# Oracle® Database

PL/SQL Language Reference

11*g* Release 2 (11.2)

**E10472-02**

August 2009

ORACLE®

Oracle Database PL/SQL Language Reference, 11*g* Release 2 (11.2)

E10472-02

Copyright © 1996, 2009, Oracle and/or its affiliates. All rights reserved.

Primary Author: Sheila Moore

Contributing Author: Eric Belden

Contributors: S. Agrawal, C. Barclay, D. Bronnikov, S. Castledine, T. Chang, B. Cheng, R. Dani, R. Decker, C. Iyer, S. Kotsovolos, N. Le, W. Li, S. Lin, B. Llewellyn, D. Lorentz, V. Moore, K. Muthukkaruppan, C. Racicot, J. Russell, C. Wetherell, M. Vemulapati, G. Viswanathan, M. Yang

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

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

If this software or related documentation is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications which may create a risk of personal injury. If you use this software in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure the safe use of this software. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software in dangerous applications.

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

This software and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

# Contents

**Preface** ... xxix

Audience ... xxix
Documentation Accessibility ... xxix
Related Documents ... xxx
Conventions ... xxx
Syntax Descriptions ... xxxi

**What's New in PL/SQL?** ... xxxiii

PL/SQL Features for 11*g* Release 2 (11.2) ... xxxiii
PL/SQL Features for 11*g* Release 1 (11.1) ... xxxv

**1 Overview of PL/SQL**

**Advantages of PL/SQL** ... 1-1
Tight Integration with SQL ... 1-1
High Performance ... 1-2
High Productivity ... 1-2
Portability ... 1-3
Scalability ... 1-3
Manageability ... 1-3
Tight Security ... 1-3
Access to Predefined Packages ... 1-3
Support for Object-Oriented Programming ... 1-4
Support for Developing Web Applications ... 1-4
Support for Developing Server Pages ... 1-4
**Main Features of PL/SQL** ... 1-4
Blocks ... 1-5
Error Handling ... 1-6
Input and Output ... 1-6
Variables and Constants ... 1-7
Data Abstraction ... 1-7
Cursors ... 1-7
Composite Variables ... 1-7
%ROWTYPE Attribute ... 1-8
%TYPE Attribute ... 1-8
Abstract Data Types ... 1-8

Control Statements ..... 1-8
Subprograms ..... 1-9
Triggers ..... 1-9
Packages (APIs Written in PL/SQL) ..... 1-9
Conditional Compilation ..... 1-10
Processing a Query Result Set One Row at a Time ..... 1-10
**Architecture of PL/SQL** ..... 1-10
PL/SQL Engine ..... 1-11
PL/SQL Units and Compilation Parameters ..... 1-11

## 2 PL/SQL Language Fundamentals

**Character Sets** ..... 2-1
Database Character Set ..... 2-1
National Character Set ..... 2-3
**Lexical Units** ..... 2-3
Delimiters ..... 2-4
Identifiers ..... 2-5
Reserved Words and Keywords ..... 2-6
Predefined Identifiers ..... 2-6
User-Defined Identifiers ..... 2-7
Quoted Identifiers ..... 2-7
Literals ..... 2-8
Numeric Literals ..... 2-8
Character Literals ..... 2-8
String Literals ..... 2-9
BOOLEAN Literals ..... 2-10
Datetime and Interval Literals ..... 2-10
Comments ..... 2-10
Single-Line Comments ..... 2-10
Multiline Comments ..... 2-11
**Declarations** ..... 2-12
Variable Declarations ..... 2-12
Constant Declarations ..... 2-13
Initial Values of Variables and Constants ..... 2-13
NOT NULL Constraint ..... 2-14
%TYPE Attribute ..... 2-15
%ROWTYPE Attribute ..... 2-16
**References to Identifiers** ..... 2-17
**Scope and Visibility of Identifiers** ..... 2-18
**Assigning Values to Variables** ..... 2-22
Assigning Values to Variables with the Assignment Statement ..... 2-23
Assigning Values to Variables with the SELECT INTO Statement ..... 2-23
Assigning Values to Variables as Parameters of a Subprogram ..... 2-24
Assigning Values to BOOLEAN Variables ..... 2-25
**Expressions** ..... 2-25
Concatenation Operator ..... 2-26
Operator Precedence ..... 2-27

Logical Operators ........ 2-29
Short-Circuit Evaluation ........ 2-33
Comparison Operators ........ 2-34
Relational Operators ........ 2-34
IS [NOT] NULL Operator ........ 2-36
LIKE Operator ........ 2-36
BETWEEN Operator ........ 2-37
IN Operator ........ 2-38
BOOLEAN Expressions ........ 2-39
CASE Expressions ........ 2-39
Simple CASE Expression ........ 2-40
Searched CASE Expression ........ 2-41
SQL Functions in PL/SQL Expressions ........ 2-42
**Error-Reporting Functions** ........ 2-43
**Pragmas** ........ 2-43
**Conditional Compilation** ........ 2-43
How Conditional Compilation Works ........ 2-44
Preprocessor Control Tokens ........ 2-44
Selection Directives ........ 2-44
Error Directives ........ 2-45
Inquiry Directives ........ 2-45
Static Expressions ........ 2-48
Conditional Compilation Examples ........ 2-51
Retrieving and Printing Post-Processed Source Text ........ 2-53
Conditional Compilation Directive Restrictions ........ 2-53

## 3 PL/SQL Data Types

**Subtypes** ........ 3-1
**Predefined PL/SQL Scalar Data Types and Subtypes** ........ 3-2
Predefined PL/SQL Numeric Data Types and Subtypes ........ 3-2
PLS_INTEGER and BINARY_INTEGER Data Types ........ 3-3
SIMPLE_INTEGER Subtype of PLS_INTEGER ........ 3-3
BINARY_FLOAT and BINARY_DOUBLE Data Types ........ 3-5
NUMBER Data Type ........ 3-6
Predefined PL/SQL Character Data Types and Subtypes ........ 3-8
CHAR and VARCHAR2 Data Types ........ 3-8
RAW Data Type ........ 3-12
NCHAR and NVARCHAR2 Data Types ........ 3-12
LONG and LONG RAW Data Types ........ 3-14
ROWID and UROWID Data Types ........ 3-14
Predefined PL/SQL BOOLEAN Data Type ........ 3-15
Predefined PL/SQL Datetime and Interval Data Types ........ 3-15
DATE Data Type ........ 3-16
TIMESTAMP Data Type ........ 3-16
TIMESTAMP WITH TIME ZONE Data Type ........ 3-18
TIMESTAMP WITH LOCAL TIME ZONE Data Type ........ 3-18
INTERVAL YEAR TO MONTH Data Type ........ 3-19

INTERVAL DAY TO SECOND Data Type ..... 3-20
Datetime and Interval Arithmetic ..... 3-20
Avoiding Truncation Problems with Date and Time Subtypes ..... 3-21
**Predefined PL/SQL Large Object (LOB) Data Types** ..... 3-21
BFILE Data Type ..... 3-22
BLOB Data Type ..... 3-22
CLOB Data Type ..... 3-23
NCLOB Data Type ..... 3-23
**User-Defined PL/SQL Subtypes** ..... 3-23
**PL/SQL Data Type Conversion** ..... 3-26
Explicit Conversion ..... 3-27
Implicit Conversion ..... 3-27

## 4 PL/SQL Control Statements

**Overview of PL/SQL Control Statements** ..... 4-1
**Conditional Selection Statements** ..... 4-1
IF THEN Statement ..... 4-2
IF THEN ELSE Statement ..... 4-3
IF THEN ELSIF Statement ..... 4-5
Simple CASE Statement ..... 4-7
Searched CASE Statement ..... 4-7
**LOOP Statements** ..... 4-9
Basic LOOP Statement ..... 4-9
EXIT Statement ..... 4-10
EXIT WHEN Statement ..... 4-10
CONTINUE Statement ..... 4-11
CONTINUE WHEN Statement ..... 4-12
WHILE LOOP Statement ..... 4-13
FOR LOOP Statement ..... 4-14
FOR LOOP Index ..... 4-15
Lower Bound and Upper Bound ..... 4-18
EXIT Statement in FOR LOOP Statement ..... 4-19
**Sequential Control Statements** ..... 4-20
GOTO Statement ..... 4-21
NULL Statement ..... 4-23

## 5 PL/SQL Collections and Records

**PL/SQL Collection Type Characteristics** ..... 5-2
Associative Arrays ..... 5-3
Nested Tables ..... 5-5
Variable-Size Arrays (Varrays) ..... 5-6
**Collection Type Definitions** ..... 5-6
**Collection Variable Declarations** ..... 5-8
**Collection Initialization** ..... 5-10
**Collection Element References** ..... 5-12
**Assignments to Collections** ..... 5-13
**Collection Comparisons** ..... 5-18

**Multidimensional Collections** .......... 5-20
**Collection Methods** .......... 5-22
EXISTS Method .......... 5-23
COUNT Method .......... 5-23
LIMIT Method .......... 5-24
FIRST and LAST Methods .......... 5-24
PRIOR and NEXT Methods .......... 5-25
EXTEND Method .......... 5-27
TRIM Method .......... 5-28
DELETE Method .......... 5-30
**Collection Exceptions** .......... 5-31
**Record Definitions and Declarations** .......... 5-34
**Records as Subprogram Parameters and Function Return Values** .......... 5-36
**Assignments to Records** .......... 5-37
**Record Comparisons** .......... 5-39
**Inserting Records into Tables** .......... 5-40
**Updating Rows with Records** .......... 5-42
**Returning Rows into Records** .......... 5-43
**Restrictions on Record Inserts and Updates** .......... 5-43
**Assignments to Collections of Records** .......... 5-44

## 6 PL/SQL Static SQL

**Description of Static SQL** .......... 6-1
**Resolution of Names in Static SQL Statements** .......... 6-3
**SQL Pseudocolumns** .......... 6-3
CURRVAL and NEXTVAL .......... 6-3
LEVEL .......... 6-5
ROWID .......... 6-5
ROWNUM .......... 6-6
**Cursors** .......... 6-6
Implicit Cursors .......... 6-6
SQL%ISOPEN Attribute: Is the Cursor Open? .......... 6-7
SQL%FOUND Attribute: Were Any Rows Affected? .......... 6-7
SQL%NOTFOUND Attribute: Were No Rows Affected? .......... 6-8
SQL%ROWCOUNT Attribute: How Many Rows Were Affected? .......... 6-8
Explicit Cursors .......... 6-9
Declaring Explicit Cursors .......... 6-9
Opening Explicit Cursors .......... 6-10
Fetching Data with Explicit Cursors .......... 6-11
Fetching Bulk Data with Explicit Cursors .......... 6-13
Closing Explicit Cursors .......... 6-13
Explicit Cursor Attributes .......... 6-14
When Explicit Cursors Need Column Aliases .......... 6-17
Explicit Cursors that Accept Parameters .......... 6-18
**Queries** .......... 6-19
Selecting At Most One Row (SELECT INTO Statement) .......... 6-20
Selecting Multiple Rows (BULK COLLECT Clause) .......... 6-20

Looping Through Multiple Rows (Cursor FOR LOOP Statement) .... 6-21
Implicit Cursor FOR LOOP .... 6-21
Explicit Cursor FOR LOOP.... 6-22
Column Aliases for Expression Values in Cursor FOR LOOP.... 6-22
Processing Complicated Queries with Explicit Cursors.... 6-23
**Subqueries** .... 6-23
**Cursor Variables** .... 6-25
Purpose of Cursor Variables.... 6-26
Cursor Variable Creation .... 6-26
Cursor Variables as Subprogram Parameters .... 6-28
Cursor Variable Control (OPEN FOR, FETCH, and CLOSE Statements) .... 6-29
Opening Cursor Variables .... 6-29
Fetching from Cursor Variables.... 6-31
Closing Cursor Variables .... 6-33
INVALID_CURSOR Exception.... 6-33
Cursor Variables as Host Variables .... 6-33
Cursor Variable Restrictions.... 6-34
**Cursor Expressions**.... 6-35
**Transaction Processing and Control** .... 6-37
COMMIT Statement.... 6-38
ROLLBACK Statement .... 6-39
SAVEPOINT Statement.... 6-41
Implicit Rollbacks.... 6-42
SET TRANSACTION Statement .... 6-43
Overriding Default Locking .... 6-44
LOCK TABLE Statement.... 6-44
SELECT FOR UPDATE and FOR UPDATE Cursors.... 6-44
Simulating CURRENT OF Clause with ROWID Pseudocolumn .... 6-45
**Autonomous Transactions** .... 6-47
Advantages of Autonomous Transactions .... 6-48
Transaction Context.... 6-48
Transaction Visibility.... 6-48
Declaring Autonomous Transactions.... 6-49
Controlling Autonomous Transactions .... 6-50
Entering and Exiting.... 6-50
Committing and Rolling Back.... 6-51
Savepoints .... 6-51
Avoiding Errors with Autonomous Transactions.... 6-51
Autonomous Triggers .... 6-51
Invoking Autonomous Functions from SQL.... 6-53

