**Introduction to Oracle9*i*: PL/SQL**

**Student Guide . Volume 2**

40054GC10

Production 1.0

June 2001

D32946

**Authors**

Nagavalli Pataballa

Priya Nathan

**Technical Contributors and Reviewers**

Anna Atkinson

Bryan Roberts

Caroline Pereda

Cesljas Zarco

Chaya Rao

Coley William

Daniel Gabel

Dr. Christoph Burandt Hakan Lindfors

Helen Robertson

John Hoff

Judy Brink

Lachlan Williams

Laszlo Czinkoczki

Laura Pezzini

Linda Boldt

Marco Verbeek

Natarajan Senthil

Priya Vennapusa

Robert Squires

Roger Abuzalaf

Ruediger Steffan

Sarah Jones

Stefan Lindblad

Sue Onraet

Susan Dee

**Publisher**

Sandya Krishna

**Copyright © Oracle Corporation, 1999, 2000, 2001. All rights reserved.**

This documentation contains proprietary information of Oracle Corporation. It is provided under a license agreement containing restrictions on use and disclosure and is also protected by copyright law. Reverse engineering of the software is prohibited. If this documentation is delivered to a U.S. Government Agency of the Department of Defense, then it is delivered with Restricted Rights and the following legend is applicable:

**Restricted Rights Legend**

Use, duplication or disclosure by the Government is subject to restrictions for commercial computer software and shall be deemed to be Restricted Rights software under Federal law, as set forth in subparagraph (c)(1)(ii) of DFARS 252.227-7013, Rights in Technical Data and Computer Software (October 1988).

This material or any portion of it may not be copied in any form or by any means without the express prior written permission of Oracle Corporation. Any other copying is a violation of copyright law and may result in civil and/or criminal penalties.

If this documentation is delivered to a U.S. Government Agency not within the Department of Defense, then it is delivered with “Restricted Rights,” as defined in FAR 52.227-14, Rights in Data-General, including Alternate III (June 1987).

The information in this document is subject to change without notice. If you find any problems in the documentation, please report them in writing to Education Products, Oracle Corporation, 500 Oracle Parkway, Box SB-6, Redwood Shores, CA 94065. Oracle Corporation does not warrant that this document is error-free.

All references to Oracle and Oracle products are trademarks or registered trademarks of Oracle Corporation.

All other products or company names are used for identification purposes only, and may be trademarks of their respective owners.

**Contents**

**Preface**

**Curriculum Map**

**Introduction**

Course Objectives I-2

About PL/SQL I-3

PL/SQL Environment I-4

Benefits of PL/SQL I-5

Benefits of Subprograms I-10

Invoking Stored Procedures and Functions I-11 Summary I-12

**1 Declaring Variables**

Objectives 1-2

PL/SQL Block Structure 1-3

Executing Statements and PL/SQL Blocks 1-4 Block Types 1-5

Program Constructs 1-6

Use of Variables 1-7

Handling Variables in PL/SQL 1-8

Types of Variables 1-9

Using iSQL\*Plus Variables Within PL/SQL Blocks 1-10 Types of Variables 1-11

Declaring PL/SQL Variables 1-12

Guidelines for Declaring PL/SQL Variables 1-13 Naming Rules 1-14

Variable Initialization and Keywords 1-15

Scalar Data Types 1-17

Base Scalar Data Types 1-18

Scalar Variable Declarations 1-22

The %TYPE Attribute 1-23

Declaring Variables with the %TYPE Attribute 1-24 Declaring Boolean Variables 1-25

Composite Data Types 1-26

LOB Data Type Variables 1-27

Bind Variables 1-28

Using Bind Variables 1-30

Referencing Non-PL/SQL Variables 1-31

DBMS\_OUTPUT.PUT\_LINE 1-32

Summary 1-33

Practice 1 Overview 1-35

**iii**

**2 Writing Executable Statements** Objectives 2-2

PL/SQL Block Syntax and Guidelines 2-3 Identifiers 2-5

PL/SQL Block Syntax and Guidelines 2-6 Commenting Code 2-7

SQL Functions in PL/SQL 2-8

SQL Functions in PL/SQL: Examples 2-9 Data type Conversion 2-10

Nested Blocks and Variable Scope 2-12 Identifier Scope 2-14

Qualify an Identifier 2-15

Determining Variable Scope 2-16

Operators in PL/SQL 2-17

Programming Guidelines 2-19

Indenting Code 2-20

Summary 2-21

Practice 2 Overview 2-22

**3 Interacting with the Oracle Server** Objectives 3-2

SQL Statements in PL/SQL 3-3

SELECT Statements in PL/SQL 3-4 Retrieving Data in PL/SQL 3-7

Naming Conventions 3-9

Manipulating Data Using PL/SQL 3-10 Inserting Data 3-11

Updating Data 3-12

Deleting Data 3-13

Merging Rows 3-13

Naming Conventions 3-16

SQL Cursor 3-18

SQL Cursor Attributes 3-19

Transaction Control Statements 3-21 Summary 3-22

Practice 3 Overview 3-24

**iv**

**4 Writing Control Structures**

Objectives 4-2

Controlling PL/SQL Flow of Execution 4-3 IF Statements 4-4

Simple IF Statements 4-5

Compound IF Statements 4-6

IF-THEN-ELSE Statement Execution Flow 4-7 IF-THEN-ELSE Statements 4-8

CASE Expressions 4-12

CASE Expressions: Example 4-13

Handling Nulls 4-15

Logic Tables 4-16

Boolean Conditions 4-17

Iterative Control: LOOP Statements 4-18 Basic Loops 4-19

WHILE Loops 4-21

FOR Loops 4-23

Guidelines While Using Loops 4-26

Nested Loops and Labels 4-27

Summary 4-29

Practice 4 Overview 4-30

**5 Working with Composite Data Types** Objectives 5-2

Composite Data Types 5-3

PL/SQL Records 5-4

Creating a PL/SQL Record 5-5

PL/SQL Record Structure 5-7

The %ROWTYPE Attribute 5-8

Advantages of Using %ROWTYPE 5-10 The %ROWTYPE Attribute 5-11

INDEX BY Tables 5-13

Creating an INDEX by Table 5-14

INDEX BY Table Structure 5-15

Creating an INDEX BY Table 5-16

Using INDEX BY Table Methods 5-17 INDEX BY Table of Records 5-18

Example of PL/SQL Table of Records 5-19 Summary 5-20

Practice 5 Overview 5-21

**v**

**6 Writing Explicit Cursors**

Objectives 6-2

About Cursors 6-3

Explicit Cursor Functions 6-4

Controlling Explicit Cursors 6-5

Declaring the Cursor 6-7

Opening the Cursor 6-9

Fetching Data from the Cursor 6-10

Closing the Cursor 6-12

Explicit Cursor Attributes 6-13

The %ISOPEN Attribute 6-14

Controlling Multiple Fetches 6-15

The %NOTFOUND and %ROWCOUNT Attributes 6-16 Example 6-18

Cursors and Records 6-19

Cursor FOR Loops 6-20

Cursor FOR Loops Using Subqueries 6-22

Summary 6-24

Practice 6 Overview 6-25

**7 Advanced Explicit Cursor Concepts**

Objectives 7-2

Cursors with Parameters 7-3

The FOR UPDATE Clause 7-5

The WHERE CURRENT OF Clause 7-7

Cursors with Subqueries 7-9

Summary 7-10

Practice 7 Overview 7-11

**vi**

**8 Handling Exceptions**

Objectives 8-2

Handling Exceptions with PL/SQL 8-3

Handling Exceptions 8-4

Exception Types 8-5

Trapping Exceptions 8-6

Trapping Exceptions Guidelines 8-7

Trapping Predefined Oracle Server Errors 8-8 Predefined Exceptions 8-11

Trapping Nonpredefined Oracle Server Errors 8-12 Nonpredefined Error 8-13

Functions for Trapping Exceptions 8-14

Trapping User-Defined Exceptions 8-16

User-Defined Exception 8-17

Calling Environments 8-18

Propagating Exceptions 8-19

RAISE\_APPLICATION\_ERROR Procedure 8-20 RAISE\_APPLICATION\_ERROR 8-22

Summary 8-23

Practice 8 Overview 8-23

**9 Creating Procedures**

Objectives 9-2

PL/SQL Program Constructs 9-4

Overview of Subprograms 9-5

Block Structure for Anonymous PL/SQL Blocks 9-6 Block Structure for PL/SQL Subprograms 9-7 PL/SQL Subprograms 9-8

Developing Subprograms by Using iSQL\*Plus 9-9 What Is a Procedure? 9-11

Syntax for Creating Procedures 9-12

Developing Procedures 9-13

Formal Versus Actual Parameters 9-14

Procedural Parameter Modes 9-15

Creating Procedures with Parameters 9-16 **vii**

IN Parameters: Example 9-17

OUT Parameters: Example 9-18

Viewing OUT Parameters 9-20

IN OUT Parameters 9-21

Viewing IN OUT Parameters 9-22

Methods for Passing Parameters 9-23

DEFAULT Option for Parameters 9-24

Examples of Passing Parameters 9-25

Declaring Subprograms 9-26

Invoking a Procedure from an Anonymous PL/SQL Block 9-27 Invoking a Procedure from Another Procedure 9-28

Handled Exceptions 9-29

Unhandled Exceptions 9-31

Removing Procedures 9-33

Benefits of Subprograms 9-34

Summary 9-35

Practice 9 Overview 9-37

**10 Creating Functions**

Objectives 10-2

Overview of Stored Functions 10-3

Syntax for Creating Functions 10-4

Creating a Function 10-5

Creating a Stored Function by Using iSQL\*Plus 10-6 Creating a Stored Function by Using iSQL\*Plus: Example 10-7 Executing Functions 10-8

Executing Functions: Example 10-9

Advantages of User-Defined Functions in SQL Expressions 10-10 Invoking Functions in SQL Expressions: Example 10-11 Locations to Call User-Defined Functions 10-12

Restrictions on Calling Functions from SQL Expressions 10-13 Restrictions on Calling from SQL 10-15

