,
            'You may delete from EMPLOYEES table only during normal business hours.');
        ELSIF INSERTING THEN
            RAISE_APPLICATION_ERROR(-20500,
            'You may insert into EMPLOYEES table only during normal business hours.');
        ELSIF UPDATING('SALARY') THEN
            RAISE_APPLICATION_ERROR(-20503,
            'You may update SALARY only during normal business hours.');
        ELSE
            RAISE_APPLICATION_ERROR(-20504,
            'You may update EMPLOYEES table only during normal business hours.');
        END IF;
    END IF;
END;
```

Quiz

A triggering event can be one or more of the following:

1. An INSERT, UPDATE, or DELETE statement on a specific table (or view, in some cases)
2. A CREATE, ALTER, or DROP statement on any schema object
3. A database startup or instance shutdown
4. A specific error message or any error message
5. A user logon or logoff



Answers: 1, 2, 3, 4, 5

Summary

In this lesson, you should have learned how to:

- Create database triggers that are invoked by DML operations
- Create statement and row trigger types
- Use database trigger-firing rules
- Enable, disable, and manage database triggers
- Develop a strategy for testing triggers
- Remove database triggers



Summary

This lesson covered creating database triggers that execute before, after, or instead of a specified DML operation. Triggers are associated with database tables or views. The BEFORE and AFTER timings apply to DML operations on tables. The INSTEAD OF trigger is used as a way to replace DML operations on a view with appropriate DML statements against other tables in the database.

Triggers are enabled by default but can be disabled to suppress their operation until enabled again. If business rules change, triggers can be removed or altered as required.

Practice 8 Overview: Creating Statement and Row Triggers

This practice covers the following topics:

- Creating row triggers
- Creating a statement trigger
- Calling procedures from a trigger



Practice 8: Overview

In this practice, you create statement and row triggers. You also create procedures that are invoked from within the triggers.



Creating Compound, DDL, and Event Database Triggers

ORACLE®

Copyright © 2009, Oracle. All rights reserved.

Objectives

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

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



Lesson Aim

In this lesson, you learn how to create and use database triggers.

What Is a Compound Trigger?

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

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



What Is a Compound Trigger?

Starting with Oracle Database 11g, you can use a compound trigger. A compound trigger is a single trigger on a table that allows you to specify actions for each of the four triggering timing points:

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

Note: For additional information about triggers, refer to the *Oracle Database PL/SQL Language Reference 11g Release 2 (11.2)*.

Working with Compound Triggers

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

ORACLE®

9 - 4

Copyright © 2009, Oracle. All rights reserved.

Working with Compound Triggers

The compound trigger body supports a common PL/SQL state that the code for each timing point can access. The common state is automatically destroyed when the firing statement completes, even when the firing statement causes an error. Your applications can avoid the mutating table error by allowing rows destined for a second table (such as a history table or an audit table) to accumulate and then bulk-inserting them.

Before Oracle Database 11g Release 1 (11.1), you needed to model the common state with an ancillary package. This approach was both cumbersome to program and subject to memory leak when the firing statement caused an error and the after-statement trigger did not fire. Compound triggers make PL/SQL easier for you to use and improve run-time performance and scalability.

The Benefits of Using a Compound Trigger

You can use compound triggers to:

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



Timing-Point Sections of a Table Compound Trigger

A compound trigger defined on a table has one or more of the following timing-point sections. Timing-point sections must appear in the order shown in the table.

Timing Point	Compound Trigger Section
Before the triggering statement executes	BEFORE statement
After the triggering statement executes	AFTER statement
Before each row that the triggering statement affects	BEFORE EACH ROW
After each row that the triggering statement affects	AFTER EACH ROW

ORACLE®

9 - 6

Copyright © 2009, Oracle. All rights reserved.

Note

Timing-point sections must appear in the order shown in the slide. If a timing-point section is absent, nothing happens at its timing point.

Compound Trigger Structure for Tables