## 7 PL/SQL Dynamic SQL

**When You Need Dynamic SQL** .... 7-1
**Native Dynamic SQL**.... 7-2
EXECUTE IMMEDIATE Statement.... 7-2
OPEN FOR, FETCH, and CLOSE Statements.... 7-4
Repeated Placeholder Names in Dynamic SQL Statements .... 7-5

Dynamic SQL Statement is Not Anonymous Block or CALL Statement .......... 7-5
Dynamic SQL Statement is Anonymous Block or CALL Statement .......... 7-6
**DBMS_SQL Package** .......... 7-6
DBMS_SQL.TO_REFCURSOR Function .......... 7-7
DBMS_SQL.TO_CURSOR_NUMBER Function .......... 7-8
**SQL Injection** .......... 7-9
SQL Injection Techniques .......... 7-10
Statement Modification .......... 7-10
Statement Injection .......... 7-11
Data Type Conversion .......... 7-13
Guarding Against SQL Injection .......... 7-14
Bind Arguments .......... 7-15
Validation Checks .......... 7-16
Explicit Format Models .......... 7-17

## 8 PL/SQL Subprograms

**Overview of PL/SQL Subprograms** .......... 8-1
**Subprogram Parts** .......... 8-3
**Nested Subprograms that Invoke Each Other** .......... 8-6
**Subprogram Parameters** .......... 8-6
Formal and Actual Subprogram Parameters .......... 8-6
Subprogram Parameter Passing Methods .......... 8-7
Subprogram Parameter Modes .......... 8-8
IN Mode .......... 8-8
OUT Mode .......... 8-8
IN OUT Mode .......... 8-9
Summary of Subprogram Parameter Modes .......... 8-10
Initial Values for Subprogram Parameters .......... 8-10
Positional, Named, and Mixed Notation for Actual Parameters .......... 8-12
**Overloaded Subprograms** .......... 8-13
Subprograms that You Can Overload .......... 8-14
Subprograms that You Cannot Overload .......... 8-15
Subprogram Overload Errors .......... 8-15
**Subprogram Call Resolution** .......... 8-17
**Invoker's Rights and Definer's Rights (AUTHID Property)** .......... 8-19
Choosing AUTHID CURRENT_USER or AUTHID DEFINER .......... 8-21
AUTHID and SQL Command SET ROLE .......... 8-21
Need for Template Objects in IR Units .......... 8-21
Overriding Default Name Resolution in IR Units .......... 8-21
IR Subprograms, Views, and Database Triggers .......... 8-22
IR Database Links .......... 8-22
IR ADTs .......... 8-22
IR Instance Methods .......... 8-24
**Recursive Subprograms** .......... 8-25
**External Subprograms** .......... 8-25
**Subprogram Side Effects** .......... 8-27
**Subprogram Parameter Aliasing** .......... 8-27

**PL/SQL Function Result Cache** .... 8-29
Enabling Result-Caching for a Function .... 8-30
Developing Applications with Result-Cached Functions .... 8-31
Restrictions on Result-Cached Functions .... 8-31
Examples of Result-Cached Functions .... 8-32
Result-Cached Application Configuration Parameters .... 8-32
Result-Cached Recursive Function .... 8-34
Advanced Result-Cached Function Topics .... 8-34
Rules for a Cache Hit .... 8-34
Result Cache Bypass .... 8-35
Making Result-Cached Functions Handle Session-Specific Settings .... 8-35
Making Result-Cached Functions Handle Session-Specific Application Contexts .... 8-36
Choosing Result-Caching Granularity .... 8-37
Result Caches in Oracle RAC Environment .... 8-38
Result Cache Management .... 8-39
Hot-Patching PL/SQL Units on Which Result-Cached Functions Depend .... 8-40

## 9 PL/SQL Triggers

**Overview of Triggers** .... 9-1
Trigger Types .... 9-2
OLD and NEW Pseudorecords .... 9-2
Trigger States .... 9-3
Data Access for Triggers .... 9-3
Trigger Uses .... 9-3
**Trigger Design Guidelines** .... 9-4
**Trigger Creation** .... 9-5
Trigger Names .... 9-6
Triggering Statements .... 9-6
SQL*Loader and Import .... 9-6
How Column Lists Affect UPDATE Triggers .... 9-7
BEFORE and AFTER Triggers .... 9-7
INSTEAD OF Triggers .... 9-8
Views that Require INSTEAD OF Triggers .... 9-9
Triggers on Nested Table View Columns .... 9-9
INSTEAD OF Trigger Example .... 9-12
FOR EACH ROW Triggers .... 9-13
Conditional Triggers .... 9-14
Compound Triggers .... 9-14
Compound Trigger Uses .... 9-15
Compound Trigger Sections .... 9-16
Triggering Statements of Compound Triggers .... 9-16
Compound Trigger Restrictions .... 9-17
Compound Trigger Example .... 9-17
Using Compound Triggers to Avoid Mutating-Table Error .... 9-19
Order in Which Triggers Fire .... 9-20
**Trigger Body** .... 9-21
Column Values Accessible to Row Triggers .... 9-22

Example: Trigger That Modifies LOB Columns ........ 9-23
INSTEAD OF Triggers on Nested Table View Columns ........ 9-23
Trigger Name Conflicts ........ 9-23
Conditional Predicates for Detecting the DML Operation that Fired a Trigger ........ 9-24
Exceptions Raised in the Trigger Body ........ 9-24
OBJECT_VALUE Pseudocolumn ........ 9-25
Remote Exception Handling ........ 9-26
Trigger Body Restrictions ........ 9-27
Size Restriction ........ 9-27
SQL Statement Restrictions ........ 9-28
LONG and LONG RAW Data Type Restrictions ........ 9-28
Mutating Table Restriction ........ 9-28
Restrictions on Mutating Tables Relaxed ........ 9-29
System Trigger Restrictions ........ 9-30
**Trigger Compilation** ........ 9-31
Trigger Dependencies ........ 9-31
Trigger Recompilation ........ 9-31
**Trigger Replacement** ........ 9-32
**Trigger Debugging** ........ 9-32
**Trigger Enabling** ........ 9-32
**Trigger Disabling** ........ 9-32
**Views for Information About Triggers** ........ 9-33
**Trigger Application Examples** ........ 9-34
Triggers for Auditing ........ 9-34
Triggers and Constraints ........ 9-38
Triggers for Ensuring Referential Integrity ........ 9-39
Foreign Key Trigger for Child Table ........ 9-40
UPDATE and DELETE RESTRICT Triggers for Parent Table ........ 9-41
UPDATE and DELETE SET NULL Triggers for Parent Table ........ 9-42
DELETE Cascade Trigger for Parent Table ........ 9-42
UPDATE Cascade Trigger for Parent Table ........ 9-43
Triggers for Complex Check Constraints ........ 9-44
Triggers for Complex Security Authorizations ........ 9-45
Triggers for Transparent Event Logging ........ 9-46
Triggers for Deriving Column Values ........ 9-46
Triggers for Building Complex Updatable Views ........ 9-47
Triggers for Fine-Grained Access Control ........ 9-50
**Triggers that Publish Database Events** ........ 9-50
How Triggers Publish Events ........ 9-51
Publication Context ........ 9-51
Error Handling ........ 9-52
Execution Model ........ 9-52
Event Attribute Functions ........ 9-52
Database Events ........ 9-55
Client Events ........ 9-56

## 10 PL/SQL Packages

**What is a Package?** ........ 10-1
**What Goes in a Package?** ........ 10-2
**Advantages of Packages** ........ 10-3
**Serially Reusable Packages** ........ 10-3
Package States ........ 10-4
Why Serially Reusable Packages? ........ 10-4
Syntax of Serially Reusable Packages ........ 10-5
Semantics of Serially Reusable Packages ........ 10-5
Examples of Serially Reusable Packages ........ 10-5
**Package Specification** ........ 10-8
**Referencing Package Contents** ........ 10-9
**Package Body** ........ 10-10
**Examples of Package Features** ........ 10-11
**Private and Public Items in Packages** ........ 10-15
**How STANDARD Package Defines the PL/SQL Environment** ........ 10-16
**Overview of Product-Specific Packages** ........ 10-16
DBMS_ALERT Package ........ 10-16
DBMS_OUTPUT Package ........ 10-17
DBMS_PIPE Package ........ 10-17
DBMS_CONNECTION_POOL Package ........ 10-18
HTF and HTP Packages ........ 10-18
UTL_FILE Package ........ 10-18
UTL_HTTP Package ........ 10-18
UTL_SMTP Package ........ 10-18
**Packing Writing Guidelines** ........ 10-18
**Separating Cursor Specifications and Bodies with Packages** ........ 10-19

## 11 PL/SQL Error Handling

**Compile-Time Warnings** ........ 11-1
**DBMS_WARNING Package** ........ 11-3
**Overview of Exception Handling** ........ 11-4
Advantages of Exceptions ........ 11-5
Guidelines for Avoiding and Handling Exceptions ........ 11-6
**Predefined Exceptions** ........ 11-7
**User-Defined Exceptions** ........ 11-9
Exception Declarations ........ 11-9
Exception Scope ........ 11-9
Associating Exceptions with Numbers ........ 11-10
User-Defined Error Messages ........ 11-11
Redeclaring Predefined Exceptions ........ 11-12
**How Exceptions Are Raised** ........ 11-13
**How Exceptions Propagate** ........ 11-13
**Reraising an Exception** ........ 11-15
**Exception Handlers** ........ 11-16
Exceptions Raised in Declarations ........ 11-17
Exceptions Raised in Exception Handlers ........ 11-18

Transferring Control to or from Exception Handlers .......... 11-18
Error Code and Error Message Retrieval .......... 11-18
Continuing Execution After an Exception Is Raised .......... 11-19
Retrying a Transaction .......... 11-20
Identifying Exception Locations With Locator Variables .......... 11-22
**Unhandled Exceptions** .......... 11-22

## 12 PL/SQL Optimization and Tuning

**PL/SQL Optimizer** .......... 12-1
**Subprogram Inlining** .......... 12-2
**PL/SQL Code to Consider Tuning** .......... 12-3
**Avoiding CPU Overhead in PL/SQL Code** .......... 12-4
Make SQL Statements as Efficient as Possible .......... 12-4
Make Function Calls as Efficient as Possible .......... 12-4
Make Loops as Efficient as Possible .......... 12-5
Use SQL String Functions .......... 12-6
Put Least Expensive Conditional Tests First .......... 12-6
Minimize Implicit Data Type Conversion .......... 12-6
Avoid NUMBER Data Type and Constrained Subtypes .......... 12-7
Recommended Data Types for Integer Arithmetic .......... 12-7
Recommended Data Types for Floating-Point Arithmetic .......... 12-7
**Avoiding Memory Overhead in PL/SQL Code** .......... 12-8
Declare VARCHAR2 Variables of 4000 or More Characters .......... 12-8
Group Related Subprograms into Packages .......... 12-8
Pin Packages in the Shared Memory Pool .......... 12-8
Apply Advice of Compiler Warnings .......... 12-8
**Collecting Data About User-Defined Identifiers** .......... 12-8
**Profiling and Tracing PL/SQL Programs** .......... 12-9
Profiler API: Package DBMS_PROFILER .......... 12-9
Trace API: Package DBMS_TRACE .......... 12-10
**Reducing Loop Overhead with Bulk SQL** .......... 12-10
Running One DML Statement Multiple Times (FORALL Statement) .......... 12-11
Effect of FORALL Exceptions on Rollbacks .......... 12-16
Handling FORALL Exceptions .......... 12-17
Counting Rows Affected by FORALL .......... 12-19
Retrieving Query Results into Collections .......... 12-21
Examples of Bulk Fetching from a Cursor .......... 12-23
Limiting Rows for a Bulk FETCH Operation (LIMIT Clause) .......... 12-25
Retrieving DML Results Into a Collection (RETURNING INTO Clause) .......... 12-27
Using FORALL and BULK COLLECT Together .......... 12-27
Host Arrays with Bulk Binds .......... 12-28
SELECT BULK COLLECT INTO Statements and Aliasing .......... 12-29
**Computation-Intensive PL/SQL Programs** .......... 12-33
**Tuning Dynamic SQL with EXECUTE IMMEDIATE Statement and Cursor Variables** .......... 12-34
**Tuning PL/SQL Subprogram Calls with NOCOPY Hint** .......... 12-34
**Compiling PL/SQL Units for Native Execution** .......... 12-36
Determining Whether to Use PL/SQL Native Compilation .......... 12-37