Removing Functions 10-16

Procedure or Function? 10-17

Comparing Procedures and Functions 10-18

Benefits of Stored Procedures and Functions 10-19

Summary 10-20

Practice 10 Overview 10-21

**viii**

**11 Managing Subprograms**

Objectives 11-2

Required Privileges 11-3

Granting Access to Data 11-4

Using Invoker’s-Rights 11-5

Managing Stored PL/SQL Objects 11-6

USER\_OBJECTS 11-7

List All Procedures and Functions 11-8

USER\_SOURCE Data Dictionary View 11-9

List the Code of Procedures and Functions 11-10

USER\_ERRORS 11-11

Detecting Compilation Errors: Example 11-12

List Compilation Errors by Using USER\_ERRORS 11-13 List Compilation Errors by Using SHOW ERRORS 11-14 DESCRIBE in iSQL\*Plus 11-15

Debugging PL/SQL Program Units 11-16

Summary 11-17

Practice 11 Overview 11-19

**12 Creating Packages**

Objectives 12-2

Overview of Packages 12-3

Components of a Package 12-4

Referencing Package Objects 12-5

Developing a Package 12-6

Creating the Package Specification 12-8

Declaring Public Constructs 12-9

Creating a Package Specification: Example 12-10

Creating the Package Body 12-11

Public and Private Constructs 12-12

Creating a Package Body: Example 12-13

Invoking Package Constructs 12-15

Declaring a Bodiless Package 12-17

Referencing a Public Variable from a Stand-alone Procedure 12-18 Removing Packages 12-19

Guidelines for Developing Packages 12-20

Advantages of Packages 12-21

Summary 12-23

Practice 12 Overview 12-26

**ix**

**13 More Package Concepts**

Objectives 13-2

Overloading 13-3

Overloading: Example 13-4

Using Forward Declarations 13-7

Creating a One-Time-Only Procedure 13-9

Restrictions on Package Functions Used in SQL 13-10

User Defined Package: taxes\_pack 13-11

Invoking a User Defined Package Function from a SQL Statement 13-12 Persistent State of Package Variables: Example 13-13

Persistent State of Package Variables 13-14

Controlling the Persistent State of a Package Cursor 13-15 Executing PACK\_CUR 13-17

PL/SQL Tables and Records in Packages 13-18

Summary 13-19

Practice 13 Overview 13-20

**14 Oracle Supplied Packages**

Objectives 14-2

Using Supplied Packages 14-3

Using Native Dynamic SQL 14-4

Execution Flow 14-5

Using the DBMS\_SQL Package 14-6

Using DBMS\_SQL 14-8

Using the EXECUTE IMMEDIATE Statement 14-9

Dynamic SQL Using EXECUTE IMMEDIATE 14-11

Using the DBMS\_DDL Package 14-12

Using DBMS\_JOB for Scheduling 14-13

DBMS\_JOB Subprograms 14-14

Submitting Jobs 14-15

Changing Job Characteristics 14-17

Running, Removing, and Breaking Jobs 14-18

Viewing Information on Submitted Jobs 14-19

Using the DBMS\_OUTPUT Package 14-20

Interacting with Operating System Files 14-21

What Is the UTL\_FILE Package? 14-22

File Processing Using UTL\_FILE 14-23

UTL\_FILE Procedures and Functions 14-24

Exceptions Specific to the UTL\_FILE Package 14-25

The FOPEN and IS\_OPEN Functions 14-26

Using UTL\_FILE 14-27

**x**

UTL\_HTTP Package 14-29

Using the UTL\_HTTP Package 14-30

Using the UTL\_TCP Package 14-31

Oracle-Supplied Packages 14-32

Summary 14-33

Practice 14 Overview 14-34

**15 Manipulating Large Objects**

Objectives 15-2

What Is a LOB? 15-3

Contrasting LONG and LOB Data Types 15-4 Anatomy of a LOB 15-5

Internal LOBs 15-6

Managing Internal LOBs 15-7

What Are BFILEs? 15-8

Securing BFILEs 15-9

A New Database Object: DIRECTORY 15-10 Guidelines for Creating DIRECTORY Objects 15-11 Managing BFILEs 15-12

Preparing to Use BFILEs 15-13

The BFILENAME Function 15-14

Loading BFILEs 15-15

Migrating from LONG to LOB 15-17

The DBMS\_LOB Package 15-19

DBMS\_LOB.READ and DBMS\_LOB.WRITE 15-22 Adding LOB Columns to a Table 15-23

Populating LOB Columns 15-24

Updating LOBs by Using SQL 15-26

Updating LOBs by Using DBMS\_LOB in PL/SQL 15-27 Selecting CLOB Values by Using SQL 15-28 Selecting CLOB Values, Using DBMS\_LOB 15-29 Selecting CLOB Values in PL/SQL 15-30

Removing LOBs 15-31

Temporary LOBs 15-32

Creating a Temporary LOB 15-33

Summary 15-34

Practice 15 Overview 15-36

**xi**

**16 Creating Database Triggers**

Objectives 16-2

Types of Triggers 16-3

Guidelines for Designing Triggers 16-4

Database Trigger: Example 16-5

Creating DML Triggers 16-6

DML Trigger Components 16-7

Firing Sequence 16-11

Syntax for Creating DML Statement Triggers 16-13

Creating DML Statement Triggers 16-14

Testing SECURE\_EMP 16-15

Using Conditional Predicates 16-16

Creating a DML Row Trigger 16-17

Creating DML Row Triggers 16-18

Using OLD and NEW Qualifiers 16-19

Using OLD and NEW Qualifiers: Example Using Audit\_Emp\_Table 16-20 Restricting a Row Trigger 16-21

INSTEAD OF Trigger 16-22

Creating an INSTEAD OF Trigger 16-23

Differentiating between Database Triggers and Stored Procedures 16-27 Differentiating between Database Triggers and Form Builder Triggers 16-28 Managing Triggers 16-29

DROP TRIGGER Syntax 16-30

Trigger Test Cases 16-31

Trigger Execution Model and Constraint Checking 16-32

Trigger Execution Model and Constraint Checking: Example 16-33 A Sample Demonstration for Triggers Using Package Constructs 16-34 After Row and After Statement Triggers 16-35

Demonstration: VAR\_PACK Package Specification 16-36

Demonstration: Using the AUDIC\_EMP Procuedure 16-38

Summary 16-39

Practice 16 Overview 16-40

**xii**

**17 More Trigger Concepts**

Objectives 17-2

Creating Database Triggers 17-3

Creating Triggers on DDL Statements 17-4

Creating Triggers on System Events 17-5

LOGON and LOGOFF Trigger Example 17-6 CALL Statement 17-7

Reading Data from a Mutating Table 17-8

Mutating Table: Example 17-9

Implementating Triggers 17-11

Controlling Security within the Server 17-12

Controlling Security with a Database Trigger 17-13 Using the Server Facility to Audit Data Operations 17-14 Auditing by Using a Trigger 17-15

Enforcing Data Integrity within the Server 17-16 Protecting Data Integrity with a Trigger 17-17 Enforcing Referential Integrity within the Server 17-18 Protecting Referential Integrity with a Trigger 17-19 Replicating a Table within the Server 17-20

Replicating a Table with a Trigger 17-21

Computing Derived Data within the Server 17-22 Computing Derived Values with a Trigger 17-23 Logging Events with a Trigger 17-24

Benefits of Database Triggers 17-26

Managing Triggers 17-27

Viewing Trigger Information 17-28

Using USER\_TRIGGERS 17-29

Listing the Code of Triggers 17-30

Summary 17-31

Practice 17 Overview 17-32

**xiii**

**18 Managing Dependencies**

Objectives 18-2

Understanding Dependencies 18-3

Dependencies 18-4

Local Dependencies 18-5

A Scenario of Local Dependencies 18-6

Displaying Direct Dependencies by Using USER\_DEPENDENCIES 18-7 Displaying Direct and Indirect Dependencies 18-8

Displaying Dependencies 18-9

Another Scenario of Local Dependencies 18-10

A Scenario of Local Naming Dependencies 18-11

Understanding Remote Dependencies 18-12

Concepts of Remote Dependencies 18-13

REMOTE\_DEPENDENCIES\_MODE Parameter 18-14

Remote Dependencies and Time stamp Mode 18-15

Remote Procedure B Compiles at 8:00 a.m. 18-16

Local Procedure A Compiles at 9:00 a.m. 18-17

Execute Procedure A 18-18

Remote Procedure B Recompiled at 11:00 a.m. 18-19

Execute Procudre A 18-20

Signature Mode 18-21

Recompiling a PL/SQL Program Unit 18-22

Unsuccessful Recompilation 18-23

Successful Recompilation 18-24

Recompilation of Procedures 18-25

Packages and Dependencies 18-26

Summary 18-28

Practice 18 Overview 18-29

**A Practice Solutions**

**B Table Descriptions and Data**

**C Creating Program Units by Using Procedure Builder D REF Cursors**

**xiv**

**12**

**Creating Packages**

****Copyright © Oracle Corporation, 2001. All rights reserved.

**Objectives**

**After completing this lesson, you should be able to**

**do the following:**

• **Describe packages and list their possible**

**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 a use for a bodiless package**

**12-2** Copyright © Oracle Corporation, 2001. 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.

**Introduction to Oracle9*i*: PL/SQL 12-2**

**Overview of Packages**

**Packages:**

• **Group logically related PL/SQL types, items, and**

**subprograms**

• **Consist of two parts:**

– **Specification**

– **Body**

• **Cannot be invoked, parameterized, or nested**

• **Allow the Oracle server to read multiple objects**

**into memory at once**

**12-3** Copyright © Oracle Corporation, 2001. All rights reserved.

**Packages Overview**

Packages bundle related PL/SQL types, items, 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 has a specification and a body, stored separately in the database.

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

The body fully defines cursors and subprograms, and so implements the specification.

The package itself cannot be called, parameterized, or nested. Still, the format of a package is similar to that of a subprogram. Once written and compiled, the contents can be shared by many applications.

When you call a packaged PL/SQL construct for the first time, the whole package is loaded into memory. Thus, later calls to constructs in the same package require no disk input/output (I/O).