```
CREATE OR REPLACE TRIGGER schema.trigger
FOR dml_event_clause ON schema.table
COMPOUND TRIGGER
```

```
-- Initial section
-- Declarations
-- Subprograms
```

1

```
-- Optional section
BEFORE STATEMENT IS ...;
```

```
-- Optional section
AFTER STATEMENT IS ...;
```

```
-- Optional section
BEFORE EACH ROW IS ...;
```

```
-- Optional section
AFTER EACH ROW IS ...;
```

2

ORACLE®

9 - 7

Copyright © 2009, Oracle. All rights reserved.

Compound Trigger Structure for Tables

A compound trigger has two main sections:

- An initial section where variables and subprograms are declared. The code in this section executes before any of the code in the optional section.
- An optional section that defines the code for each possible trigger point. Depending on whether you are defining a compound trigger for a table or for a view, these triggering points are different and are listed in the image shown above and on the following page. The code for the triggering points must follow the order shown above.

Note: For additional information about Compound Triggers, refer to the *Oracle Database PL/SQL Language Reference 11g Release 2 (11.2)* guide.

Compound Trigger Structure for Views

```
CREATE OR REPLACE TRIGGER
schema.trigger
FOR dml_event_clause ON schema.view
COMPOUND TRIGGER
    -- Initial section
    -- Declarations
    -- Subprograms
    -- Optional section (exclusive)
    INSTEAD OF EACH ROW IS
    ....;
```

ORACLE®

9 - 8

Copyright © 2009, Oracle. All rights reserved.

Compound Trigger Structure for Views

With views, the only allowed section is an INSTEAD OF EACH ROW clause.

Compound Trigger Restrictions

- A compound trigger must be a DML trigger and defined on either a table or a view.
- The body of a compound trigger must be compound trigger block, written in PL/SQL.
- A compound trigger body cannot have an initialization block; therefore, it cannot have an exception section.
- An exception that occurs in one section must be handled in that section. It cannot transfer control to another section.
- `:OLD` and `:NEW` cannot appear in the declaration, BEFORE STATEMENT, or the AFTER STATEMENT sections.
- Only the BEFORE EACH ROW section can change the value of `:NEW`.
- The firing order of compound triggers is not guaranteed unless you use the `FOLLOWS` clause.

ORACLE®

9 - 9

Copyright © 2009, Oracle. All rights reserved.

Compound Trigger Restrictions

The following are some of the restrictions when working with compound triggers:

- The body of a compound trigger must compound trigger block, written in PL/SQL.
- A compound trigger must be a DML trigger.
- A compound trigger must be defined on either a table or a view.
- A compound trigger body cannot have an initialization block; therefore, it cannot have an exception section. This is not a problem, because the BEFORE STATEMENT section always executes exactly once before any other timing-point section executes.
- An exception that occurs in one section must be handled in that section. It cannot transfer control to another section.
- `:OLD`, `:NEW`, and `:PARENT` cannot appear in the declaration section, the BEFORE STATEMENT section, or the AFTER STATEMENT section.
- The firing order of compound triggers is not guaranteed unless you use the `FOLLOWS` clause.

Trigger Restrictions on Mutating Tables

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

ORACLE®

9 - 10

Copyright © 2009, Oracle. All rights reserved.

Rules Governing Triggers

Reading and writing data using triggers is subject to certain rules. The restrictions apply only to row triggers, unless a statement trigger is fired as a result of ON DELETE CASCADE.

Mutating Table

A mutating table is a table that is currently being modified by an UPDATE, DELETE, or INSERT statement, or a table that might need to be updated by the effects of a declarative DELETE CASCADE referential integrity action. For STATEMENT triggers, a table is not considered a mutating table.

A mutating table error (ORA-4091) occurs when a row-level trigger attempts to change or examine a table that is already undergoing change via a DML statement.

The triggered table itself is a mutating table, as well as any table referencing it with the FOREIGN KEY constraint. This restriction prevents a row trigger from seeing an inconsistent set of data.

Mutating Table: Example

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

ORACLE®

9 - 11

Copyright © 2009, Oracle. All rights reserved.

Mutating Table: Example

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

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

Mutating Table: Example

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

```
TRIGGER check_salary Compiled.