How PL/SQL Native Compilation Works ..... 12-37
Dependencies, Invalidation, and Revalidation ..... 12-38
Setting Up a New Database for PL/SQL Native Compilation ..... 12-38
Compiling the Entire Database for PL/SQL Native or Interpreted Compilation ..... 12-38
**Performing Multiple Transformations with Pipelined Table Functions** ..... 12-40
Overview of Pipelined Table Functions ..... 12-41
Writing a Pipelined Table Function ..... 12-42
Pipelined Table Functions for Transformations ..... 12-43
Returning Results from Pipelined Table Functions ..... 12-44
Pipelining Data Between PL/SQL Table Functions ..... 12-45
Optimizing Multiple Calls to Pipelined Table Functions ..... 12-45
Fetching from Results of Pipelined Table Functions ..... 12-45
Passing Data with Cursor Variables ..... 12-46
Performing DML Statements Inside Pipelined Table Functions ..... 12-49
Performing DML Statements on Pipelined Table Functions ..... 12-49
Exception Handlers in Pipelined Table Functions ..... 12-50
**Updating Large Tables in Parallel** ..... 12-50

## 13 PL/SQL Language Elements

**Assignment Statement** ..... 13-3
**AUTONOMOUS_TRANSACTION Pragma** ..... 13-6
**Block** ..... 13-8
**Basic LOOP Statement** ..... 13-16
**CASE Statement** ..... 13-18
**CLOSE Statement** ..... 13-21
**Collection** ..... 13-23
**Collection Method Call** ..... 13-27
**Comment** ..... 13-30
**Constant** ..... 13-32
**CONTINUE Statement** ..... 13-35
**Cursor FOR LOOP Statement** ..... 13-37
**Cursor Variable** ..... 13-39
**DELETE Statement Extension** ..... 13-42
**EXCEPTION_INIT Pragma** ..... 13-43
**Exception** ..... 13-45
**Exception Handler** ..... 13-47
**EXECUTE IMMEDIATE Statement** ..... 13-49
**EXIT Statement** ..... 13-52
**Explicit Cursor** ..... 13-54
**Expression** ..... 13-58
**FETCH Statement** ..... 13-67
**FOR LOOP Statement** ..... 13-70
**FORALL Statement** ..... 13-73
**Function** ..... 13-76
**GOTO Statement** ..... 13-80
**IF Statement** ..... 13-82
**Implicit Cursor Attribute** ..... 13-84

INLINE Pragma ........ 13-87
INSERT Statement Extension ........ 13-89
Literal ........ 13-91
Named Cursor Attribute ........ 13-93
NULL Statement ........ 13-96
OPEN Statement ........ 13-97
OPEN FOR Statement ........ 13-98
Parameter ........ 13-101
Procedure ........ 13-103
RAISE Statement ........ 13-105
Record ........ 13-106
RESTRICT_REFERENCES Pragma ........ 13-109
RETURN Statement ........ 13-111
RETURNING INTO Clause ........ 13-113
%ROWTYPE Attribute ........ 13-116
SELECT INTO Statement ........ 13-118
SERIALLY_REUSABLE Pragma ........ 13-122
SQLCODE Function ........ 13-124
SQLERRM Function ........ 13-125
%TYPE Attribute ........ 13-127
UPDATE Statement Extensions ........ 13-129
Variable ........ 13-131
WHILE LOOP Statement ........ 13-133

## 14 SQL Statements for Stored PL/SQL Units

ALTER FUNCTION Statement ........ 14-3
ALTER LIBRARY Statement ........ 14-6
ALTER PACKAGE Statement ........ 14-8
ALTER PROCEDURE Statement ........ 14-11
ALTER TRIGGER Statement ........ 14-14
ALTER TYPE Statement ........ 14-17
CREATE FUNCTION Statement ........ 14-32
CREATE LIBRARY Statement ........ 14-40
CREATE PACKAGE Statement ........ 14-42
CREATE PACKAGE BODY Statement ........ 14-45
CREATE PROCEDURE Statement ........ 14-49
CREATE TRIGGER Statement ........ 14-53
CREATE TYPE Statement ........ 14-69
CREATE TYPE BODY Statement ........ 14-86
DROP FUNCTION Statement ........ 14-91
DROP LIBRARY Statement ........ 14-93
DROP PACKAGE Statement ........ 14-94
DROP PROCEDURE Statement ........ 14-96
DROP TRIGGER Statement ........ 14-98
DROP TYPE Statement ........ 14-99
DROP TYPE BODY Statement ........ 14-101

## A PL/SQL Source Code Wrapping


**Overview of Wrapping** .......... A-1
**Guidelines for Wrapping** .......... A-1
**Limitations of Wrapping** .......... A-2
**Wrapping PL/SQL Code with wrap Utility** .......... A-2
Input and Output Files for the PL/SQL wrap Utility .......... A-3
Running the wrap Utility .......... A-3
Limitations of the wrap Utility .......... A-4
**Wrapping PL/QL Code with DBMS_DDL Subprograms** .......... A-4
DBMS_DDL.CREATE_WRAPPED Procedure .......... A-5
Limitation of the DBMS_DDL.WRAP Function .......... A-6


## B PL/SQL Name Resolution


**What is Name Resolution?** .......... B-1
Name Resolution .......... B-2
**Examples of Qualified Names and Dot Notation** .......... B-4
**How Name Resolution Differs in PL/SQL and SQL** .......... B-5
**What is Capture?** .......... B-6
Inner Capture .......... B-6
Same-Scope Capture .......... B-7
Outer Capture .......... B-7
**Avoiding Inner Capture in DML Statements** .......... B-8
Qualifying References to Attributes and Methods .......... B-8
Qualifying References to Row Expressions .......... B-9


## C PL/SQL Program Limits

## D PL/SQL Reserved Words and Keywords

## Index

## List of Examples


1–1 PL/SQL Block Structure ........ 1-5
1–2 Processing Query Result Rows One at a Time ........ 1-10
2–1 Delimiter with Embedded Space ........ 2-3
2–2 Whitespace Characters Improving Source Code Readability ........ 2-4
2–3 Identifiers Not Separated by Space or Punctuation ........ 2-6
2–4 Datetime and Interval Literals ........ 2-10
2–5 Single-Line Comments ........ 2-11
2–6 Multiline Comments ........ 2-11
2–7 Variable Declarations ........ 2-13
2–8 Constant Declarations ........ 2-13
2–9 Variable and Constant Declarations with Initial Values ........ 2-14
2–10 Variable Initialized to NULL by Default ........ 2-14
2–11 Variable Declaration with NOT NULL Constraint ........ 2-14
2–12 Variables Initialized to NULL Values ........ 2-15
2–13 Declaring Variable of Same Type as Database Column ........ 2-15
2–14 Declaring Variable of Same Type as Another Variable ........ 2-16
2–15 Declaring Record that Represents Table Row ........ 2-16
2–16 Declaring a Record that Represents a Subset of Table Columns ........ 2-17
2–17 Declaring a Record that Represents a Row from a Join ........ 2-17
2–18 Scope and Visibility of Identifiers ........ 2-18
2–19 Qualifying a Redeclared Global Identifier with a Block Label ........ 2-19
2–20 Qualifying an Identifier with a Subprogram Name ........ 2-19
2–21 Duplicate Identifiers in Same Scope ........ 2-20
2–22 Declaring the Same Identifier in Two Different Units ........ 2-21
2–23 Label and Subprogram with Same Name in Same Scope ........ 2-21
2–24 Block with Multiple and Duplicate Labels ........ 2-22
2–25 Assigning Values to Variables with Assignment Statement ........ 2-23
2–26 SELECT INTO Assigns Values to Variables ........ 2-23
2–27 Assigning Values to Variables as Parameters of a Subprogram ........ 2-24
2–28 Assigning BOOLEAN Values ........ 2-25
2–29 Concatenation Operator ........ 2-26
2–30 Concatenation Operator with NULL Operands ........ 2-26
2–31 Controlling Evaluation Order with Parentheses ........ 2-27
2–32 Expression with Nested Parentheses ........ 2-27
2–33 Improving Readability with Parentheses ........ 2-28
2–34 Operator Precedence ........ 2-28
2–35 AND Operator ........ 2-29
2–36 OR Operator ........ 2-30
2–37 NOT Operator ........ 2-31
2–38 NULL Value in Unequal Comparison ........ 2-32
2–39 NULL Value in Equal Comparison ........ 2-32
2–40 NOT NULL Equals NULL ........ 2-33
2–41 Changing Evaluation Order of Logical Operators ........ 2-33
2–42 Short-Circuit Evaluation ........ 2-33
2–43 Relational Operators in Expressions ........ 2-35
2–44 LIKE Operator in Expression ........ 2-36
2–45 Escape Character in Pattern ........ 2-37
2–46 BETWEEN Operator in Expressions ........ 2-37
2–47 IN Operator in Expressions ........ 2-38
2–48 IN Operator with Sets with NULL Values ........ 2-38
2–49 Equivalent BOOLEAN Expressions as Conditions in Loops ........ 2-39
2–50 Simple CASE Expression ........ 2-40
2–51 Simple CASE Expression with WHEN NULL ........ 2-40
2–52 Searched CASE Expression ........ 2-41

2–53 Searched CASE Expression with WHEN condition IS NULL .......... 2-42
2–54 Predefined Inquiry Directives $$PLSQL_LINE and $$PLSQL_UNIT .......... 2-46
2–55 Displaying Values of PL/SQL Compilation Parameters .......... 2-47
2–56 PLSQL_CCFLAGS Assigns Value to Itself .......... 2-48
2–57 Static Constants .......... 2-50
2–58 Code for Checking Database Version .......... 2-52
2–59 Compiling Different Code for Different Database Versions .......... 2-52
2–60 Displaying Post-Processed Source Code .......... 2-53
3–1 Comparing Two CHAR Values .......... 3-11
3–2 Comparing Two VARCHAR2 Values .......... 3-11
3–3 Comparing CHAR Value and VARCHAR2 Value .......... 3-11
3–4 Assigning a Literal Value to a TIMESTAMP Variable .......... 3-17
3–5 SCN_TO_TIMESTAMP and TIMESTAMP_TO_SCN Functions .......... 3-17
3–6 Assigning a Literal to a TIMESTAMP WITH TIME ZONE Variable .......... 3-18
3–7 Correct Assignment to TIMESTAMP WITH LOCAL TIME ZONE .......... 3-19
3–8 Incorrect Assignment to TIMESTAMP WITH LOCAL TIME ZONE .......... 3-19
3–9 Assigning Literals to an INTERVAL YEAR TO MONTH Variable .......... 3-20
3–10 Assigning Literals to an INTERVAL DAY TO SECOND Variable .......... 3-20
3–11 Defining Subtypes .......... 3-23
3–12 Declaring Variables of User-Defined Subtype .......... 3-23
3–13 Ranges with Subtypes .......... 3-24
3–14 Type Compatibility with the NUMBER Data Type .......... 3-24
3–15 Assigning Initial Value to Subtype Variable .......... 3-25
3–16 Subtype Constraints Inherited by Subprograms .......... 3-25
3–17 Column Constraints Inherited by Subtypes .......... 3-26
3–18 Implicit Conversion .......... 3-27
4–1 IF THEN Statement .......... 4-2
4–2 IF THEN ELSE Statement .......... 4-3
4–3 Nested IF THEN ELSE Statements .......... 4-4
4–4 IF THEN ELSIF Statement .......... 4-5
4–5 IF THEN ELSIF Statement that Simulates Simple CASE Statement .......... 4-6
4–6 Simple CASE Statement .......... 4-7
4–7 Searched CASE Statement .......... 4-8
4–8 EXCEPTION Instead of ELSE Clause in CASE Statement .......... 4-8
4–9 Basic LOOP Statement with EXIT Statement .......... 4-10
4–10 Basic LOOP Statement with EXIT WHEN Statement .......... 4-10
4–11 Nested, Labeled Basic LOOP Statements with EXIT WHEN Statements .......... 4-11
4–12 CONTINUE Statement in Basic LOOP Statement .......... 4-12
4–13 CONTINUE WHEN Statement in Basic LOOP Statement .......... 4-12
4–14 WHILE LOOP Statements .......... 4-13
4–15 FOR LOOP Statements .......... 4-14
4–16 Reverse FOR LOOP Statements .......... 4-15
4–17 FOR LOOP Statement Tries to Change Index Value .......... 4-16
4–18 Statement Outside FOR LOOP Tries to Reference Index .......... 4-16
4–19 FOR LOOP Index with Same Name as Declared Variable .......... 4-16
4–20 FOR LOOP References Declared Variable with Same Name as Index .......... 4-17
4–21 Nested FOR LOOP Statements with Same Index Name .......... 4-17
4–22 Several Types of FOR LOOP Bounds .......... 4-18
4–23 Changing the Increment of the Counter in a FOR LOOP Statement .......... 4-18
4–24 Specifying a LOOP Range at Run Time .......... 4-18
4–25 FOR LOOP with Lower Bound > Upper Bound .......... 4-19
4–26 EXIT in FOR LOOP .......... 4-20
4–27 EXIT with Label in FOR LOOP .......... 4-20
4–28 GOTO Statement .......... 4-21
4–29 Incorrect Label Placement .......... 4-21