**Introduction to Oracle9*i*: PL/SQL 12-3**

**Package**

**Components of a Package Public variable**

**specification**

**Package**

**body**

**Procedure A declaration**

**Procedure B definition**

**Procedure A definition**

**Public procedure**

**Private variable Private procedure**

**Public procedure Local variable**

**12-4** Copyright © Oracle Corporation, 2001. All rights reserved.

**Package Development**

You create a package in two parts: first the package specification, and then the package body. Public package constructs are those that are declared in the package specification and defined in the package body. Private package constructs are those that are defined solely within the package body.

|  |  |  |
| --- | --- | --- |
| **Scope of the Construct** | **Description** | **Placement within the Package** |
| Public | Can be referenced from any Oracle server environment | Declared within the package specification and may be defined within the package body |
| Private | Can be referenced only by other constructs which are part of the same package | Declared and defined within the package body |

**Note:** The Oracle server stores the specification and body of a package separately in the database. This enables you to change the definition of a program construct in the package body without causing the Oracle server to invalidate other schema objects that call or reference the program construct.

**Introduction to Oracle9*i*: PL/SQL 12-4**

**Referencing Package Objects Package**

**specification**

**Package**

**body**

**Procedure A declaration**

**Procedure B definition**

**Procedure A definition**

**12-5** Copyright © Oracle Corporation, 2001. All rights reserved.

**Package Development (continued)**

|  |  |
| --- | --- |
| **Visibility of the Construct** | **Description** |
| Local | A variable defined within a subprogram that is not visible to external users.  Private (local to the package) variable: You can define variables in a package body. These variables can be accessed only by other objects in the same package. They are not visible to any subprograms or objects outside of the package. |
| Global | A variable or subprogram that can be referenced (and changed) outside the package and is visible to external users. Global package items must be declared in the package specification. |

**Introduction to Oracle9*i*: PL/SQL 12-5**

**Developing a Package Editor**

**Code**

**1**

***i*SQL\*Plus**

**2 Load and run the file.sql**

**Oracle**

**Source code Compile**

**P code**

**Execute**

**12-6** Copyright © Oracle Corporation, 2001. All rights reserved.

**How to Develop a Package**

1. Write the syntax: Enter the code in a text editor and save it as a SQL script file. 2. Compile the code: Run the SQL script file to generate and compile the source code. The source code is compiled into P code*.*

**Introduction to Oracle9*i*: PL/SQL 12-6**

**Developing a Package**

• **Saving the text of the CREATE PACKAGE statement in two different SQL files facilitates later**

**modifications to the package.**

• **A package specification can exist without a**

**package body, but a package body cannot exist**

**without a package specification.**

**12-7** Copyright © Oracle Corporation, 2001. All rights reserved.

**How to Develop a Package**

There are three basic steps to developing a package, similar to those steps that are used to develop a stand-alone procedure.

1. Write the text of the CREATE PACKAGE statement within a SQL script file to create the package specification and run the script file. The source code is compiled into P code and is stored within the data dictionary.

2. Write the text of the CREATE PACKAGE BODY statement within a SQL script file to create the package body and run the script file.

The source code is compiled into P code and is also stored within the data dictionary. 3. Invoke any public construct within the package from an Oracle server environment.

**Introduction to Oracle9*i*: PL/SQL 12-7**

**Creating the Package Specification**

**Syntax:**

**CREATE [OR REPLACE] PACKAGE package\_name**

**IS|AS**

**public type and item declarations**

**subprogram specifications**

**END package\_name;**

• **The REPLACE option drops and recreates 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.**

**12-8** Copyright © Oracle Corporation, 2001. All rights reserved.

**How to Create a Package Specification**

To create packages, you declare all public constructs within the package specification. • Specify the REPLACE option when the package specification already exists.

• Initialize a variable with a constant value or formula within the declaration, if required; otherwise, the variable is initialized implicitly to NULL.

**Syntax Definition**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| package\_name | Name the package |
| public type and item declarations | Declare variables, constants, cursors, exceptions, or types |
| subprogram  specifications | Declare the PL/SQL subprograms |

**Introduction to Oracle9*i*: PL/SQL 12-8**

**Declaring Public Constructs COMM\_PACKAGE package**

**1**

**G\_COMM**

**Package**

**specification**

**RESET\_COMM procedure declaration**

**2**

**12-9** Copyright © Oracle Corporation, 2001. All rights reserved.

**Example of a Package Specification**

In the preceding slide, G\_COMM is a public (global) variable, and RESET\_COMM is a public procedure. In the package specification, you declare public variables, public procedures, and public functions.

The public procedures or functions are routines that can be invoked repeatedly by other constructs in the same package or from outside the package.

**Introduction to Oracle9*i*: PL/SQL 12-9**

**Creating a Package Specification:**

**Example**

**CREATE OR REPLACE PACKAGE comm\_package IS**

**g\_comm NUMBER := 0.10; --initialized to 0.10**

**PROCEDURE reset\_comm**

**(p\_comm IN NUMBER);**

**END comm\_package;**

**/**

****

• **G\_COMM is a global variable and is initialized to 0.10.** • **RESET\_COMM is a public procedure that is**

**implemented in the package body.**

**12-10** Copyright © Oracle Corporation, 2001. All rights reserved.

**Package Specification for COMM\_PACKAGE**

In the preceding slide, the variable G\_COMM and the procedure RESET\_COMM are public constructs.

**Introduction to Oracle9*i*: PL/SQL 12-10**

**Creating the Package Body**

**Syntax:**

**CREATE [OR REPLACE] PACKAGE BODY package\_name**

**IS|AS**

**private type and item declarations**

**subprogram bodies**

**END package\_name;**

• **The REPLACE option drops and recreates the**

**package body.**

• **Identifiers defined only in the package body are**

**private constructs. These are not visible outside**

**the package body.**

• **All private constructs must be declared before**

**they are used in the public constructs.**

**12-11** Copyright © Oracle Corporation, 2001. All rights reserved.

**Creating the Package Body**

To create packages, define all public and private constructs within the package body. • Specify the REPLACE option when the package body already exists.

• The order in which subprograms are defined within the package body is important: you must declare a variable before another variable or subprogram can refer to it, and you must declare or define private subprograms before calling them from other subprograms. It is quite common in the package body to see all private variables and subprograms defined first and the public subprograms defined last.

**Syntax Definition**

Define all public and private procedures and functions in the package body.

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| package\_name | Is the name of the package |
| private type and item declarations | Declares variables, constants, cursors, exceptions, or types |
| subprogram bodies | Defines the PL/SQL subprograms, public and private |

**Introduction to Oracle9*i*: PL/SQL 12-11**

**Public and Private Constructs COMM\_PACKAGE package**

**Package**

**specification**

**Package**

**body**

**G\_COMM**

**RESET\_COMM**

**procedure declaration**

**VALIDATE\_COMM**

**function definition**

**1**

**2 3**

**2 RESET\_COMM**

**procedure definition**

**12-12** Copyright © Oracle Corporation, 2001. All rights reserved.

**Create a Package Body Example**

In the preceding slide:

• 1 is a public (global) variable

• 2 is a public procedure

• 3 is a private function

You can define a private procedure or function to modularize and clarify the code of public procedures and functions.

**Note:** In the slide, the private function is shown above the public procedure. When you are coding the package body, the definition of the private function has to be above the definition of the public procedure.

Only subprograms and cursors declarations without body in a package specification have an underlying implementation in the package body. So if a specification declares only types, constants, variables, exceptions, and call specifications, the package body is unnecessary. However, the body can still be used to initialize items declared in the package specification.

**Introduction to Oracle9*i*: PL/SQL 12-12**

**Creating a Package Body: Example**

**comm\_pack.sql**

**CREATE OR REPLACE PACKAGE BODY comm\_package**

**IS**

**FUNCTION validate\_comm (p\_comm IN NUMBER)**

**RETURN BOOLEAN**

**IS**

**v\_max\_comm NUMBER;**

**BEGIN**

**SELECT MAX(commission\_pct)**

**INTO v\_max\_comm**

**FROM employees;**

**IF p\_comm > v\_max\_comm THEN RETURN(FALSE);**

**ELSE RETURN(TRUE);**

**END IF;**

**END validate\_comm; ...**

**12-13** Copyright © Oracle Corporation, 2001. All rights reserved.

**Package Body for COMM\_PACKAGE**

Define a function to validate the commission. The commission may not be greater than the highest commission among all existing employees.

**Introduction to Oracle9*i*: PL/SQL 12-13**

**Creating a Package Body: Example**

**comm\_pack.sql**

**PROCEDURE reset\_comm (p\_comm IN NUMBER)**

**IS**

**BEGIN**

**IF validate\_comm(p\_comm)**

**THEN g\_comm:=p\_comm; --reset global variable**

**ELSE**

**RAISE\_APPLICATION\_ERROR(-20210,’Invalid commission’); END IF;**

**END reset\_comm;**

**END comm\_package;**

**/**

****

**12-14** Copyright © Oracle Corporation, 2001. All rights reserved.

**Package Body for COMM\_PACKAGE (continued)**

Define a procedure that enables you to reset and validate the prevailing commission.

**Introduction to Oracle9*i*: PL/SQL 12-14**

**Invoking Package Constructs**

**Example 1: Invoke a function from a procedure within the same package.**

**CREATE OR REPLACE PACKAGE BODY comm\_package IS**

**. . .**

**PROCEDURE reset\_comm**

**(p\_comm IN NUMBER)**

**IS**

**BEGIN**

**IF validate\_comm(p\_comm)**

**THEN g\_comm := p\_comm;**

**ELSE**

**RAISE\_APPLICATION\_ERROR**

**(-20210, ’Invalid commission’);**

**END IF;**

**END reset\_comm;**

**END comm\_package;**

**12-15** Copyright © Oracle Corporation, 2001. All rights reserved.

**Invoking Package Constructs**

After the package is stored in the database, you can invoke a package construct within the package or from outside the package, depending on whether the construct is private or public. When you invoke a package procedure or function from within the same package, you do not need to qualify its name.