Error starting at line 1 in command:
UPDATE employees
SET salary = 3400
WHERE last_name = 'Stiles'
Error report:
SQL Error: ORA-04091: table ORA42.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "ORA42.CHECK_SALARY", line 5
ORA-04088: error during execution of trigger 'ORA42.CHECK_SALARY'
04091. 00000 -  "table *s.*s is mutating, trigger/function may not see it"
*Cause:  A trigger (or a user defined plsql function that is referenced in
this statement) attempted to look at (or modify) a table that was
in the middle of being modified by the statement which fired it.
*Action: Rewrite the trigger (or function) so it does not read that table.
```

ORACLE®

9 - 12

Copyright © 2009, Oracle. All rights reserved.

Mutating Table: Example (continued)

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

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

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

Possible Solutions

Possible solutions to this mutating table problem include the following:

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

Depending on the nature of the problem, a solution can become more convoluted and difficult to solve. In this case, consider implementing the rules in the application or middle tier and avoid using database triggers to perform overly complex business rules. An insert statement in the code example in the slide will not generate a mutating table example.

Using a Compound Trigger to Resolve the Mutating Table Error

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

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

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

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

-- example continues on next slide
```

ORACLE®

9 - 13

Copyright © 2009, Oracle. All rights reserved.

Using a Compound Trigger to Resolve the Mutating Table Error

The CHECK_SALARY compound trigger resolves the mutating table error in the earlier example. This is achieved by storing the values in PL/SQL collections, and then performing a bulk insert/update in the “before statement” section of the compound trigger. In the example in the slide, PL/SQL collections are used. The element types used are based on the SALARY and DEPARTMENT_ID columns from the EMPLOYEES table. To create collections, you define a collection type, and then declare variables of that type. Collections are instantiated when you enter a block or subprogram, and cease to exist when you exit. min_salaries is used to hold the minimum salary for each department and max_salaries is used to hold the maximum salary for each department. department_ids is used to hold the department IDs. If the employee who earns the minimum or maximum salary does not have an assigned department, you use the NVL function to store –1 for the department id instead of NULL. Next, you collect the minimum salary, maximum salary, and the department ID using a bulk insert into the min_salaries, max_salaries, and department_ids respectively grouped by department ID. The select statement returns 13 rows. The values of the department_ids are used as an index for the department_min_salaries and department_max_salaries tables. Therefore, the index for those two tables (VARCHAR2) represents the actual department_ids.

Using a Compound Trigger to Resolve the Mutating Table Error

```
 . . .
BEFORE STATEMENT IS
BEGIN
    SELECT MIN(salary), MAX(salary), NVL(department_id, -1)
    BULK COLLECT INTO min_Salaries, max_salaries, department_ids
    FROM employees
    GROUP BY department_id;
    FOR j IN 1..department_ids.COUNT() LOOP
        department_min_salaries(department_ids(j)) := min_salaries(j);
        department_max_salaries(department_ids(j)) := max_salaries(j);
    END LOOP;
END BEFORE STATEMENT;

AFTER EACH ROW IS
BEGIN
    IF :NEW.salary < department_min_salaries(:NEW.department_id)
    OR :NEW.salary > department_max_salaries(:NEW.department_id) THEN
        RAISE_APPLICATION_ERROR(-20505,'New Salary is out of acceptable
                                range');
    END IF;
END AFTER EACH ROW;
END check_salary;
```

ORACLE®

9 - 14

Copyright © 2009, Oracle. All rights reserved.

Using a Compound Trigger to Resolve the Mutating Table Error (continued)

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

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

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

1 rows updated

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

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

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

Creating Triggers on DDL Statements

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

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



Creating Triggers on DDL Statements

You can specify one or more types of DDL statements that can cause the trigger to fire. You can create triggers for these events on DATABASE or SCHEMA unless otherwise noted. You can also specify BEFORE and AFTER for the timing of the trigger. The Oracle database fires the trigger in the existing user transaction.