4–30 NULL Statement Allows GOTO to Label ..... 4-22
4–31 GOTO Statement Transfers Control to Enclosing Block ..... 4-22
4–32 GOTO Statement Cannot Transfer Control into IF Statement ..... 4-22
4–33 NULL Statement Showing No Action ..... 4-23
4–34 NULL Statement as Placeholder During Subprogram Creation ..... 4-24
4–35 NULL Statement in WHEN OTHER Clause ..... 4-24
5–1 Declaring and Populating Associative Array Indexed by String ..... 5-3
5–2 Declaring an Associative Array ..... 5-7
5–3 Declaring Nested Tables, Varrays, and Associative Arrays ..... 5-8
5–4 Declaring Collections with %TYPE ..... 5-8
5–5 Declaring a Procedure Parameter as a Nested Table ..... 5-9
5–6 Invoking a Procedure with a Nested Table Parameter ..... 5-9
5–7 Specifying Collection Element Types with %TYPE and %ROWTYPE ..... 5-9
5–8 VARRAY of Records ..... 5-10
5–9 NOT NULL Constraint on Collection Elements ..... 5-10
5–10 Constructor for a Nested Table ..... 5-11
5–11 Constructor for a Varray ..... 5-11
5–12 Collection Constructor Including Null Elements ..... 5-11
5–13 Combining Collection Declaration and Constructor ..... 5-11
5–14 Empty Varray Constructor ..... 5-12
5–15 Referencing a Nested Table Element ..... 5-12
5–16 Referencing an Element of an Associative Array ..... 5-13
5–17 Data Type Compatibility for Collection Assignment ..... 5-14
5–18 Assigning a Null Value to a Nested Table ..... 5-14
5–19 Assigning Nested Tables with Set Operators ..... 5-15
5–20 Assigning Values to VARRAYs with Complex Data Types ..... 5-16
5–21 Assigning Values to Tables with Complex Data Types ..... 5-17
5–22 Checking if a Collection Is Null ..... 5-18
5–23 Comparing Two Nested Tables ..... 5-19
5–24 Comparing Nested Tables with Set Operators ..... 5-19
5–25 Multilevel VARRAY ..... 5-20
5–26 Multilevel Nested Table ..... 5-21
5–27 Multilevel Associative Array ..... 5-21
5–28 Checking Whether a Collection Element EXISTS ..... 5-23
5–29 Counting Collection Elements with COUNT ..... 5-23
5–30 Checking the Maximum Size of a Collection with LIMIT ..... 5-24
5–31 FIRST and LAST Methods ..... 5-25
5–32 PRIOR and NEXT Methods ..... 5-26
5–33 NEXT Method Accesses Elements of Nested Table ..... 5-26
5–34 EXTEND Method ..... 5-27
5–35 TRIM Method ..... 5-29
5–36 TRIM Method on Deleted Elements ..... 5-30
5–37 DELETE Method ..... 5-30
5–38 Collection Exceptions ..... 5-31
5–39 How Invalid Subscripts are Handled with DELETE(n) ..... 5-33
5–40 Incompatibility Between Package and Local Collection Types ..... 5-33
5–41 Declaring and Initializing a Simple Record Type ..... 5-34
5–42 Declaring and Initializing Record Types ..... 5-34
5–43 %ROWTYPE in Record Declaration ..... 5-35
5–44 Returning a Record from a Function ..... 5-36
5–45 Record as Procedure Parameter ..... 5-36
5–46 Nested Record Declaration ..... 5-37
5–47 Assigning Initial Values to a Record ..... 5-37
5–48 Assigning All the Fields of a Record in One Statement ..... 5-38
5–49 Assigning a %ROWTYPE Record to a User-Defined Record ..... 5-38

5–50 SELECT INTO Assigns Values to Record ........ 5-39
5–51 Testing Records for Nullity and Equality........ 5-39
5–52 Initializing a Table by Inserting a Record of Default Values........ 5-41
5–53 Updating Rows with a Record........ 5-42
5–54 RETURNING INTO Clause with Record ........ 5-43
5–55 BULK COLLECT with SELECT INTO Statement ........ 5-44
6–1 Static SQL Statements........ 6-2
6–2 CURRVAL and NEXTVAL Pseudocolumns ........ 6-4
6–3 SQL%FOUND Attribute ........ 6-7
6–4 SQL%ROWCOUNT Attribute ........ 6-8
6–5 Explicit Cursor Declaration ........ 6-9
6–6 Opening an Explicit Cursor........ 6-10
6–7 Fetching with a Cursor........ 6-11
6–8 Referencing PL/SQL Variables in Its Scope........ 6-12
6–9 Fetching the Same Cursor Into Different Variables ........ 6-12
6–10 Fetching Bulk Data with a Cursor ........ 6-13
6–11 %ISOPEN Attribute ........ 6-14
6–12 %FOUND Attribute........ 6-14
6–13 %NOTFOUND Attribute ........ 6-15
6–14 %ROWCOUNT Attribute ........ 6-16
6–15 Explicit Cursor with Calculated Column that Needs Alias........ 6-17
6–16 Passing Parameters to a Cursor FOR LOOP ........ 6-18
6–17 Passing Parameters to Explicit Cursors ........ 6-19
6–18 Implicit Cursor FOR Loop ........ 6-21
6–19 Explicit Cursor FOR LOOP........ 6-22
6–20 Alias for Expressions in Query ........ 6-22
6–21 Subqueries in Cursor Declarations........ 6-24
6–22 Subquery in FROM Clause ........ 6-24
6–23 Correlated Subquery ........ 6-25
6–24 Cursor Variable Declarations ........ 6-27
6–25 Cursor Variables Returning %ROWTYPE Variables........ 6-27
6–26 %ROWTYPE Attribute Provides Data Type ........ 6-27
6–27 Cursor Variable Returning a Record Type........ 6-28
6–28 Cursor Variable as Parameter ........ 6-28
6–29 Opening Cursor Variable........ 6-30
6–30 Stored Procedure to Open a Cursor Variable ........ 6-30
6–31 Stored Procedure to Open Cursor Variables with Different Queries........ 6-30
6–32 Cursor Variable with Different Return Types ........ 6-31
6–33 Fetching from Cursor Variable into Record........ 6-31
6–34 Fetching from Cursor Variable into Collections........ 6-32
6–35 Cursor Variable as Host Variable ........ 6-33
6–36 Reducing Network Traffic When Passing Host Variables........ 6-34
6–37 Declaration of Cursor Variables in a Package ........ 6-35
6–38 Cursor Expression........ 6-35
6–39 COMMIT Statement with COMMENT and WRITE Clauses ........ 6-38
6–40 ROLLBACK Statement........ 6-39
6–41 SAVEPOINT and ROLLBACK Statements ........ 6-41
6–42 Reusing a SAVEPOINT with ROLLBACK........ 6-42
6–43 SET TRANSACTION Statement in Read-Only Transaction ........ 6-43
6–44 FOR UPDATE Cursor in CURRENT OF Clause of UPDATE Statement ........ 6-45
6–45 SELECT FOR UPDATE with Multiple Tables ........ 6-45
6–46 Trying to Fetch with FOR UPDATE Cursor After COMMIT Statement ........ 6-46
6–47 Simulating CURRENT OF Clause with ROWID Pseudocolumn ........ 6-46
6–48 Declaring an Autonomous Function in a Package........ 6-49
6–49 Declaring an Autonomous Standalone Procedure........ 6-49

6–50 Declaring an Autonomous PL/SQL Block ........ 6-50
6–51 Autonomous Trigger the Logs INSERT Statements ........ 6-51
6–52 Autonomous Trigger Using Native Dynamic SQL for DDL ........ 6-53
6–53 Invoking an Autonomous Function ........ 6-54
7–1 Invoking a Subprogram from a Dynamic PL/SQL Block ........ 7-3
7–2 Unsupported Data Type in Native Dynamic SQL ........ 7-4
7–3 Uninitialized Variable for NULL in USING Clause ........ 7-4
7–4 Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements ........ 7-5
7–5 Repeated Placeholder Names in Dynamic PL/SQL Block ........ 7-6
7–6 Switching from DBMS_SQL Package to Native Dynamic SQL ........ 7-7
7–7 Switching from Native Dynamic SQL to DBMS_SQL Package ........ 7-8
7–8 Setup for SQL Injection Examples ........ 7-9
7–9 Procedure Vulnerable to Statement Modification ........ 7-10
7–10 Procedure Vulnerable to Statement Injection ........ 7-11
7–11 Procedure Vulnerable to SQL Injection Through Data Type Conversion ........ 7-13
7–12 Bind Arguments Guarding Against SQL Injection ........ 7-15
7–13 Validation Checks Guarding Against SQL Injection ........ 7-16
7–14 Explicit Format Models Guarding Against SQL Injection ........ 7-18
8–1 Declaring, Defining, and Invoking a Simple PL/SQL Procedure ........ 8-3
8–2 Declaring, Defining, and Invoking a Simple PL/SQL Function ........ 8-5
8–3 Creating Nested Subprograms that Invoke Each Other ........ 8-6
8–4 Formal Parameters and Actual Parameters ........ 8-7
8–5 Avoiding and Causing Implicit Conversion of Actual Parameters ........ 8-7
8–6 Subprogram Parameter Mode OUT ........ 8-8
8–7 Procedure with Initial Parameter Values ........ 8-10
8–8 Formal Parameter with Expression as Initial Value ........ 8-11
8–9 Subprogram Calls Using Positional, Named, and Mixed Notation ........ 8-12
8–10 Overloaded Subprogram ........ 8-13
8–11 Overload Error That Causes Compile-Time Error ........ 8-15
8–12 Overload Error That Compiles Successfully ........ 8-16
8–13 Invocation of Improperly Overloaded Subprogram ........ 8-16
8–14 Properly Overloaded Subprogram ........ 8-16
8–15 Invocation of Properly Overloaded Subprogram ........ 8-16
8–16 Package Specification Without Overload Errors ........ 8-16
8–17 Improper Invocation of Properly Overloaded Subprogram ........ 8-17
8–18 Resolving PL/SQL Procedure Names ........ 8-18
8–19 Creating an ADT with AUTHID CURRENT USER ........ 8-22
8–20 Invoking an IR Instance Method ........ 8-24
8–21 Invoking an External Procedure from PL/SQL ........ 8-26
8–22 Invoking a Java Function from PL/SQL ........ 8-26
8–23 Aliasing from Passing Global Variable with NOCOPY Hint ........ 8-27
8–24 Aliasing Passing Same Parameter Multiple Times ........ 8-28
8–25 Aliasing from Assigning Cursor Variables to Same Work Area ........ 8-28
8–26 Declaration and Definition of Result-Cached Function ........ 8-30
8–27 Result-Cached Function that Returns Configuration Parameter Setting ........ 8-33
8–28 Function that Depends on Session-Specific Settings ........ 8-35
8–29 Result-Cached Function that Depends on Session-Specific Application Context ........ 8-37
8–30 Caching One Name at a Time (Finer Granularity) ........ 8-38
8–31 Caching Translated Names One Language at a Time (Coarser Granularity) ........ 8-38
9–1 CREATE TRIGGER Statement ........ 9-5
9–2 INSTEAD OF Trigger ........ 9-12
9–3 FOR EACH ROW Trigger ........ 9-13
9–4 Compound Trigger ........ 9-15
9–5 Compound Trigger Records Changes to One Table in Another Table ........ 9-18
9–6 Compound Trigger for Avoiding Mutating-Table Error ........ 9-19