**Example 1**

Call the VALIDATE\_COMM function from the RESET\_COMM procedure. Both subprograms are in the COMM\_PACKAGE package.

**Introduction to Oracle9*i*: PL/SQL 12-15**

**Invoking Package Constructs**

**Example 2: Invoke a package procedure from *i*SQL\*Plus. EXECUTE comm\_package.reset\_comm(0.15)**

**Example 3: Invoke a package procedure in a different schema.**

**EXECUTE scott.comm\_package.reset\_comm(0.15)**

**Example 4: Invoke a package procedure in a remote**

**database.**

**EXECUTE comm\_package.reset\_comm@ny(0.15)**

**12-16** Copyright © Oracle Corporation, 2001. All rights reserved.

**Invoking Package Constructs (continued)**

When you invoke a package procedure or function from outside the package, you must qualify its name with the name of the package.

**Example 2**

Call the RESET\_COMM procedure from *i*SQL\*Plus, making the prevailing commission 0.15 for the user session.

**Example 3**

Call the RESET\_COMM procedure that is located in the SCOTT schema from *i*SQL\*Plus, making the prevailing commission 0.15 for the user session.

**Example 4**

Call the RESET\_COMM procedure that is located in a remote database that is determined by the database link named NY from *i*SQL\*Plus, making the prevailing commission 0.15 for the user session.

Adhere to normal naming conventions for invoking a procedure in a different schema, or in a different database on another node.

**Introduction to Oracle9*i*: PL/SQL 12-16**

**Declaring a Bodiless Package**

**CREATE OR REPLACE PACKAGE global\_consts IS**

**mile\_2\_kilo CONSTANT NUMBER := 1.6093;**

**kilo\_2\_mile CONSTANT NUMBER := 0.6214;**

**yard\_2\_meter CONSTANT NUMBER := 0.9144;**

**meter\_2\_yard CONSTANT NUMBER := 1.0936;**

**END global\_consts;**

**/**

**EXECUTE DBMS\_OUTPUT.PUT\_LINE(’20 miles = ’||20\***

**global\_consts.mile\_2\_kilo||’ km’)**

****

**12-17** Copyright © Oracle Corporation, 2001. All rights reserved.

**Declaring a Bodiless Package**

You can declare public (global) variables that exist for the duration of the user session. You can create a package specification that does not need a package body. As discussed earlier in this lesson, if a specification declares only types, constants, variables, exceptions, and call specifications, the package body is unnecessary.

**Example**

In the preceding slide, a package specification containing several conversion rates is defined. All the global identifiers are declared as constants.

A package body is not required to support this package specification because implementation details are not required for any of the constructs of the package specification.

**Introduction to Oracle9*i*: PL/SQL 12-17**

**Referencing a Public Variable from**

**a Stand-Alone Procedure**

**Example:**

**CREATE OR REPLACE PROCEDURE meter\_to\_yard**

**(p\_meter IN NUMBER, p\_yard OUT NUMBER)**

**IS**

**BEGIN**

**p\_yard := p\_meter \* global\_consts.meter\_2\_yard;**

**END meter\_to\_yard;**

**/**

**VARIABLE yard NUMBER**

**EXECUTE meter\_to\_yard (1, :yard)**

****

**12-18** Copyright © Oracle Corporation, 2001. All rights reserved.

**Example**

Use the procedure METER\_TO\_YARD to convert meters to yards, using the conversion rate packaged in GLOBAL\_CONSTS.

When you reference a variable, cursor, constant, or exception from outside the package, you must qualify its name with the name of the package.

**Introduction to Oracle9*i*: PL/SQL 12-18**

**Removing Packages**

**To remove the package specification and the body,**

**use the following syntax:**

**DROP PACKAGE package\_name;**

**To remove the package body, use the following syntax : DROP PACKAGE BODY package\_name;**

**12-19** Copyright © Oracle Corporation, 2001. All rights reserved.

**Removing a Package**

When a package is no longer required, you can use a SQL statement in *i*SQL\*Plus to drop it. A package has two parts, so you can drop the whole package or just the package body and retain the package specification.

**Introduction to Oracle9*i*: PL/SQL 12-19**

**Guidelines for Developing Packages**

• **Construct 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.**

• **Changes to the package specification require**

**recompilation of each referencing subprogram.**

• **The package specification should contain as few constructs as possible.**

**12-20** Copyright © Oracle Corporation, 2001. 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 users of the package. That way 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 gets initialized in a session the first time a construct from the package is invoked.

Changes to the package body do not require recompilation of dependent constructs, whereas changes to the package specification require recompilation of every stored subprogram that references the package. To reduce the need for recompiling when code is changed, place as few constructs as possible in a package specification.

**Introduction to Oracle9*i*: PL/SQL 12-20**

**Advantages of Packages**

• **Modularity: Encapsulate related constructs**

• **Easier application design: Code and compile**

**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**

**12-21** Copyright © Oracle Corporation, 2001. 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**

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. Then 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 can decide which constructs are public (visible and accessible) or private (hidden and inaccessible). Only the 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 calling programs. Also, by hiding implementation details from users, you protect the integrity of the package.

**Introduction to Oracle9*i*: PL/SQL 12-21**

**Advantages of Packages**

• **Added functionality: Persistency of 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**

**12-22** Copyright © Oracle Corporation, 2001. All rights reserved.

**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 only be accessed within the package.

**Better Performance**

When you call a packaged subprogram the first time, the entire package is loaded into memory. This way, later calls to related subprograms in the package require no further disk I/O. Packaged subprograms also stop cascading dependencies and so 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 datatype.

**Introduction to Oracle9*i*: PL/SQL 12-22**

**Summary**

**In this lesson, you should have learned how to:**

• **Improve organization, management, security, and performance by using packages**

• **Group related procedures and functions together in a package**

• **Change a package body without affecting a**

**package specification**

• **Grant security access to the entire package**

**12-23** Copyright © Oracle Corporation, 2001. All rights reserved.

**Summary**

You group related procedures and function together into a package. Packages improve organization, management, security, and performance.

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

**Introduction to Oracle9*i*: PL/SQL 12-23**

**Summary**

**In this lesson, you should have learned how to:**

• **Hide the source code from users**

• **Load the entire package into memory on the**

**first call**

• **Reduce disk access for subsequent calls**

• **Provide identifiers for the user session**

**12-24** Copyright © Oracle Corporation, 2001. All rights reserved.

**Summary (continued)**

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.

**Introduction to Oracle9*i*: PL/SQL 12-24**

**Command**

**Summary Task**

**CREATE [OR REPLACE] PACKAGE**

**CREATE [OR REPLACE] PACKAGE BODY**

**DROP PACKAGE**

**DROP PACKAGE BODY**

**Create (or modify) an existing package specification**

**Create (or modify) an existing package body**

**Remove both the package**

**specification and the package body Remove the package body only**

**12-25** Copyright © Oracle Corporation, 2001. All rights reserved.

**Summary (continued)**

You can create, delete, and modify packages. You can remove both package specification and body by using the DROP PACKAGE command. You can drop the package body without affecting its specification.

**Introduction to Oracle9*i*: PL/SQL 12-25**

**Practice 12 Overview**

**This practice covers the following topics:**

• **Creating packages**

• **Invoking package program units**

**12-26** Copyright © Oracle Corporation, 2001. All rights reserved.

**Practice 12 Overview**

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

**Introduction to Oracle9*i*: PL/SQL 12-26**

**Practice 12**

1. Create a package specification and body called JOB\_PACK. (You can save the package body and specification in two separate files.) This package contains your ADD\_JOB, UPD\_JOB, and DEL\_JOB procedures, as well as your Q\_JOB function.

**Note:** Use the code in your previously saved script files when creating the package. a. Make all the constructs public.

**Note:** Consider whether you still need the stand-alone procedures and functions you just packaged.

b. Invoke your ADD\_JOB procedure by passing values IT\_SYSAN and SYSTEMS ANALYST as parameters.

c. Query the JOBS table to see the result.



2. Create and invoke a package that contains private and public constructs.

a. Create a package specification and package body called EMP\_PACK that contains your NEW\_EMP procedure as a public construct, and your VALID\_DEPTID function as a private construct. (You can save the specification and body into separate files.)

b. Invoke the NEW\_EMP procedure, using 15 as a department number. As the department ID 15 does not exist in the DEPARTMENTS table, you should get an error message as specified in the exception handler of your procedure.

c. Invoke the NEW\_EMP procedure, using an existing department ID 80.

**If you have time:**

3. a. Create a package called CHK\_PACK that contains the procedures CHK\_HIREDATE and CHK\_DEPT\_MGR. Make both constructs public. (You can save the specification and body into separate files.)

The procedure CHK\_HIREDATE checks whether an employee’s hire date is within the following range: [SYSDATE - 50 years, SYSDATE + 3 months].

**Note:**

• If the date is invalid, you should raise an application error with an appropriate message indicating why the date value is not acceptable.

• Make sure the time component in the date value is ignored.

• Use a constant to refer to the 50 years boundary.

• A null value for the hire date should be treated as an invalid hire date.

The procedure CHK\_DEPT\_MGR checks the department and manager combination for a given employee. The CHK\_DEPT\_MGR procedure accepts an employee ID and a manager ID. The procedure checks that the manager and employee work in the same department. The procedure also checks that the job title of the manager number provided is MANAGER.

**Note:** If the department number and manager combination is invalid, you should raise an application error with an appropriate message.

**Introduction to Oracle9*i*: PL/SQL 12-27**

**Practice 12 (continued)**

b. Test the CHK\_HIREDATE procedure with the following command: **EXECUTE chk\_pack.chk\_hiredate(’01-JAN-47’)** What happens, and why?

c. Test the CHK\_HIREDATE procedure with the following command: **EXECUTE chk\_pack.chk\_hiredate(NULL)**

What happens, and why?

d. Test the CHK\_DEPT\_MGR procedure with the following command: **EXECUTE chk\_pack.chk\_dept\_mgr(117,100)**

What happens, and why?