You cannot specify as a triggering event any DDL operation performed through a PL/SQL procedure.

The trigger body in the syntax in the slide represents a complete PL/SQL block.

DDL triggers fire only if the object being created is a cluster, function, index, package, procedure, role, sequence, synonym, table, tablespace, trigger, type, view, or user.

Creating Database-Event Triggers

- Triggering user event:
 - CREATE, ALTER, or DROP
 - Logging on or off
- Triggering database or system event:
 - Shutting down or starting up the database
 - A specific error (or any error) being raised

ORACLE®

9 - 16

Copyright © 2009, Oracle. All rights reserved.

Creating Database Triggers

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

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

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

Triggering events that can cause a trigger to fire:

- A data definition statement on an object in the database or schema
- A specific user (or any user) logging on or off
- A database shutdown or startup
- Any error that occurs

Creating Triggers on System Events

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

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

ORACLE®

9 - 17

Copyright © 2009, Oracle. All rights reserved.

Creating Triggers on System Events

You can create triggers for the events listed in the table on DATABASE or SCHEMA, except SHUTDOWN and STARTUP, which apply only to DATABASE.

LOGON and LOGOFF Triggers: Example

```
-- Create the log_trig_table shown in the notes page
-- first

CREATE OR REPLACE TRIGGER logon_trig
AFTER LOGON ON SCHEMA
BEGIN
  INSERT INTO log_trig_table(user_id,log_date,action)
  VALUES (USER, SYSDATE, 'Logging on');
END;
/
```

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

ORACLE®

9 - 18

Copyright © 2009, Oracle. All rights reserved.

LOGON and LOGOFF Triggers: Example

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

The definition of the log_trig_table used in the slide examples is as follows:

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

CALL Statements in Triggers

```
CREATE [OR REPLACE] TRIGGER trigger_name
timing
event1 [OR event2 OR event3]
ON table_name
[REFERENCING OLD AS old | NEW AS new]
[FOR EACH ROW]
[WHEN condition]
CALL procedure_name
/
```

```
CREATE OR REPLACE PROCEDURE log_execution IS
BEGIN
    DBMS_OUTPUT.PUT_LINE('log_execution: Employee Inserted');
END;
/
CREATE OR REPLACE TRIGGER log_employee
BEFORE INSERT ON EMPLOYEES
CALL log_execution -- no semicolon needed
/
```

ORACLE®

9 - 19

Copyright © 2009, Oracle. All rights reserved.

CALL Statements in Triggers

A CALL statement enables you to call a stored procedure, rather than code the PL/SQL body in the trigger itself. The procedure can be implemented in PL/SQL, C, or Java.

The call can reference the trigger attributes :NEW and :OLD as parameters, as in the following example:

```
CREATE OR REPLACE TRIGGER salary_check
    BEFORE UPDATE OF salary, job_id ON employees
    FOR EACH ROW
    WHEN (NEW.job_id <> 'AD_PRES')
    CALL check_salary(:NEW.job_id, :NEW.salary)
```

Note: There is no semicolon at the end of the CALL statement.

In the preceding example, the trigger calls a `check_salary` procedure. The procedure compares the new salary with the salary range for the new job ID from the JOBS table.

Benefits of Database-Event Triggers

- Improved data security:
 - Provides enhanced and complex security checks
 - Provides enhanced and complex auditing
- Improved data integrity:
 - Enforces dynamic data integrity constraints
 - Enforces complex referential integrity constraints
 - Ensures that related operations are performed together implicitly

ORACLE®

9 - 20

Copyright © 2009, Oracle. All rights reserved.