9–7 Trigger for Monitoring Logons .......... 9-21
9–8 Trigger That Invokes Java Subprogram .......... 9-21
9–9 Trigger that Modifies LOB Columns .......... 9-23
9–10 REFERENCING Option .......... 9-24
9–11 Trigger with OBJECT_VALUE Pseudocolumn .......... 9-25
9–12 Remote Exception Handling Failure .......... 9-26
9–13 Workaround for Example 9–12 .......... 9-27
9–14 Row-Level Trigger Causes Mutating-Table Error .......... 9-28
9–15 Statement-Level Trigger Avoids Mutating-Table Error .......... 9-29
9–16 Update Cascade .......... 9-30
9–17 Viewing Information About Triggers .......... 9-33
9–18 Trigger for Auditing .......... 9-36
9–19 Trigger for Auditing .......... 9-37
9–20 Foreign Key Trigger for Child Table .......... 9-40
9–21 UPDATE and DELETE RESTRICT Trigger for Parent Table .......... 9-41
9–22 UPDATE and DELETE SET NULL Triggers for Parent Table .......... 9-42
9–23 DELETE Cascade Trigger for Parent Table .......... 9-42
9–24 UPDATE Cascade Trigger for Parent Table .......... 9-43
9–25 Trigger for Complex Check Constraints .......... 9-44
9–26 Trigger for Enforcing Security .......... 9-45
9–27 Trigger That Derives New Column Values for Table .......... 9-46
10–1 Serially Reusable Package Specification .......... 10-5
10–2 Serially Reusable Package Specification and Package Body .......... 10-6
10–3 Open Cursors in Serially Reusable Packages at Call Boundaries .......... 10-7
10–4 Simple Package Specification Without Body .......... 10-9
10–5 Matching Package Specification and Body .......... 10-10
10–6 Creating emp_admin Package .......... 10-11
10–7 DBMS_OUTPUT.PUT_LINE Procedure .......... 10-17
10–8 Separating Cursor Specifications with Packages .......... 10-19
10–9 Referencing Packaged Cursor .......... 10-19
11–1 Setting Value of PLSQL_WARNINGS Compilation Parameter .......... 11-2
11–2 DBMS_WARNING Package .......... 11-3
11–3 Anonymous Block with Exception Handlers .......... 11-5
11–4 Avoiding the Exception that Example 11–3 Handles .......... 11-5
11–5 Managing Multiple Errors with a Single Exception Handler .......... 11-5
11–6 Scope of Exceptions .......... 11-10
11–7 PRAGMA EXCEPTION_INIT .......... 11-11
11–8 RAISE_APPLICATION_ERROR Procedure .......... 11-11
11–9 Raising a User-Defined Exception .......... 11-13
11–10 Raising a Predefined Exception .......... 11-13
11–11 Scope of an Exception .......... 11-15
11–12 Reraising an Exception .......... 11-16
11–13 Raising an Exception in a Declaration .......... 11-17
11–14 Displaying SQLCODE and SQLERRM .......... 11-19
11–15 Exception Handler .......... 11-19
11–16 Continuing After an Exception .......... 11-20
11–17 Retrying a Transaction After an Exception .......... 11-21
11–18 Identifying Exception Locations with Locator Variables .......... 11-22
12–1 Specifying that a Subprogram Is To Be Inlined .......... 12-2
12–2 Specifying that an Overloaded Subprogram Is To Be Inlined .......... 12-2
12–3 Specifying that a Subprogram Is Not To Be Inlined .......... 12-3
12–4 Applying Two INLINE Pragmas to the Same Subprogram .......... 12-3
12–5 Nesting a Query to Improve Performance .......... 12-5
12–6 Issuing DELETE Statements in a Loop .......... 12-11
12–7 Issuing INSERT Statements in a Loop .......... 12-12

12–8 FORALL Statement for Part of Collection ........ 12-13
12–9 FORALL Statement for Nonconsecutive Index Values ........ 12-13
12–10 Rollbacks with FORALL Statement ........ 12-17
12–11 FORALL Statement and SQL%BULK_EXCEPTIONS ........ 12-18
12–12 FORALL Statement and SQL%BULK_ROWCOUNT ........ 12-19
12–13 Counting Rows Affected by FORALL with SQL%BULK_ROWCOUNT ........ 12-20
12–14 Retrieving Query Results with BULK COLLECT ........ 12-21
12–15 Limiting Query Results with Pseudocolumn ROWNUM ........ 12-22
12–16 Bulk-Fetching from a Cursor Into One or More Collections ........ 12-23
12–17 Bulk-Fetching from a Cursor Into a Collection of Records ........ 12-25
12–18 Controlling Number of BULK COLLECT Rows with LIMIT ........ 12-26
12–19 BULK COLLECT with RETURNING INTO Clause ........ 12-27
12–20 FORALL with BULK COLLECT ........ 12-28
12–21 SELECT BULK COLLECT INTO Statement with Unexpected Results ........ 12-29
12–22 Cursor Workaround for Example 12–21 ........ 12-30
12–23 Second Collection Workaround for Example 12–21 ........ 12-31
12–24 Associating a Cursor with a Dynamic SELECT Statement ........ 12-34
12–25 NOCOPY with Parameters ........ 12-35
12–26 Assigning the Result of a Table Function ........ 12-42
12–27 Pipelined Table Function for Transformation ........ 12-43
12–28 Function with Two Cursor Variable Parameters ........ 12-46
12–29 Pipelined Table Function as Aggregate Function ........ 12-48
A–1 Wrapping Package with DBMS_DDL.CREATE_WRAPPED Procedure ........ A-5
B–1 Resolving Global and Local Variable Names ........ B-1
B–2 Block Label for Name Resolution ........ B-2
B–3 Subprogram Name for Name Resolution ........ B-3
B–4 Dot Notation for Qualifying Names ........ B-4

## List of Figures


1–1 PL/SQL Engine ........ 1-11
5–1 Array and Nested Table ........ 5-5
5–2 Varray of Size 10........ 5-6
6–1 Transaction Control Flow ........ 6-47
8–1 How the PL/SQL Compiler Resolves Calls ........ 8-18
10–1 Package Scope........ 10-9
11–1 Propagation Rules: Example 1 ........ 11-14
11–2 Propagation Rules: Example 2 ........ 11-14
11–3 Propagation Rules: Example 3 ........ 11-15

## List of Tables


1–1 PL/SQL Compilation Parameters ........ 1-12
2–1 Symbols in Database Character Set ........ 2-2
2–2 Comparison of AL16UTF16 and UTF8 Encodings ........ 2-3
2–3 PL/SQL Delimiters ........ 2-4
2–4 Operator Precedence ........ 2-27
2–5 Logical Truth Table ........ 2-29
2–6 Relational Operators ........ 2-34
3–1 Categories of Predefined PL/SQL Data Types ........ 3-1
3–2 Categories of Predefined PL/SQL Scalar Data Types ........ 3-2
3–3 Predefined PL/SQL Numeric Data Types ........ 3-2
3–4 Predefined Subtypes of PLS_INTEGER Data Type ........ 3-3
3–5 Predefined PL/SQL BINARY_FLOAT and BINARY_DOUBLE Constants ........ 3-6
3–6 Predefined Subtypes of NUMBER Data Type ........ 3-7
3–7 Predefined PL/SQL Character Data Types ........ 3-8
3–8 Fields of Datetime and Interval Data Types ........ 3-15
3–9 Predefined PL/SQL Large Object (LOB) Data Types ........ 3-21
3–10 Possible Implicit PL/SQL Data Type Conversions ........ 3-29
5–1 Characteristics of PL/SQL Collection Types ........ 5-2
5–2 Collection Methods ........ 5-22
6–1 Named Cursor Attribute Values ........ 6-17
8–1 Parameter Modes ........ 8-10
8–2 PL/SQL Subprogram Parameter Notations ........ 8-12
8–3 Comparison of Finer and Coarser Caching Granularity ........ 8-37
9–1 Timing-Point Sections of a Compound Trigger Defined ........ 9-16
9–2 Comparison of Built-in Auditing and Trigger-Based Auditing ........ 9-35
9–3 System-Defined Event Attributes ........ 9-52
9–4 Database Events ........ 9-56
9–5 Client Events ........ 9-57
11–1 Compile-Time Warning Categories ........ 11-2
11–2 Predefined Exceptions ........ 11-7
C–1 PL/SQL Compiler Limits ........ C-1
D–1 PL/SQL Reserved Words ........ D-1
D–2 PL/SQL Keywords ........ D-2

# Preface

*Oracle Database PL/SQL Language Reference* describes and explains how to use PL/SQL, the Oracle procedural extension of SQL.

Preface topics:

- Audience
- Documentation Accessibility
- Related Documents
- Conventions
- Syntax Descriptions

## Audience

*Oracle Database PL/SQL Language Reference* is intended for anyone who is developing PL/SQL-based applications for either an Oracle Database or an Oracle TimesTen In-Memory Database, including:

- Programmers
- Systems analysts
- Project managers
- Database administrators

To use this document effectively, you need a working knowledge of:

- Oracle Database
- Structured Query Language (SQL)
- Basic programming concepts such as `IF-THEN` statements, loops, procedures, and functions

## Documentation Accessibility

Our goal is to make Oracle products, services, and supporting documentation accessible to all users, including users that are disabled. To that end, our documentation includes features that make information available to users of assistive technology. This documentation is available in HTML format, and contains markup to facilitate access by the disabled community. Accessibility standards will continue to evolve over time, and Oracle is actively engaged with other market-leading technology vendors to address technical obstacles so that our documentation can be

accessible to all of our customers. For more information, visit the Oracle Accessibility Program Web site at http://www.oracle.com/accessibility/.

### Accessibility of Code Examples in Documentation

Screen readers may not always correctly read the code examples in this document. The conventions for writing code require that closing braces should appear on an otherwise empty line; however, some screen readers may not always read a line of text that consists solely of a bracket or brace.

### Accessibility of Links to External Web Sites in Documentation

This documentation may contain links to Web sites of other companies or organizations that Oracle does not own or control. Oracle neither evaluates nor makes any representations regarding the accessibility of these Web sites.

### Deaf/Hard of Hearing Access to Oracle Support Services

To reach Oracle Support Services, use a telecommunications relay service (TRS) to call Oracle Support at 1.800.223.1711. An Oracle Support Services engineer will handle technical issues and provide customer support according to the Oracle service request process. Information about TRS is available at http://www.fcc.gov/cgb/consumerfacts/trs.html, and a list of phone numbers is available at http://www.fcc.gov/cgb/dro/trsphonebk.html.

## Related Documents

For more information, see these documents in the Oracle Database 11*g* Release 2 (11.2) documentation set:

- *Oracle Database Administrator's Guide*
- *Oracle Database Advanced Application Developer's Guide*
- *Oracle Database SecureFiles and Large Objects Developer's Guide*
- *Oracle Database Object-Relational Developer's Guide*
- *Oracle Database Concepts*
- *Oracle Database PL/SQL Packages and Types Reference*
- *Oracle Database Sample Schemas*
- *Oracle Database SQL Language Reference*

**See Also:** http://www.oracle.com/technology/tech/pl_sql/

## Conventions

This document uses these text conventions:

| Convention | Meaning |
|---|---|
| **boldface** | Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary. |
| *italic* | Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values. |

| Convention | Meaning |
| --- | --- |
| `monospace` | Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter. |
| {A\|B\|C} | Choose either A, B, or C. |

Also:

- `*_view` means all static data dictionary views whose names end with *view*. For example, `*_ERRORS` means `ALL_ERRORS`, `DBA_ERRORS`, and `USER_ERRORS`. For more information about any static data dictionary view, or about static dictionary views in general, see *Oracle Database Reference*.
- Table names not qualified with schema names are in the sample schema `HR`. For information about the sample schemas, see *Oracle Database Sample Schemas*.
- In this document, a **data manipulation language (DML) statement** is a SQL `INSERT`, `UPDATE`, `DELETE`, or `MERGE` statement. (*Oracle Database SQL Language Reference* defines DML differently.)

# Syntax Descriptions

Syntax descriptions are provided in this book for various SQL, PL/SQL, or other command-line constructs in graphic form or Backus Naur Form (BNF). See *Oracle Database SQL Language Reference* for information about how to interpret these descriptions.