**Introduction to Oracle9*i*: PL/SQL 12-28**

**13**

**More Package Concepts**

****Copyright © Oracle Corporation, 2001. All rights reserved.

**Objectives**

**After completing this lesson, you should be able to**

**do the following:**

• **Write packages that use the overloading feature**

• **Describe errors with mutually referential**

**subprograms**

• **Initialize variables with a one-time-only procedure** • **List the four purity levels of a function**

• **Identify persistent states**

**13-2** Copyright © Oracle Corporation, 2001. All rights reserved.

**Lesson Aim**

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

**Introduction to Oracle9*i*: PL/SQL 13-2**

**Overloading**

• **Enables you to use the same name for different**

**subprograms inside a PL/SQL block, a**

**subprogram, or a package**

• **Requires the formal parameters of the**

**subprograms to differ in number, order, or**

**datatype family**

• **Enables you to build more flexibility because a**

**user or application is not restricted by the specific**

**datatype or number of formal parameters**

**Note: Only local or packaged subprograms can be**

**overloaded. You cannot overload stand-alone**

**subprograms.**

**13-3** Copyright © Oracle Corporation, 2001. All rights reserved.

**Overloading**

This feature enables you to define different subprograms with the same name. You can distinguish the subprograms both by name and by parameters. Sometimes the processing in two subprograms is the same, but the parameters passed to them varies. In that case it is logical to give them the same name. PL/SQL determines which subprogram is called by checking its formal parameters. Only local or packaged subprograms can be overloaded. Stand-alone subprograms cannot be overloaded.

**Restrictions**

You cannot overload:

• Two subprograms if their formal parameters differ only in datatype and the different datatypes 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 above features.

**Note:** The above restrictions apply if the names of the parameters are also the same. If you use different names for the parameters, then 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 like-named subprograms at the same level of scope, the compiler needs an exact match in number, order, and datatype between the actual and

formal parameters.

**Introduction to Oracle9*i*: PL/SQL 13-3**

**Overloading: Example**

**over\_pack.sql**

**CREATE OR REPLACE PACKAGE over\_pack**

**IS**

**PROCEDURE add\_dept**

**(p\_deptno IN departments.department\_id%TYPE,**

**p\_name IN departments.department\_name%TYPE**

**DEFAULT ’unknown’,**

**p\_loc IN departments.location\_id%TYPE DEFAULT 0);**

**PROCEDURE add\_dept**

**(p\_name IN departments.department\_name%TYPE**

**DEFAULT ’unknown’,**

**p\_loc IN departments.location\_id%TYPE DEFAULT 0);**

**END over\_pack;**

**/**

****

**13-4** Copyright © Oracle Corporation, 2001. All rights reserved.

**Overloading: Example**

The preceding slide shows the package specification of a package with overloaded procedures. The package contains ADD\_DEPT as the name of two overloaded procedures. The first definition takes three parameters to be able to insert a new department to the department table. The second definition takes only two parameters, because the department ID is populated through a sequence.

**Introduction to Oracle9*i*: PL/SQL 13-4**

**Overloading: Example**

**over\_pack\_body.sql**

**CREATE OR REPLACE PACKAGE BODY over\_pack IS**

**PROCEDURE add\_dept**

**(p\_deptno IN departments.department\_id%TYPE,**

**p\_name IN departments.department\_name%TYPE DEFAULT ’unknown’, p\_loc IN departments.location\_id%TYPE DEFAULT 0)**

**IS**

**BEGIN**

**INSERT INTO departments (department\_id,**

**department\_name, location\_id)**

**VALUES (p\_deptno, p\_name, p\_loc);**

**END add\_dept;**

**PROCEDURE add\_dept**

**(p\_name IN departments.department\_name%TYPE DEFAULT ’unknown’, p\_loc IN departments.location\_id%TYPE DEFAULT 0)**

**IS**

**BEGIN**

**INSERT INTO departments (department\_id,**

**department\_name, location\_id)**

**VALUES (departments\_seq.NEXTVAL, p\_name, p\_loc);**

**END add\_dept;**

**END over\_pack;**

**/**

**13-5** Copyright © Oracle Corporation, 2001. All rights reserved.

**Overloading Example (continued)**

If you call ADD\_DEPT with an explicitly provided department ID, PL/SQL uses the first version of the procedure. If you call ADD\_DEPT with no department ID, PL/SQL uses the second version. EXECUTE over\_pack.add\_dept (980,’Education’,2500)

EXECUTE over\_pack.add\_dept (’Training’, 2400)

SELECT \* FROM departments

WHERE department\_id = 980;

SELECT \* FROM departments

WHERE department\_name = ’Training’;

**Introduction to Oracle9*i*: PL/SQL 13-5**

**Overloading: Example**

• **Most built-in functions are overloaded.**

• **For example, see the TO\_CHAR function of the**

**STANDARD package.**

**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;**

• **If you redeclare a built-in subprogram in a PL/SQL program, your local declaration overrides the**

**global declaration.**

**13-6** Copyright © Oracle Corporation, 2001. All rights reserved.

**Overloading Example (continued)**

Most built-in functions are overloaded. For example, the function TO\_CHAR in the package STANDARD has four different declarations, as shown in the slide. The function can take either the DATE or the NUMBER datatype and convert it to the character datatype. The format into which the date or number has to be converted can also be specified in the function call.

If you redeclare a built-in subprogram in another PL/SQL program, your local declaration overrides the standard or built-in subprogram. To be able to access the built-in subprogram, you need to qualify it with its package name. For example, if you redeclare the TO\_CHAR function, to access the built-in function you refer it as: STANDARD.TO\_CHAR.

If you redeclare a built-in subprogram as a stand-alone subprogram, to be able to access your subprogram you need to qualify it with your schema name, for example, SCOTT.TO\_CHAR.

**Introduction to Oracle9*i*: PL/SQL 13-6**

**Using Forward Declarations**

**You must declare identifiers before referencing them.**

**CREATE OR REPLACE PACKAGE BODY forward\_pack**

**ISPROCEDURE award\_bonus(. . .)**

**IS**

**BEGIN**

**calc\_rating(. . .); --illegal reference**

**END;**

**PROCEDURE calc\_rating(. . .)**

**IS**

**BEGIN ...**

**END;**

**END forward\_pack;**

**/**

**13-7** Copyright © Oracle Corporation, 2001. All rights reserved.

**Using Forward Declarations**

PL/SQL does not allow forward references. You must declare an identifier before using it. Therefore, a subprogram must be declared before calling it.

In the preceding slide, the procedure CALC\_RATING 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 always work. Suppose the procedures call each other or you absolutely want to define them in alphabetical order.

PL/SQL enables for a special subprogram declaration called a forward declaration. It consists of the subprogram specification terminated by a semicolon. You can use forward declarations to do the following: • Define subprograms in logical or alphabetical order

• Define mutually recursive subprograms

• Group subprograms in a package

Mutually recursive programs are programs that call each other directly or indirectly.

**Introduction to Oracle9*i*: PL/SQL 13-7**

**Using Forward Declarations**

**CREATE OR REPLACE PACKAGE BODY forward\_pack**

**IS**

**PROCEDURE calc\_rating(. . .); -- forward declaration**

**PROCEDURE award\_bonus(. . .)**

**IS -- subprograms defined BEGIN -- in alphabetical order calc\_rating(. . .);**

**. . .**

**END;**

**PROCEDURE calc\_rating(. . .)**

**IS**

**BEGIN**

**. . .**

**END;**

**END forward\_pack;**

**/**

**13-8** Copyright © Oracle Corporation, 2001. All rights reserved.

**Using Forward Declarations (continued)**

• The formal parameter list 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**

Forward declarations typically let you group related subprograms in a package. The subprogram specifications go in the package specification, and the subprogram bodies go in the package body, where they are invisible to the applications. In this way, packages enable you to hide implementation details.

**Introduction to Oracle9*i*: PL/SQL 13-8**

**Creating a One-Time-Only Procedure**

**CREATE OR REPLACE PACKAGE taxes**

**IS**

**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 tax**

**FROM tax\_rates**

**WHERE rate\_name = ’TAX’;**

**END taxes;**

**/**

**13-9** Copyright © Oracle Corporation, 2001. All rights reserved.

**Define an Automatic, One-Time-Only Procedure**

A one-time-only procedure is executed only once, when the package is first invoked within the user session. In the preceding slide, the current value for TAX is set to the value in the TAX\_RATES table the first time the TAXES package is referenced.

**Note:** Initialize public or private variables with an automatic, one-time-only procedure when the derivation is too complex to embed within the variable declaration. In this case, do not initialize the variable in the declaration, because the value is reset by the one-time-only procedure.

The keyword END is not used at the end of a one-time-only procedure. Observe that in the example in the slide, there is no END at the end of the one-time-only procedure.

**Introduction to Oracle9*i*: PL/SQL 13-9**

**Restrictions on Package Functions**

**Used in SQL**

**A function called from:**

• **A query or DML statement may not end the current transaction, create or roll back to a savepoint, or**

**ALTER the system or session.**

• **A query statement or a parallelized DML statement may not execute a DML statement or modify the**

**database.**

• **A DML statement may not read or modify the**

**particular table being modified by that DML**

**statement.**

**Note: Calls to subprograms that break the above**

**restrictions are not allowed.**

**13-10** Copyright © Oracle Corporation, 2001. All rights reserved.

**Controlling Side Effects**

For the Oracle server to execute a SQL statement that calls a stored function, it must know the purity level of a stored functions, that is, whether the functions are free of side effects. Side effects are changes to database tables or public packaged variables (those declared in a package specification). Side effects could 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 side effects are not allowed when a function is called from a SQL query or DML statement. Therefore, the following restrictions apply to stored functions called from SQL expressions:

• A function called from a query or DML statement may not end the current transaction, create or roll back to a savepoint, or alter the system or session

• A function called from a query statement or from a parallelized DML statement may not execute a DML statement or otherwise modify the database

• A function called from a DML statement may not read or modify the particular table being modified by that DML statement

**Note:** In releases prior to Oracle8*i*, the purity checking used to be performed during compilation time, by including the PRAGMA RESTRICT\_REFERENCES compiler directive in the package specification. But from Oracle8*i*, a user-written function can be called from a SQL statement without any compile-time checking of its purity. You can use PRAGMA RESTRICT\_REFERENCES to ask the PL/SQL compiler to verify that a function has only the side effects that you expect. SQL statements, package variable accesses, or calls to functions that violate the declared restrictions continue to raise PL/SQL compilation errors to help you isolate the code that has unintended effects.