Benefits of Database-Event Triggers

You can use database triggers:

- As alternatives to features provided by the Oracle server
- If your requirements are more complex or more simple than those provided by the Oracle server
- If your requirements are not provided by the Oracle server at all

System Privileges Required to Manage Triggers

The following system privileges are required to manage triggers:

- The privileges that enable you to create, alter, and drop triggers in any schema:
 - GRANT CREATE TRIGGER TO ora61
 - GRANT ALTER ANY TRIGGER TO ora61
 - GRANT DROP ANY TRIGGER TO ora61
- The privilege that enables you to create a trigger on the database:
 - GRANT ADMINISTER DATABASE TRIGGER TO ora61
- The EXECUTE privilege (if your trigger refers to any objects that are not in your schema)



System Privileges Required to Manage Triggers

To create a trigger in your schema, you need the CREATE TRIGGER system privilege, and you must own the table specified in the triggering statement, have the ALTER privilege for the table in the triggering statement, or have the ALTER ANY TABLE system privilege. You can alter or drop your triggers without any further privileges being required.

If the ANY keyword is used, you can create, alter, or drop your own triggers and those in another schema and can be associated with any user's table.

You do not need any privileges to invoke a trigger in your schema. A trigger is invoked by DML statements that you issue. But if your trigger refers to any objects that are not in your schema, the user creating the trigger must have the EXECUTE privilege on the referenced procedures, functions, or packages, and not through roles.

To create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER privilege. If this privilege is later revoked, you can drop the trigger but you cannot alter it.

Note: Similar to stored procedures, statements in the trigger body use the privileges of the trigger owner, not the privileges of the user executing the operation that fires the trigger.

Guidelines for Designing Triggers

- You can design triggers to:
 - Perform related actions
 - Centralize global operations
- You must not design triggers:
 - Where functionality is already built into the Oracle server
 - That duplicate other triggers
- You can create stored procedures and invoke them in a trigger, if the PL/SQL code is very lengthy.
- Excessive use of triggers can result in complex interdependencies, which may be difficult to maintain in large applications.

ORACLE®

9 - 22

Copyright © 2009, Oracle. All rights reserved.

Guidelines for Designing Triggers

- Use triggers to guarantee that related actions are performed for a specific operation and for centralized, global operations that should be fired for the triggering statement, independent of the user or application issuing the statement.
- Do not define triggers to duplicate or replace the functionality already built into the Oracle database. For example, implement integrity rules using declarative constraints instead of triggers. To remember the design order for a business rule:
 - Use built-in constraints in the Oracle server, such as primary key, and so on.
 - Develop a database trigger or an application, such as a servlet or Enterprise JavaBeans (EJB) on your middle tier.
 - Use a presentation interface, such as Oracle Forms, HTML, JavaServer Pages (JSP) and so on, for data presentation rules.
- Excessive use of triggers can result in complex interdependencies, which may be difficult to maintain. Use triggers when necessary, and be aware of recursive and cascading effects.
- Avoid lengthy trigger logic by creating stored procedures or packaged procedures that are invoked in the trigger body.
- Database triggers fire for every user each time the event occurs on the trigger that is created.

Quiz

1. A trigger is defined with a CREATE TRIGGER statement
2. A trigger's source code is contained in the USER_TRIGGERS data dictionary.
3. A trigger is explicitly invoked.
4. A trigger is implicitly invoked by DML.
5. The COMMIT, SAVEPOINT, and ROLLBACK are not allowed when working with a trigger.

ORACLE®

9 - 23

Copyright © 2009, Oracle. All rights reserved.

Answers: 1, 2, 4, 5

Summary

In this lesson, you should have learned how to:

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



Practice 9: Overview

This practice covers the following topics:

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



Practice 9: Overview

In this practice, you implement a simple business rule for ensuring data integrity of employees' salaries with respect to the valid salary range for their job. You create a trigger for this rule.

During this process, your new triggers cause a cascading effect with triggers created in the practice section of the previous lesson. The cascading effect results in a mutating table exception on the JOBS table. You then create a PL/SQL package and additional triggers to solve the mutating table issue.