# What's New in PL/SQL?

This topic briefly describes the new PL/SQL features that this book documents and provides links to more information.

Topics:

- PL/SQL Features for 11g Release 2 (11.2)
- PL/SQL Features for 11g Release 1 (11.1)

## PL/SQL Features for 11*g* Release 2 (11.2)

The PL/SQL features for 11*g* Release 2 (11.2) are:

- DBMS_PARALLEL_EXECUTE Package
- FORCE Option in CREATE TYPE Statement
- Crossedition Triggers
- ALTER TYPE Statement Restrictions for Editioned ADTs
- RESET option for ALTER TYPE Statement
- Automatic Detection of Data Sources of Result-Cached Function
- Result Caches in Oracle RAC Environment Are No Longer Private

### DBMS_PARALLEL_EXECUTE Package

The `DBMS_PARALLEL_EXECUTE` package enables you to incrementally update the data in a large table in parallel, in two high-level steps:

1. Group sets of rows in the table into smaller chunks.
2. Apply the desired `UPDATE` statement to the chunks in parallel, committing each time you have finished processing a chunk.

This technique is recommended whenever you are updating a lot of data. It improves performance, reduces rollback space consumption, and reduces the number of row locks held.

For more information, see "Updating Large Tables in Parallel" on page 12-50.

### FORCE Option in CREATE TYPE Statement

Before Release 11.2, if a `CREATE OR REPLACE TYPE` statement specified an existing type that had either type dependents or table dependents, the statement failed with error ORA-02303. As of Release 11.2, if you specify `FORCE` in this situation, the

statement fails only if the existing type has table dependents, *not* if it has type dependents.

For more information, see "CREATE TYPE Statement" on page 14-69.

### Crossedition Triggers

Crossedition triggers are intended to fire when database manipulation language (DML) changes are made in a database while an online application that uses the database is in being patched or upgraded with edition-based redefinition. The body of a crossedition trigger is designed to handle these DML changes so that they can be appropriately applied after the changes to the application code are completed.

For more information, see "CREATE TRIGGER Statement" on page 14-53.

> **See Also:** *Oracle Database Advanced Application Developer's Guide* for information about edition-based redefinition in general and crossedition triggers in particular, including the relationship between crossedition triggers and editions

### ALTER TYPE Statement Restrictions for Editioned ADTs

If you use edition-based redefinition to patch or upgrade an application, you use editioned objects. If any of your editioned objects are Abstract Data Types (ADTs), see "Restriction on type" on page 14-21.

> **See Also:** *Oracle Database Advanced Application Developer's Guide* for information about edition-based redefinition in general and editioned objects in particular

### RESET option for ALTER TYPE Statement

The `RESET` option of the `ALTER TYPE` statement resets the version of a type to 1, so that it is no longer considered to be evolved. `RESET` is intended for evolved ADTs that are preventing their owners from being editions-enabled. For more information, see "ALTER TYPE Statement" on page 14-17.

> **See Also:** *Oracle Database Advanced Application Developer's Guide* for information about enabling editions for users

### Automatic Detection of Data Sources of Result-Cached Function

Before Release 11.2, you had to specify any data sources on which a result-cached function depended.

As of Release 11.2, Oracle Database automatically detects all data sources that are queried while a result-cached function is running.

For more information, see "PL/SQL Function Result Cache" on page 8-29.

### Result Caches in Oracle RAC Environment Are No Longer Private

For Release 11.1, each database instance in an Oracle RAC environment had a private function result cache, available only to sessions on that instance. If a required result was missing from the private cache of the local instance, the body of the function ran to compute the result, which was then added to the local cache. The result was not retrieved from the private cache of another instance.

For Release 11.2, each database instance manages its own local result cache, but the local result cache is no longer private—sessions attached to remote database instances can access its contents. If a required result is missing from the result cache of the local

instance, the result might be retrieved from the local cache of another instance, instead of being locally computed.

For more information, see "Result Caches in Oracle RAC Environment" on page 8-38.

## PL/SQL Features for 11*g* Release 1 (11.1)

The PL/SQL features for 11*g*Release 1 (11.1) are:

- Enhancements to Regular Expression Built-In SQL Functions
- SIMPLE_INTEGER, SIMPLE_FLOAT, and SIMPLE_DOUBLE Data Types
- CONTINUE Statement
- Sequences in PL/SQL Expressions
- Dynamic SQL Enhancements
- Named and Mixed Notation in PL/SQL Subprogram Invocations
- PL/SQL Function Result Cache
- Compound Triggers
- More Control Over Triggers
- Database Resident Connection Pool
- Automatic Subprogram Inlining
- PL/Scope
- PL/SQL Hierarchical Profiler
- PL/SQL Native Compiler Generates Native Code Directly

### Enhancements to Regular Expression Built-In SQL Functions

The regular expression built-in SQL functions REGEXP_INSTR and REGEXP_SUBSTR have increased functionality. A new regular expression built-in function, REGEXP_COUNT, returns the number of times a pattern appears in a string. These functions act the same in SQL and PL/SQL.

**See Also:**

- *Oracle Database Advanced Application Developer's Guide* for information about the implementation of regular expressions
- *Oracle Database SQL Language Reference* for detailed descriptions of the REGEXP_INSTR, REGEXP_SUBSTR, and REGEXP_COUNT functions

### SIMPLE_INTEGER, SIMPLE_FLOAT, and SIMPLE_DOUBLE Data Types

The SIMPLE_INTEGER, SIMPLE_FLOAT, and SIMPLE_DOUBLE data types are predefined subtypes of PLS_INTEGER, BINARY_FLOAT, and BINARY_DOUBLE, respectively. Each subtype has the same range as its base type and has a NOT NULL constraint.

SIMPLE_INTEGER differs significantly from PLS_INTEGER in its overflow semantics, but SIMPLE_FLOAT and SIMPLE_DOUBLE are identical to their base types, except for their NOT NULL constraint.

You can use SIMPLE_INTEGER when the value will never be NULL and overflow checking is unnecessary. You can use SIMPLE_FLOAT and SIMPLE_DOUBLE when the

value will never be NULL. Without the overhead of checking for nullness and overflow, these subtypes provide significantly better performance than their base types when PLSQL_CODE_TYPE='NATIVE', because arithmetic operations on SIMPLE_INTEGER values are done directly in the hardware. When PLSQL_CODE_TYPE='INTERPRETED', the performance improvement is smaller.

For more information, see:

- "SIMPLE_INTEGER Subtype of PLS_INTEGER" on page 3-3
- "BINARY_FLOAT and BINARY_DOUBLE Data Types" on page 3-5
- "Recommended Data Types for Integer Arithmetic" on page 12-7
- "Recommended Data Types for Floating-Point Arithmetic" on page 12-7

### CONTINUE Statement

The CONTINUE statement exits the current iteration of a loop and transfers control to the next iteration (in contrast with the EXIT statement, which exits a loop and transfers control to the end of the loop). The CONTINUE statement has two forms: the unconditional CONTINUE and the conditional CONTINUE WHEN.

For more information, see:

- "LOOP Statements" on page 4-9
- "CONTINUE Statement" on page 13-35

### Sequences in PL/SQL Expressions

The pseudocolumns CURRVAL and NEXTVAL make writing PL/SQL source code easier for you and improve run-time performance and scalability. You can use *sequence_name*.CURRVAL and *sequence_name*.NEXTVAL wherever you can use a NUMBER expression.

For more information, see "CURRVAL and NEXTVAL" on page 6-3.

### Dynamic SQL Enhancements

Both native dynamic SQL and the DBMS_SQL package have been enhanced.

Native dynamic SQL now supports a dynamic SQL statement larger than 32 KB by allowing it to be a CLOB—see "EXECUTE IMMEDIATE Statement" on page 13-49 and "OPEN FOR Statement" on page 13-98.

In the DBMS_SQL package:

- All data types that native dynamic SQL supports are supported.
- The DBMS_SQL.PARSE function accepts a CLOB argument, allowing dynamic SQL statements larger than 32 KB.
- The new "DBMS_SQL.TO_REFCURSOR Function" on page 7-7 enables you to switch from the DBMS_SQL package to native dynamic SQL.
- The new "DBMS_SQL.TO_CURSOR_NUMBER Function" on page 7-8 enables you to switch from native dynamic SQL to the DBMS_SQL package.

### Named and Mixed Notation in PL/SQL Subprogram Invocations

Before Release 11.1, a SQL statement that invoked a PL/SQL subprogram had to specify the actual parameters in positional notation. As of Release 11.1, named and mixed notation are also allowed. This improves usability when a SQL statement

invokes a PL/SQL subprogram that has many defaulted parameters, and few of the actual parameters must differ from their default values.

For an example, see the SELECT statements following Example 8–9 on page 8-12.

### PL/SQL Function Result Cache

A function result cache can save significant space and time. Each time a result-cached function is invoked with different parameter values, those parameters and their result are stored in the cache. Subsequently, when the same function is invoked with the same parameter values, the result is retrieved from the cache, instead of being recomputed.

Before Release 11.1, if you wanted your PL/SQL application to cache the results of a function, you had to design and code the cache and cache-management subprograms. If multiple sessions ran your application, each session had to have its own copy of the cache and cache-management subprograms. Sometimes each session had to perform the same expensive computations.

As of Release 11.1, PL/SQL provides a function result cache. To use it, use the RESULT_CACHE clause in each PL/SQL function whose results you want cached. Because the function result cache is stored in a shared global area (SGA), it is available to any session that runs your application.

After you convert your application to PL/SQL function result caching, it uses more SGA, but significantly less total system memory.

For more information, see:

- "PL/SQL Function Result Cache" on page 8-29
- "Function" on page 13-76

### Compound Triggers

A compound trigger is a DML trigger that can fire at multiple timing points.

The body of a compound trigger supports a common PL/SQL state that the code for all of its sections can access. The common state is established when the triggering statement starts and destroyed when the triggering statement completes, even when the triggering statement causes an error.

Before Release 11.1, application developers modeled the common state with an ancillary package. This approach was both cumbersome to program and subject to memory leak when the triggering statement caused an error and the after-statement trigger did not fire. Compound triggers make it easier to program an approach where you want the actions you implement for the various timing points to share common data.

For more information, see "Compound Triggers" on page 9-14.

### More Control Over Triggers

The SQL statement CREATE TRIGGER now supports ENABLE, DISABLE, and FOLLOWS clauses that give you more control over triggers. The DISABLE clause lets you create a trigger in the disabled state, so that you can ensure that your code compiles successfully before you enable the trigger. The ENABLE clause explicitly specifies the default state. The FOLLOWS clause lets you control the firing order of triggers that are defined on the same table and have the same timing point.

For more information, see:

- "Order in Which Triggers Fire" on page 9-20

- "Trigger Enabling" on page 9-32
- "Trigger Disabling" on page 9-32

### Database Resident Connection Pool

DBMS_CONNECTION_POOL package is meant for managing the Database Resident Connection Pool, which is shared by multiple middle-tier processes. The database administrator uses procedures in DBMS_CONNECTION_POOL to start and stop the Database Resident Connection Pool and to configure pool parameters such as size and time limit.

For more information, see "DBMS_CONNECTION_POOL Package" on page 10-18.

### Automatic Subprogram Inlining

Subprogram inlining replaces a subprogram call (to a subprogram in the same PL/SQL unit) with a copy of the called subprogram, which almost always improves program performance.

You can use PRAGMA INLINE to specify that individual subprogram calls are, or are not, to be inlined. You can also turn on automatic inlining—that is, ask the compiler to search for inlining opportunities—by setting the compilation parameter PLSQL_OPTIMIZE_LEVEL to 3 (the default is 2).

In the rare cases when automatic inlining does not improve program performance, you can use the PL/SQL hierarchical profiler to identify subprograms for which you want to turn off inlining.

For more information, see:

- "PL/SQL Optimizer" on page 12-1
- "INLINE Pragma" on page 13-87

> **See Also:** *Oracle Database Reference* for information about the compilation parameter PLSQL_OPTIMIZE_LEVEL

### PL/Scope

PL/Scope is a compiler-driven tool that collects and organizes data about user-defined identifiers from PL/SQL source code. Because PL/Scope is a compiler-driven tool, you use it through interactive development environments (such as SQL Developer and JDeveloper), rather than directly.

PL/Scope enables the development of powerful and effective PL/Scope source code browsers that increase PL/SQL developer productivity by minimizing time spent browsing and understanding source code.