**Note:** The restrictions on functions discussed above are the same as those discussed in the lesson "*Creating Functions."*

**Introduction to Oracle9*i*: PL/SQL 13-10**

**User Defined Package: taxes\_pack**

**CREATE OR REPLACE PACKAGE taxes\_pack**

**IS**

**FUNCTION tax (p\_value IN NUMBER) RETURN NUMBER;**

**END taxes\_pack;**

**/**

**CREATE OR REPLACE PACKAGE BODY taxes\_pack**

**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\_pack;**

**/**

****

**13-11** Copyright © Oracle Corporation, 2001. All rights reserved.

**Example**

Encapsulate the function TAX in the package TAXES\_PACK. The function is called from SQL statements on remote databases.

**Introduction to Oracle9*i*: PL/SQL 13-11**

**Invoking a User-Defined Package Function from a SQL Statement**

**SELECT taxes\_pack.tax(salary), salary, last\_name**

**FROM employees;**

****

**13-12** Copyright © Oracle Corporation, 2001. All rights reserved.

**Calling Package Functions**

You call PL/SQL functions the same way that you call built-in SQL functions.

**Example**

Call the TAX function (in the TAXES\_PACK package) from a SELECT statement.

**Note:** If you are using Oracle versions prior to 8*i*, you need to assert the purity level of the function in the package specification by using PRAGMA RESTRICT\_REFERENCES. If this is not specified, you get an error message saying that the function TAX does not guarantee that it will not update the database while invoking the package function in a query.

**Introduction to Oracle9*i*: PL/SQL 13-12**

**Persistent State of Package**

**Variables: Example**

**CREATE OR REPLACE PACKAGE comm\_package IS**

**g\_comm NUMBER := 10; --initialized to 10**

**PROCEDURE reset\_comm (p\_comm IN NUMBER);**

**END comm\_package;**

**/**

**CREATE OR REPLACE PACKAGE BODY comm\_package IS**

**FUNCTION validate\_comm (p\_comm IN NUMBER)**

**RETURN BOOLEAN**

**IS v\_max\_comm NUMBER;**

**BEGIN**

**... -- validates commission to be less than maximum**

**-- commission in the table**

**END validate\_comm;**

**PROCEDURE reset\_comm (p\_comm IN NUMBER)**

**IS BEGIN**

**... -- calls validate\_comm with specified value**

**END reset\_comm;**

**END comm\_package;**

**/**

**13-13** Copyright © Oracle Corporation, 2001. All rights reserved.

**Persistent State of Package Variables**

This sample package illustrates the persistent state of package variables. The VALIDATE\_COMM function validates commission to be no more than maximum currently earned. The RESET\_COMM procedure invokes the VALIDATE\_COMM function. If you try to reset the commission to be higher than the maximum, the exception RAISE\_APPLICATION\_ERROR is raised. On the next page, the RESET\_COMM procedure is used in the example.

**Note:** Refer to page 12-13 for the code of the VALIDATE\_COMM function and the RESET\_COMM procedure. In the VALIDATE\_COMM function, maximum salary from the EMPLOYEES table is selected into the variable V\_MAXSAL. Once the variable is assigned a value, the value persists in the session until it is modified again. The example in the following slide shows how the value of a global package variable persists for a session.

**Introduction to Oracle9*i*: PL/SQL 13-13**

**Persistent State of Package Variables**

**Time**

**9:00 EXECUTE**

**Scott Jones**

**comm\_package.reset\_comm**

**(0.25)**

**max\_comm=0.4 > 0.25**

**g\_comm = 0.25 9:30INSERT INTO employees**

**(last\_name, commission\_pct)**

**VALUES (’Madonna’, 0.8);**

**max\_comm=0.8**

**9:35 EXECUTE**

**comm\_package.reset\_comm(0.5)**

**max\_comm=0.8 > 0.5**

**10:00 EXECUTE g\_comm = 0.5**

**comm\_package.reset\_comm**

**(0.6)**

**max\_comm=0.4 < 0.6 INVALID 11:00 ROLLBACK;**

**11:01 EXIT**

**11:45 Logged In again. g\_comm = 10,**

**max\_comm=0.4**

**12:00 EXECUTE**

**VALID**

**comm\_package.reset\_comm(0.25)**

**13-14** Copyright © Oracle Corporation, 2001. All rights reserved.

**Controlling the Persistent State of a Package Variable**

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. 1. Initialize the variable within its declaration or within an automatic, one-time-only procedure. 2. Change the value of the variable by means of package procedures.

3. The value of the variable is released when the user disconnects.

The sequence of steps in the preceding slide shows how the state of a package variable persists. 9:00: When Scott invoked the procedure RESET\_COMM with a commission percentage value 0.25, the global variable G\_COMM was initialized to 10 in his session. The value 0.25 was validated with the maximum commission percentage value 0.4 (obtained from the EMPLOYEES table). Because 0.25 is less than 0.4, the global variable was set to 0.25. 9:30: Jones inserted a new row into EMPLOYEES table with commission percentage value 0.8.

9:35: Jones invoked the procedure RESET\_COMM with a commission percentage value 0.5. The global variable G\_COMM was initialized to 10 in his session. The value 0.5 was validated with the maximum commission percentage value 0.8 (because the new row has 0.8). Because 0.5 is less than 0.8, the global variable was set to 0.5.

10:00: Scott invoked the procedure with commission percentage value of 0.6. This value is more than the maximum commission percentage 0.4 (Scott could not see new value because Jones did not complete the transaction). Hence, it was invalid.

11:00 to 12:00: Jones rolled back the transaction and exited the session. The global value was initialized to 10 when he logged in at 11:45. The procedure was successful because the new value 0.25 is less than the maximum

value 0.4.

**Introduction to Oracle9*i*: PL/SQL 13-14**

**Controlling the Persistent State of a**

**Package Cursor**

**Example:**

**CREATE OR REPLACE PACKAGE pack\_cur**

**IS**

**CURSOR c1 IS SELECT employee\_id**

**FROM employees**

**ORDER BY employee\_id DESC;**

**PROCEDURE proc1\_3rows;**

**PROCEDURE proc4\_6rows;**

**END pack\_cur;**

**/**

****

**13-15** Copyright © Oracle Corporation, 2001. All rights reserved.

**Controlling the Persistent State of a Package Cursor**

**Example**

Use the following steps to control a public cursor:

1. Declare the public (global) cursor in the package specification.

2. Open the cursor and fetch successive rows from the cursor, using one (public) packaged procedure, PROC1\_3ROWS.

3. Continue to fetch successive rows from the cursor, and then close the cursor by using another (public) packaged procedure, PROC4\_6ROWS.

The preceding slide shows the package specification for PACK\_CUR.

**Introduction to Oracle9*i*: PL/SQL 13-15**

**Controlling the Persistent State of a**

**Package Cursor**

**CREATE OR REPLACE PACKAGE BODY pack\_cur IS**

**v\_empno NUMBER;**

**PROCEDURE proc1\_3rows IS**

**BEGIN**

**OPEN c1;**

**LOOP**

**FETCH c1 INTO v\_empno;**

**DBMS\_OUTPUT.PUT\_LINE(’Id :’ ||(v\_empno));**

**EXIT WHEN c1%ROWCOUNT >= 3;**

**END LOOP**

**END proc1\_3rows;**

**PROCEDURE proc4\_6rows IS**

**BEGIN**

**LOOP**

**FETCH c1 INTO v\_empno;**

**DBMS\_OUTPUT.PUT\_LINE(’Id :’ ||(v\_empno));**

**EXIT WHEN c1%ROWCOUNT >= 6;**

**END LOOP;**

**CLOSE c1;**

**END proc4\_6rows;**

**END pack\_cur;**

**/**

**13-16** Copyright © Oracle Corporation, 2001. All rights reserved.

**Controlling the Persistent State of a Package Cursor (continued)**

**Example**

The preceding slide shows the package body for PACK\_CUR to support the package specification. In the package body:

1. Open the cursor and fetch successive rows from the cursor by using one packaged procedure, PROC1\_3ROWS.

2. Continue to fetch successive rows from the cursor and close the cursor, using another packaged procedure, PROC4\_6ROWS.

**Introduction to Oracle9*i*: PL/SQL 13-16**

**Executing PACK\_CUR**

**SET SERVEROUTPUT ON**

**EXECUTE pack\_cur.proc1\_3rows**

**EXECUTE pack\_cur.proc4\_6rows**

****

****

**13-17** Copyright © Oracle Corporation, 2001. All rights reserved.

**Result of Executing PACK\_CUR**

The state of a package variable or cursor persists across transactions within a session. The state does not persist from session to session for the same user, nor does it persist from user to user.

**Introduction to Oracle9*i*: PL/SQL 13-17**

**PL/SQL Tables**

**and Records in Packages**

**CREATE OR REPLACE PACKAGE emp\_package IS**

**TYPE emp\_table\_type IS TABLE OF employees%ROWTYPE**

**INDEX BY BINARY\_INTEGER;**

**PROCEDURE read\_emp\_table**

**(p\_emp\_table OUT emp\_table\_type);**

**END emp\_package;**

**/**

**CREATE OR REPLACE PACKAGE BODY emp\_package IS**

**PROCEDURE read\_emp\_table**

**(p\_emp\_table OUT emp\_table\_type) IS**

**i BINARY\_INTEGER := 0;**

**BEGIN**

**FOR emp\_record IN (SELECT \* FROM employees)**

**LOOP**

**emp\_table(i) := emp\_record;**

**i:= i+1;**

**END LOOP;**

**END read\_emp\_table;**

**END emp\_package;**

**/**

**13-18** Copyright © Oracle Corporation, 2001. All rights reserved.

**Passing Tables of Records to Procedures or Functions inside a Package** Invoke the READ\_EMP\_TABLE procedure from an anonymous PL/SQL block, using *i*SQL\*Plus. DECLARE

v\_emp\_table emp\_package.emp\_table\_type;

BEGIN

emp\_package.read\_emp\_table(v\_emp\_table);

DBMS\_OUTPUT.PUT\_LINE(’An example: ’||v\_emp\_table(4).last\_name); END;

/



To invoke the procedure READ\_EMP\_TABLE from another procedure or any PL/SQL block outside the package, the actual parameter referring to the OUT parameter P\_EMP\_TABLE must be prefixed with its package name. In the example above, the variable V\_EMP\_TABLE is declared of the EMP\_TABLE\_TYPE type with the package name added as a prefix.