For more information, see "Collecting Data About User-Defined Identifiers" on page 12-8.

> **See Also:** *Oracle Database Advanced Application Developer's Guide*

### PL/SQL Hierarchical Profiler

The PL/SQL hierarchical profiler reports the dynamic execution profile of your PL/SQL program, organized by subprogram calls. It accounts for SQL and PL/SQL execution times separately. Each subprogram-level summary in the dynamic execution profile includes information such as number of calls to the subprogram, time spent in the subprogram itself, time spent in the subprogram's subtree (that is, in its descendent subprograms), and detailed parent-children information.

You can browse the generated HTML reports in any browser. The browser's navigational capabilities, combined with well chosen links, provide a powerful way to analyze performance of large applications, improve application performance, and lower development costs.

For more information, see "Profiling and Tracing PL/SQL Programs" on page 12-9.

**See Also:** *Oracle Database Advanced Application Developer's Guide*

### PL/SQL Native Compiler Generates Native Code Directly

The PL/SQL native compiler now generates native code directly, instead of translating PL/SQL code to C code and having the C compiler generate the native code. An individual developer can now compile PL/SQL units for native execution without any set-up on the part of the DBA. Execution speed of natively compiled PL/SQL programs improves, in some cases by an order of magnitude.

For more information, see "Compiling PL/SQL Units for Native Execution" on page 12-36.

# 1

# Overview of PL/SQL

PL/SQL, the Oracle procedural extension of SQL, is a portable, high-performance transaction-processing language. This chapter explains its advantages and briefly describes its main features and its architecture.

Topics:

- Advantages of PL/SQL
- Main Features of PL/SQL
- Architecture of PL/SQL

## Advantages of PL/SQL

PL/SQL has these advantages:

- Tight Integration with SQL
- High Performance
- High Productivity
- Portability
- Scalability
- Manageability
- Tight Security
- Access to Predefined Packages
- Support for Object-Oriented Programming
- Support for Developing Web Applications
- Support for Developing Server Pages

### Tight Integration with SQL

PL/SQL is tightly integrated with SQL, the most widely used database manipulation language:

- PL/SQL lets you use all SQL data manipulation, cursor control, and transaction control statements, and all SQL functions, operators, and pseudocolumns.
- PL/SQL fully supports SQL data types.

You need not convert between PL/SQL and SQL data types. For example, if your PL/SQL program retrieves a value from a database column of the SQL type `VARCHAR2`, it can store that value in a PL/SQL variable of the type `VARCHAR2`.

You can give a PL/SQL data item the data type of a column or row of a database table without explicitly specifying that data type (see "%TYPE Attribute" on page 1-8 and "%ROWTYPE Attribute" on page 1-8).

- PL/SQL lets you run a SQL query and process the rows of the result set one at a time (see "Processing a Query Result Set One Row at a Time" on page 1-10).

PL/SQL supports both static and dynamic SQL. **Static SQL** is SQL whose full text is known at compilation time. **Dynamic SQL** is SQL whose full text is not known until run time. Dynamic SQL enables you to make your applications more flexible and versatile. For more information, see Chapter 6, "PL/SQL Static SQL" and Chapter 7, "PL/SQL Dynamic SQL".

## High Performance

PL/SQL lets you send a block of statements to the database at once, significantly reducing traffic between the application and the database.

### Bind Arguments

When you embed a SQL `INSERT`, `UPDATE`, `DELETE`, or `SELECT` statement directly in your PL/SQL code, the PL/SQL compiler turns the variables in the `WHERE` and `VALUES` clauses into bind arguments (for details, see "Resolution of Names in Static SQL Statements" on page 6-3). Oracle Database can reuse these SQL statements each time the same code runs, which improves performance.

PL/SQL does not create bind arguments automatically when you use dynamic SQL, but you can use them with dynamic SQL by specifying them explicitly (for details, see "EXECUTE IMMEDIATE Statement" on page 7-2).

### Subprograms

PL/SQL lets you write subprograms. Each subprogram is compiled once and stored in executable form, which can be invoked repeatedly. Because stored subprograms run in the database server, a single call over the network can start a large job. This division of work reduces network traffic and improves response times. Stored subprograms are cached and shared among users, which lowers memory requirements and call overhead. For more information about subprograms, see "Subprograms" on page 1-9.

### Optimizer

The PL/SQL compiler has an optimizer that can rearrange code for better performance. For more information about the optimizer, see "PL/SQL Optimizer" on page 12-1.

## High Productivity

PL/SQL lets you write compact code for manipulating data. Just as a scripting language like PERL can read, transform, and write data in files, PL/SQL can query, transform, and update data in a database.

PL/SQL has many features that save designing and debugging time, and it is the same in all environments. If you learn to use PL/SQL with one Oracle tool, you can transfer your knowledge to other Oracle tools. For example, you can create a PL/SQL block in SQL Developer and then use it in an Oracle Forms trigger. For an overview of PL/SQL features, see "Main Features of PL/SQL" on page 1-4.

## Portability

You can run PL/SQL applications on any operating system and platform where Oracle Database runs.

## Scalability

PL/SQL stored subprograms increase scalability by centralizing application processing on the database server. The shared memory facilities of the shared server enable Oracle Database to support thousands of concurrent users on a single node. For more information about subprograms, see "Subprograms" on page 1-9.

For further scalability, you can use Oracle Connection Manager to multiplex network connections. For information about Oracle Connection Manager, see *Oracle Database Net Services Reference*.

## Manageability

PL/SQL stored subprograms increase manageability because you can maintain only one copy of a subprogram, on the database server, rather than one copy on each client system. Any number of applications can use the subprograms, and you can change the subprograms without affecting the applications that invoke them. For more information about subprograms, see "Subprograms" on page 1-9.

## Tight Security

PL/SQL stored subprograms let you divide application logic between the database server and the clients. You can restrict client access to server data by allowing users to manipulate the data only through stored subprograms that have restricted privileges. For example, you can grant users access to a subprogram that updates a table, but not grant them access to the table itself. For information about granting access to users, see *Oracle Database SQL Language Reference*.

PL/SQL database triggers can prevent applications from making specified updates and can audit user queries. For more information about triggers, see "Triggers" on page 1-9.

You can wrap your PL/SQL source code to protect it from business competitors and others who might misuse it. For more information about wrapping, see Appendix A, "PL/SQL Source Code Wrapping."

> **See Also:** *Oracle Database Security Guide* for more information about database security features

## Access to Predefined Packages

A **package** is a PL/SQL unit that consists of related subprograms and the explicit cursors and variables that they use. Packages are stored in the database, where they can be shared by many applications.

Oracle provides product-specific packages that define application program interfaces (APIs) you can invoke from your PL/SQL programs to perform many useful tasks. For information about some widely used predefined packages, see "Overview of Product-Specific Packages" on page 10-16.

For more information about packages in general, see "Packages (APIs Written in PL/SQL)" on page 1-9.

## Support for Object-Oriented Programming

PL/SQL supports object-oriented programming with "Abstract Data Types" on page 1-8.

## Support for Developing Web Applications

PL/SQL lets you create applications that generate web pages directly from the database, allowing you to make your database available on the Web and make back-office data accessible on the intranet.

The program flow of a PL/SQL Web application is similar to that in a CGI PERL script. Developers often use CGI scripts to produce web pages dynamically, but such scripts are often not optimal for accessing the database. Delivering Web content with PL/SQL stored subprograms provides the power and flexibility of database processing. For example, you can use DML, dynamic SQL, and cursors. You also eliminate the process overhead of forking a new CGI process to handle each HTTP request.

You can implement a Web browser-based application entirely in PL/SQL with PL/SQL Gateway and the PL/SQL Web Toolkit.

PL/SQL Gateway enables a Web browser to invoke a PL/SQL stored subprogram through an HTTP listener. `mod_plsql`, one implementation of the PL/SQL Gateway, is a plug-in of Oracle HTTP Server and enables Web browsers to invoke PL/SQL stored subprograms.

PL/SQL Web Toolkit is a set of PL/SQL packages that provides a generic interface to use stored subprograms invoked by `mod_plsql` at run time.

> **See Also:** *Oracle Database Advanced Application Developer's Guide* for information about developing PL/SQL Web applications

## Support for Developing Server Pages

PL/SQL Server Pages (PSPs) let you develop web pages with dynamic content. PSPs are an alternative to coding a stored subprogram that writes the HTML code for a web page one line at a time.

Special tags let you embed PL/SQL scripts into HTML source code. The scripts run when Web clients, such as browsers, request the pages. A script can accept parameters, query or update the database, and then display a customized page showing the results.

During development, PSPs can act like templates, with a static part for page layout and a dynamic part for content. You can design the layouts using your favorite HTML authoring tools, leaving placeholders for the dynamic content. Then, you can write the PL/SQL scripts that generate the content. When finished, you simply load the resulting PSP files into the database as stored subprograms.

> **See Also:** *Oracle Database Advanced Application Developer's Guide* for information about developing PSPs

# Main Features of PL/SQL

PL/SQL combines the data-manipulating power of SQL with the processing power of procedural languages.

When you can solve a problem with SQL, you can issue SQL statements from your PL/SQL program, without learning new APIs.

Like other procedural programming languages, PL/SQL lets you declare constants and variables, control program flow, define subprograms, and trap run-time errors.

You can break complex problems into easily understandable subprograms, which you can reuse in multiple applications.

Topics:

- Blocks
- Error Handling
- Input and Output
- Variables and Constants
- Data Abstraction
- Control Statements
- Subprograms
- Triggers
- Packages (APIs Written in PL/SQL)
- Conditional Compilation
- Processing a Query Result Set One Row at a Time

## Blocks

The basic unit of a PL/SQL source program is the **block**, which groups related declarations and statements.

A PL/SQL block is defined by the keywords `DECLARE`, `BEGIN`, `EXCEPTION`, and `END`. These keywords partition the block into a declarative part, an executable part, and an exception-handling part. Only the executable part is required. A block can have a label.

Example 1–1 shows the basic structure of a PL/SQL block. For syntax details, see "Block" on page 13-8.

***Example 1–1 PL/SQL Block Structure***

```
<< label >> (optional)
DECLARE    -- Declarative part (optional)
  -- Declarations of local types, variables, & subprograms

BEGIN      -- Executable part (required)
  -- Statements (which can use items declared in declarative part)

[EXCEPTION -- Exception-handling part (optional)
  -- Exception handlers for exceptions raised in executable part]
END;
```

Declarations are local to the block and cease to exist when the block completes execution, helping to avoid cluttered namespaces for variables and subprograms.

Blocks can be nested: Because a block is an executable statement, it can appear in another block wherever an executable statement is allowed.

You can submit a block to an interactive tool (such as SQL*Plus or Enterprise Manager) or embed it in an Oracle Precompiler or OCI program. The interactive tool or program

runs the block only once. The block is not stored in the database, and for that reason, it is called an **anonymous block** (even if it has a label).

---

**Note:** An anonymous block is a SQL statement.

---

## Error Handling

PL/SQL makes it easy to detect and process error conditions, which are called **exceptions**. When an error occurs, an exception is raised: normal execution stops and control transfers to special exception-handling code, which comes at the end of any PL/SQL block. Each different exception is processed by a particular exception handler.

PL/SQL exception handling differs from the manual checking that you do in C programming, where you insert a check to ensure that every operation succeeded. Instead, the checks and calls to exception handlers are performed automatically, similar to the exception mechanism in Java programming.

Predefined exceptions are raised automatically for certain common error conditions involving variables or database operations. For example, if you try to divide a number by zero, PL/SQL raises the predefined exception `ZERO_DIVIDE` automatically.

You can define exceptions of your own, for conditions that you decide are errors, or to correspond to database errors that normally result in ORA-*n* error messages. When you detect a user-defined error condition, you raise an exception with either a `RAISE` statement or the procedure `DBMS_STANDARD.RAISE_APPLICATION_ERROR`. In the example, if the commission is null, the exception `comm_missing` is raised.

Typically, you put an exception handler at the end of a subprogram to handle exceptions that are raised anywhere inside the subprogram. To continue running from the spot where an exception happens, enclose the code that might raise an exception inside another `BEGIN` `END` block with its own exception handler. For example, you might put separate `BEGIN` `END` blocks around groups of SQL statements that might raise `NO_DATA_FOUND`, or around arithmetic operations that might raise `DIVIDE_BY_ZERO`. By putting a `BEGIN` `END` block with an exception handler inside a loop, you can continue running the loop even if some loop iterations raise exceptions. See Example 5–38 on page 5-31.