**Introduction to Oracle9*i*: PL/SQL 13-18**

**Summary**

**In this lesson, you should have learned how to:**

• **Overload subprograms**

• **Use forward referencing**

• **Use one-time-only procedures**

• **Describe the purity level of package functions**

• **Identifiy the persistent state of packaged objects**

**13-19** Copyright © Oracle Corporation, 2001. All rights reserved.

**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 in situations when the processing in both the subprograms is the same, but the parameters passed to them varies.

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

A one-time-only procedure 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 that the user disconnects.

**Introduction to Oracle9*i*: PL/SQL 13-19**

**Practice 13 Overview**

**This practice covers the following topics:**

• **Using overloaded subprograms**

• **Creating a one-time-only procedure**

**13-20** Copyright © Oracle Corporation, 2001. All rights reserved.

**Practice 13 Overview**

In this practice you create a package containing an overloaded function. You also create a one-time-only procedure within a package to populate a PL/SQL table.

**Introduction to Oracle9*i*: PL/SQL 13-20**

**Practice 13**

1. Create a package called OVER\_LOAD. Create two functions in this package; name each function PRINT\_IT. The function accepts a date or character string and prints a date or a number, depending on how the function is invoked.

**Note:**

• To print the date value, use DD-MON-YY as the input format, and FmMonth,dd yyyy as the output format. Make sure you handle invalid input.

• To print out the number, use 999,999.00 as the input format.

a. Test the first version of PRINT\_IT with the following set of commands:

**VARIABLE display\_date VARCHAR2(20)**

**EXECUTE :display\_date := over\_load.print\_it(’08-MAR-01’) PRINT display\_date**

****b. Test the second version of PRINT\_IT with the following set of commands: **VARIABLE g\_emp\_sal NUMBER**

**EXECUTE :g\_emp\_sal := over\_load.print\_it(’33,600’)**

**PRINT g\_emp\_sal**

****

2. Create a new package, called CHECK\_PACK, to implement a new business rule. a. Create a procedure called CHK\_DEPT\_JOB to verify whether a given combination of department ID and job is a valid one. In this case *valid* means that it must be a combination that currently exists in the EMPLOYEES table.

**Note:**

• Use a PL/SQL table to store the valid department and job combination.

• The PL/SQL table needs to be populated only once.

• Raise an application error with an appropriate message if the combination is not valid.

b. Test your CHK\_DEPT\_JOB package procedure by executing the following command: **EXECUTE check\_pack.chk\_dept\_job(50,’ST\_CLERK’)**

What happens, and why?

c. Test your CHK\_DEPT\_JOB package procedure by executing the following command: **EXECUTE check\_pack.chk\_dept\_job(20,’ST\_CLERK’)**

What happens, and why?

**Introduction to Oracle9*i*: PL/SQL 13-21**

**Introduction to Oracle9*i*: PL/SQL 13-22**

**14**

**Oracle Supplied Packages**

****Copyright © Oracle Corporation, 2001. All rights reserved.

**Objectives**

**After completing this lesson, you should be able to**

**do the following:**

• **Write dynamic SQL statements using DBMS\_SQL**

**and EXECUTE IMMEDIATE**

• **Describe the use and application of some Oracle server supplied packages:**

– **DBMS\_DDL**

– **DBMS\_JOB**

– **DBMS\_OUTPUT**

– **UTL\_FILE**

– **UTL\_HTTP and UTL\_TCP**

**14-2** Copyright © Oracle Corporation, 2001. All rights reserved.

**Lesson Aim**

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

**Introduction to Oracle9*i*: PL/SQL 14-2**

**Using Supplied Packages**

**Oracle-supplied packages:**

• **Are provided with the Oracle server**

• **Extend the functionality of the database**

• **Enable access to certain SQL features normally**

**restricted for PL/SQL**

**14-3** Copyright © Oracle Corporation, 2001. All rights reserved.

**Using Supplied Packages**

Packages are provided with the Oracle server to allow either PL/SQL access to certain SQL features, or to extend the functionality of the database.

You may take advantage of 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.

**Introduction to Oracle9*i*: PL/SQL 14-3**

**Using Native Dynamic SQL**

**Dynamic SQL:**

• **Is a SQL statement that contains variables that**

**may change during run-time**

• **Is a SQL statement with placeholders and is stored as a character string**

• **Enables general-purpose code to be written**

• **Enables data-definition and data-control or**

**session-control statements to be written and**

**executed from PL/SQL**

• **Is written using either DBMS\_SQL or native dynamic SQL**

**14-4** Copyright © Oracle Corporation, 2001. All rights reserved.

**Using Native Dynamic SQL (Dynamic SQL)**

You can write PL/SQL blocks that use dynamic SQL. Dynamic SQL statements are not embedded in your source program but rather are stored in character strings that are input to, or built by, the program. That is, the SQL statements can be created dynamically at run time by using variables. For example, you use dynamic SQL to create a procedure that operates on a table whose name is not known until run time, or to write and execute a data definition language (DDL) statement (such as CREATE TABLE), a data control statement (such as GRANT), or a session control statement (such as ALTER SESSION). In PL/SQL, such statements cannot be executed statically.

In Oracle8, and earlier, you have to use DBMS\_SQL to write dynamic SQL.

In Oracle 8*i*, you can use DBMS\_SQL or native dynamic SQL. The EXECUTE IMMEDIATE statement can perform dynamic single-row queries. Also, this is used for functionality such as objects and collections, which are not supported by DBMS\_SQL. If the statement is a multirow SELECT statement, you use OPEN-FOR, FETCH, and CLOSE statements.

**Introduction to Oracle9*i*: PL/SQL 14-4**

**Execution Flow**

**SQL statements go through various stages:**

• **Parse**

• **Bind**

• **Execute**

• **Fetch**

**Note: Some stages may be skipped.**

**14-5** Copyright © Oracle Corporation, 2001. All rights reserved.

**Steps to Process SQL Statements**

All SQL statements have to go through various stages. Some stages may be skipped. **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 ensuring that the relevant privileges to those objects exist.

**Bind**

After parsing, the Oracle server knows the meaning of the Oracle statement but still may not have enough information to execute the statement. The Oracle server may need values for any bind variable in the statement. The process of obtaining these values is called binding variables.

**Execute**

At this point, the Oracle server has all necessary information and resources, and the statement is executed. **Fetch**

In the fetch stage, 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. You can fetch queries, but not the DML statements.

**Introduction to Oracle9*i*: PL/SQL 14-5**

**Using the DBMS\_SQL Package**

**The DBMS\_SQL package is used to write dynamic SQL in stored procedures and to parse DDL statements. Some of the procedures and functions of the package include:**

– **OPEN\_CURSOR**

– **PARSE**

– **BIND\_VARIABLE**

– **EXECUTE**

– **FETCH\_ROWS**

– **CLOSE\_CURSOR**

**14-6** Copyright © Oracle Corporation, 2001. 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.

DBMS\_SQL can issue data definition language statements in PL/SQL. For example, you can choose to issue a DROP TABLE statement from within a stored procedure.

The operations provided by this package are performed under the current user, not under the package owner SYS. Therefore, if the caller is an anonymous PL/SQL block, the operations are performed according to the privileges of the current user; if the caller is a stored procedure, the operations are performed according to the owner of the stored procedure.

Using this package to execute DDL statements can result in a deadlock. The most likely reason for this is that the package is being used to drop a procedure that you are still using.

**Introduction to Oracle9*i*: PL/SQL 14-6**

**Components of the DBMS\_SQL Package**

The DBMS\_SQL package uses dynamic SQL to access the database.

|  |  |
| --- | --- |
| **Function or Procedure Description** |  |
| OPEN\_CURSOR | Opens a new cursor and assigns a cursor ID number |
| PARSE | Parses the DDL or DML statement: that is, checks the statement’s syntax and associates it with the opened cursor (DDL statements are immediately executed when parsed) |
| BIND\_VARIABLE | Binds the given value to the variable identified by its name in the parsed statement in the given cursor |
| EXECUTE | Executes the SQL statement and returns the number of rows processed |
| FETCH\_ROWS | Retrieves a row for the specified cursor (for multiple rows, call in a loop) |
| CLOSE\_CURSOR | Closes the specified cursor |

**Introduction to Oracle9*i*: PL/SQL 14-7**

**Using DBMS\_SQL**

**CREATE OR REPLACE PROCEDURE delete\_all\_rows**

**(p\_tab\_name IN VARCHAR2, p\_rows\_del OUT NUMBER)**

**IScursor\_name INTEGER;**

**BEGIN**

**cursor\_name := DBMS\_SQL.OPEN\_CURSOR;**

**DBMS\_SQL.PARSE(cursor\_name, ’DELETE FROM ’||p\_tab\_name, DBMS\_SQL.NATIVE );**

**p\_rows\_del := DBMS\_SQL.EXECUTE (cursor\_name);**

**DBMS\_SQL.CLOSE\_CURSOR(cursor\_name);**

**END;**

**/**

**Use dynamic SQL to delete rows**

**VARIABLE deleted NUMBER**

**EXECUTE delete\_all\_rows(’employees’, :deleted)**

**PRINT deleted**

****

**14-8** Copyright © Oracle Corporation, 2001. All rights reserved.

**Example of a DBMS\_SQL Package**

In the preceding slide, the table name is passed into the procedure DELETE\_ALL\_ROWS by using an IN parameter. The procedure uses dynamic SQL to delete rows from the specified table. The number of rows deleted as a result of the successful execution of the dynamic SQL are passed to the calling environment through an OUT parameter.

**How to Process Dynamic DML**

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 row processed.

4. Use CLOSE\_CURSOR to close the cursor.

**Introduction to Oracle9*i*: PL/SQL 14-8**

**Using the EXECUTE IMMEDIATE statement**

**Use the EXECUTE IMMEDIATE for native dynamic SQL with better performance.**

**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.**

**14-9** Copyright © Oracle Corporation, 2001. All rights reserved.

**Using the EXECUTE IMMEDIATE Statement**

**Syntax Definition**

|  |  |
| --- | --- |
| **Parameter** | **Description** |
| dynamic\_string | A string expression that represents a dynamic SQL statement (without terminator) or a PL/SQL block (with terminator) |
| define\_variable | A variable that stores the selected column value |
| record | A user-defined or %ROWTYPE record that stores a selected row |
| bind\_argument | An expression whose value is passed to the dynamic SQL statement or PL/SQL block |

You can use the INTO clause for a single-row query, but you must use OPEN-FOR, FETCH, and CLOSE for a multirow query.

**Note:** The syntax shown in the slide is not complete. The other clauses of the statement are discussed in the *Advanced PL/SQL* course.

**Introduction to Oracle9*i*: PL/SQL 14-9**

**Using the EXECUTE IMMEDIATE Statement (continued)**

In the EXECUTE IMMEDIATE statement:

• 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 RETURNING INTO clause specifies the variables into which column values are returned. It is used only for DML statements that have a RETURNING clause (without a BULK COLLECT clause). For each value returned by the DML statement, there must be a corresponding, type compatible variable in the RETURNING INTO clause.

• The USING clause holds all bind arguments. The default parameter mode is IN. For DML statements that have a RETURNING clause, you can place OUT arguments in the RETURNING INTO clause without specifying the parameter mode, which, by definition, is OUT. If you use both the USING clause and the RETURNING INTO clause, the USING clause can contain only IN arguments.

At run time, bind arguments replace corresponding placeholders in the dynamic string. Thus, every placeholder must be associated with a bind argument in the USING clause or RETURNING INTO clause. You can use numeric, character, and string literals as bind arguments, but you cannot use Boolean literals (TRUE, FALSE, and NULL).

Dynamic SQL supports all the SQL data types. For example, define variables and bind arguments can be collections, LOBs, instances of an object type, and REFs. As a rule, dynamic SQL does not support PL/SQL-specific types. For example, define variables and bind arguments cannot be Booleans or index by tables. The only exception is that a PL/SQL record can appear in the INTO clause.

You can execute a dynamic SQL statement repeatedly, using new values for the bind arguments. However, you incur some overhead because EXECUTE IMMEDIATE reprepares the dynamic string before every execution.

**Introduction to Oracle9*i*: PL/SQL 14-10**

**Dynamic SQL Using EXECUTE IMMEDIATE**

**CREATE PROCEDURE del\_rows**

**(p\_table\_name IN VARCHAR2,**

**IS**

**p\_rows\_deld OUT NUMBER)**

**BEGIN**

**EXECUTE IMMEDIATE ’delete from ’||p\_table\_name;**

**p\_rows\_deld := SQL%ROWCOUNT;**

**END;**

**/**

**VARIABLE deleted NUMBER**

**EXECUTE del\_rows(’test\_employees’,:deleted)**

**PRINT deleted**

****

**14-11** Copyright © Oracle Corporation, 2001. All rights reserved.

**Dynamic SQL Using EXECUTE IMMEDIATE**

This is the same dynamic SQL as seen with DBMS\_SQL, using the Oracle8*i* statement EXECUTE IMMEDIATE. The EXECUTE IMMEDIATE statement prepares (parses) and immediately executes the dynamic SQL statement.

**Introduction to Oracle9*i*: PL/SQL 14-11**

**Using the DBMS\_DDL Package**

**The DBMS\_DDL Package:**

• **Provides access to some SQL DDL statements**

**from stored procedures**

• **Includes some procedures:**

– **ALTER\_COMPILE (object\_type, owner, object\_name)**

**DBMS\_DDL.ALTER\_COMPILE(’PROCEDURE’,’A\_USER’,’QUERY\_EMP’)**

– **ANALYZE\_OBJECT (object\_type, owner, name,**

**method)**

**DBMS\_DDL.ANALYZE\_OBJECT(’TABLE’,’A\_USER’,’JOBS’,’COMPUTE’)**

**Note: This package runs with the privileges of calling user, rather than the package owner SYS.**

**14-12** Copyright © Oracle Corporation, 2001. All rights reserved.

**Using the DBMS\_DDL package**

This package provides access to some SQL DDL statements, which you can use in PL/SQL programs. DBMS\_DDL is not allowed in triggers, in procedures called from Forms Builder, or in remote sessions. This package runs with the privileges of calling user, rather than the package owner SYS.

**Practical Uses**

• You can recompile your modified PL/SQL program units by using DBMS\_DDL.ALTER\_COMPILE. The object type must be either procedure, function, package, package body, or trigger. • You can analyze a single object, using DBMS\_DDL.ANALYZE\_OBJECT. (There is a way of analyzing more than one object at a time, using DBMS\_UTILITY.) The object type should be TABLE, CLUSTER, or INDEX. The method must be COMPUTE, ESTIMATE, or DELETE.

• This package gives developers access to ALTER and ANALYZE SQL statements through PL/SQL environments.

**Introduction to Oracle9*i*: PL/SQL 14-12**

**Using DBMS\_JOB for Scheduling**

**DBMS\_JOB Enables the scheduling and execution of**

**PL/SQL programs:**

• **Submitting jobs**

• **Executing jobs**

• **Changing execution parameters of jobs**

• **Removing jobs**

• **Suspending Jobs**

**14-13** Copyright © Oracle Corporation, 2001. All rights reserved.

**Scheduling Jobs by Using DBMS\_JOB**

The package DBMS\_JOB is used to schedule PL/SQL programs to run. Using DBMS\_JOB, you can submit PL/SQL programs for execution, execute PL/SQL programs on a schedule, identify when PL/SQL programs should run, remove PL/SQL programs from the schedule, and suspend PL/SQL programs from running.

It can be used to schedule batch jobs during nonpeak hours or to run maintenance programs during times of low usage.

**Introduction to Oracle9*i*: PL/SQL 14-13**

**DBMS\_JOB Subprograms**

**Available subprograms include:**

• **SUBMIT**

• **REMOVE**

• **CHANGE**

• **WHAT**

• **NEXT\_DATE**

• **INTERVAL**

• **BROKEN**

• **RUN**

**14-14** Copyright © Oracle Corporation, 2001. All rights reserved.

**DBMS\_JOB Subprograms**

|  |  |
| --- | --- |
| **Subprogram** | **Description** |
| SUBMIT | Submits a job to the job queue |
| REMOVE | Removes a specified job from the job queue |
| CHANGE | Alters a specified job that has already been submitted to the job queue (you can alter the job description, the time at which the job will be run, or the interval between executions of the job) |
| WHAT | Alters the job description for a specified job |
| NEXT\_DATE | Alters the next execution time for a specified job |
| INTERVAL | Alters the interval between executions for a specified job |
| BROKEN | Disables job execution (if a job is marked as broken, the Oracle server does not attempt to execute it) |
| RUN | Forces a specified job to run |

**Introduction to Oracle9*i*: PL/SQL 14-14**

**Submitting Jobs**

**You can submit jobs by using DBMS\_JOB.SUBMIT.**

**Available parameters include:**

• **JOB OUT BINARY\_INTEGER**

• **WHAT IN VARCHAR2**

• **NEXT\_DATE IN DATE DEFAULT SYSDATE**

• **INTERVAL IN VARCHAR2 DEFAULT ’NULL’**

• **NO\_PARSE IN BOOLEAN DEFAULT FALSE**

**14-15** Copyright © Oracle Corporation, 2001. All rights reserved.

**DBMS\_JOB.SUBMIT Parameters**

The DBMS\_JOB.SUBMIT procedure adds a new job to the job queue. It accepts five parameters and returns the number of a job submitted through the OUT parameter JOB. The descriptions of the parameters are listed below.

|  |  |  |
| --- | --- | --- |
| **Parameter** | **M ode** | **Description** |
| JOB | OUT | Unique identifier of the job |
| WHAT | IN | PL/SQL code to execute as a job |
| NEXT\_DATE | IN | Next execution date of the job |
| INTERVAL | IN | Date function to compute the next execution date of a job |
| NO\_PARSE | IN | Boolean flag that indicates whether to parse the job at job submission (the default is false) |

**Note:** An exception is raised if the interval does not evaluate to a time in the future.

**Introduction to Oracle9*i*: PL/SQL 14-15**

**Submitting Jobs**

**Use DBMS\_JOB.SUBMIT to place a job to be executed in the job queue.**

**VARIABLE jobno NUMBER**

**BEGIN**

**DBMS\_JOB.SUBMIT (**

**job => :jobno,**

**what => ’OVER\_PACK.ADD\_DEPT(’’EDUCATION’’,2710);’,**

**next\_date => TRUNC(SYSDATE + 1),**

**interval => ’TRUNC(SYSDATE + 1)’**

**);**

**COMMIT;**

**END;**

**/**

**PRINT jobno**

****

**14-16** Copyright © Oracle Corporation, 2001. All rights reserved.

**Example**

The block of code in the preceding slide submits the ADD\_DEPT procedure of the OVER\_PACK package to the job queue. The job number is returned through the JOB parameter. The WHAT parameter must be enclosed in single quotation marks and must include a semicolon at the end of the text string. This job is submitted to run every day at midnight.

**Note:** In the example, the parameters are passed using named notation.

The transactions in the submitted job are not committed until either COMMIT is issued, or DBMS\_JOB.RUN is executed to run the job. The COMMIT in the slide commits the transaction.

**Introduction to Oracle9*i*: PL/SQL 14-16**