For more information, see Chapter 11, "PL/SQL Error Handling".

## Input and Output

Most PL/SQL input and output (I/O) is through SQL statements that store data in database tables or query those tables. All other PL/SQL I/O is done through APIs, such as the PL/SQL package `DBMS_OUTPUT`. For more information, see "DBMS_OUTPUT Package" on page 10-17.

To display output passed to `DBMS_OUTPUT`, you need another program, such as SQL*Plus. To see `DBMS_OUTPUT` output with SQL*Plus, you must first issue the SQL*Plus command `SET SERVEROUTPUT ON`. For more information, see *SQL*Plus User's Guide and Reference*.

Other PL/SQL APIs for processing I/O are provided by these packages:

| Package(s) | Purpose | More Information |
|---|---|---|
| `HTF` and `HTP` | Displays output on a web page | "HTF and HTP Packages" on page 10-18 |

| Package(s) | Purpose | More Information |
|---|---|---|
| DBMS_PIPE | Passes information between PL/SQL and operating-system commands | "DBMS_PIPE Package" on page 10-17 |
| UTL_FILE | Reads and writes operating system files | "UTL_FILE Package" on page 10-18 |
| UTL_HTTP | Communicates with web servers | "UTL_HTTP Package" on page 10-18 |
| UTL_SMTP | Communicates with mail servers | "UTL_SMTP Package" on page 10-18 |

Some of the preceding APIs can both accept input and display output, but they cannot accept data directly from the keyboard. To accept data directly from the keyboard, use the SQL*Plus commands PROMPT and ACCEPT.

**See Also:**

- *SQL*Plus User's Guide and Reference* for information about the SQL*Plus command PROMPT
- *SQL*Plus User's Guide and Reference* for information about the SQL*Plus command ACCEPT

## Variables and Constants

PL/SQL lets you declare variables and constants, and then use them wherever you can use an expression. As the program runs, the values of variables can change, but the values of constants cannot. For more information, see "Declarations" on page 2-12 and "Assigning Values to Variables" on page 2-22.

## Data Abstraction

Data abstraction lets you work with the essential properties of data without being too involved with details. You can design a data structure first, and then design algorithms that manipulate it.

Topics:

- Cursors
- Composite Variables
- %ROWTYPE Attribute
- %TYPE Attribute
- Abstract Data Types

### Cursors

A **cursor** is a pointer to a private SQL area that stores information about processing a specific SQL statement or PL/SQL SELECT INTO statement. You can use the cursor to retrieve the rows of the result set one at a time. You can use cursor attributes to get information about the state of the cursor—for example, how many rows the statement has affected so far. For more information about cursors, see "Cursors" on page 6-6.

### Composite Variables

A **composite variable** has internal components. You can treat the internal components as individual variables. You can also pass entire composite variables to subprograms as parameters. PL/SQL has two kinds of composite variables, collections and records.

In a **collection**, the internal components are always of the same data type, and are called **elements**. You access each element by its unique subscript. Lists and arrays are classic examples of collections.

In a **record**, the internal components can be of different data types, and are called **fields**. You access each field by its name. A record variable can hold a table row, or some columns from a table row.

For more information about composite variables, see Chapter 5, "PL/SQL Collections and Records."

### %ROWTYPE Attribute

The `%ROWTYPE` attribute lets you declare a record that represents a row of a database table or view: For every column of the row, the record has a field with the same name and data type. The record can store an entire row that is either selected from the table or view or fetched from a cursor. If the structure of the row changes, the structure of the record changes accordingly. For more information about `%ROWTYPE`, see "%ROWTYPE Attribute" on page 2-16.

### %TYPE Attribute

The `%TYPE` attribute lets you declare a data item of the same data type as a previously declared data item or database column. If the declaration of the referenced item changes, the declaration of the referencing item changes accordingly. The `%TYPE` attribute is particularly useful when declaring variables to hold database values. For more information about `%TYPE`, see "%TYPE Attribute" on page 2-15.

### Abstract Data Types

An **Abstract Data Type (ADT)** consists of a data structure and subprograms that manipulate the data. The variables that form the data structure are called **attributes**. The subprograms that manipulate the attributes are called **methods**.

ADTs are stored in the database. Instances of ADTs can be stored in tables and used as PL/SQL variables.

ADTs let you reduce complexity by separating a large system into logical components, which you can reuse.

In the static data dictionary view `*_OBJECTS`, the `OBJECT_TYPE` of an ADT is `TYPE`. In the static data dictionary view `*_TYPES`, the `TYPECODE` of an ADT is `OBJECT`.

For more information about ADTs, see "CREATE TYPE Statement" on page 14-69.

**Note:** ADTs are also called **user-defined types** and **object types**.

**See Also:** *Oracle Database Object-Relational Developer's Guide* for information about ADTs

## Control Statements

Control statements are the most important PL/SQL extension to SQL.

PL/SQL has three categories of control statements:

- **Conditional selection statements**, which let you run different statements for different data values.

  For more information, see "Conditional Selection Statements" on page 4-1.

- **Loop statements**, which let you repeat the same statements with a series of different data values.

  For more information, see "LOOP Statements" on page 4-9.

- **Sequential control statements**, which allow you to go to a specified, labeled statement, or to do nothing.

  For more information, see "Sequential Control Statements" on page 4-20.

## Subprograms

A PL/SQL **subprogram** is a named PL/SQL block that can be invoked (with a set of parameters, if it has them). PL/SQL has two types of subprograms, procedures and functions. A function returns a result.

You can create a PL/SQL subprogram either at schema level, inside a package, or inside a PL/SQL block (which can be another subprogram).

For more information about PL/SQL subprograms, see Chapter 8, "PL/SQL Subprograms."

PL/SQL also lets you invoke external programs written in other languages. For more information, see "External Subprograms" on page 8-25.

## Triggers

A **trigger** is a named PL/SQL unit that is stored in the database and run in response to an event that occurs in the database. You can specify the event, whether the trigger fires before or after the event, and whether the trigger runs once for each event or once for each row affected by the event. For example, you can create a trigger that runs every time an `INSERT` statement affects the `EMPLOYEES` table.

For more information about triggers, see Chapter 9, "PL/SQL Triggers."

## Packages (APIs Written in PL/SQL)

A **package** is a PL/SQL unit that consists of related subprograms and the explicit cursors and variables that they use. Packages are stored in the database, where they can be shared by many applications.

The **package specification** defines the package, declaring the types, variables, constants, exceptions, explicit cursors, and subprograms that can be referenced from outside the package. A package specification is an API: It has all the information that client programs need to call its subprograms, but no information about their implementation.

The **package body** defines the queries for the explicit cursors, and the code for the subprograms, that are declared in the package specification (therefore, a package with neither explicit cursors nor subprograms does not need a body). The package body can also define local subprograms, which are not declared in the specification and can be invoked only by other subprograms in the package. Package body contents are hidden from client programs. You can change the package body without invalidating the applications that call the package.

Invoking a packaged subprogram for the first time loads the whole package and caches it in memory. Therefore, packages improve performance in a multiuser, multi-application environment.

For more information about packages, see Chapter 10, "PL/SQL Packages."

## Conditional Compilation

Conditional compilation lets you customize the functionality in a PL/SQL application without removing source code. For example, you can:

- Use new features with the latest database release, and disable them when running the application in an older database release.
- Activate debugging or tracing statements in the development environment, and hide them when running the application at a production site.

For more information, see "Conditional Compilation" on page 2-43.

## Processing a Query Result Set One Row at a Time

PL/SQL lets you issue a SQL query and process the rows of the result set one at a time. You can use a simple loop, as in Example 1–2, or you can control the process precisely by using individual statements to perform the query, retrieve data, and finish processing.

***Example 1–2 Processing Query Result Rows One at a Time***

```
BEGIN
  FOR someone IN (SELECT * FROM employees WHERE employee_id < 120)
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      'First name = ' || someone.first_name ||
      ', Last name = ' || someone.last_name
    );
  END LOOP;
END;
/
```

Result:

```
First name = Steven, Last name = King
First name = Neena, Last name = Kochhar
First name = Lex, Last name = De Haan
First name = Alexander, Last name = Hunold
First name = Bruce, Last name = Ernst
First name = David, Last name = Austin
First name = Valli, Last name = Pataballa
First name = Diana, Last name = Lorentz
First name = Nancy, Last name = Greenberg
First name = Daniel, Last name = Faviet
First name = John, Last name = Chen
First name = Ismael, Last name = Sciarra
First name = Jose Manuel, Last name = Urman
First name = Luis, Last name = Popp
First name = Den, Last name = Raphaely
First name = Alexander, Last name = Khoo
First name = Shelli, Last name = Baida
First name = Sigal, Last name = Tobias
First name = Guy, Last name = Himuro
First name = Karen, Last name = Colmenares
```

# Architecture of PL/SQL

Topics:

- PL/SQL Engine

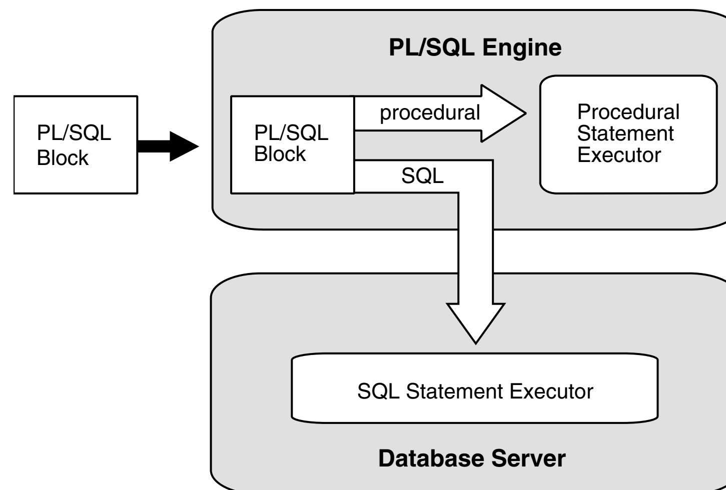
- PL/SQL Units and Compilation Parameters

## PL/SQL Engine

The PL/SQL compilation and run-time system is an engine that compiles and runs PL/SQL units. The engine can be installed in the database or in an application development tool, such as Oracle Forms.

In either environment, the PL/SQL engine accepts as input any valid PL/SQL unit. The engine runs procedural statements, but sends SQL statements to the SQL engine in the database, as shown in Figure 1–1.

***Figure 1–1 PL/SQL Engine***




Typically, the database processes PL/SQL units.

When an application development tool processes PL/SQL units, it passes them to its local PL/SQL engine. If a PL/SQL unit contains no SQL statements, the local engine processes the entire PL/SQL unit. This is useful if the application development tool can benefit from conditional and iterative control.

For example, Oracle Forms applications frequently use SQL statements to test the values of field entries and do simple computations. By using PL/SQL instead of SQL, these applications can avoid calls to the database.

## PL/SQL Units and Compilation Parameters

A PL/SQL unit is any one of these:

- PL/SQL anonymous block
- `FUNCTION`
- `LIBRARY`
- `PACKAGE`
- `PACKAGE BODY`
- `PROCEDURE`
- `TRIGGER`
- `TYPE`

- TYPE BODY

PL/SQL units are affected by PL/SQL compilation parameters (a category of database initialization parameters). Different PL/SQL units—for example, a package specification and its body—can have different compilation parameter settings.

Table 1–1 summarizes the PL/SQL compilation parameters. To display the values of these parameters for specified or all PL/SQL units, query the static data dictionary view ALL_PLSQL_OBJECT_SETTINGS. For information about this view, see *Oracle Database Reference*.

***Table 1–1 PL/SQL Compilation Parameters***

| Parameter | Description |
|---|---|
| PLSCOPE_SETTINGS | Controls the compile-time collection, cross reference, and storage of PL/SQL source code identifier data. Used by the PL/Scope tool (see *Oracle Database Advanced Application Developer's Guide*).<br>For more information about PLSCOPE_SETTINGS, see *Oracle Database Reference*. |
| PLSQL_CCFLAGS | Enables you to control conditional compilation of each PL/SQL unit independently.<br>For more information about PLSQL_CCFLAGS, see "How Conditional Compilation Works" on page 2-44 and *Oracle Database Reference*. |
| PLSQL_CODE_TYPE | Specifies the compilation mode for PL/SQL units—INTERPRETED (the default) or NATIVE. For information about which mode to use, see "Determining Whether to Use PL/SQL Native Compilation" on page 12-37.<br>If the optimization level (set by PLSQL_OPTIMIZE_LEVEL) is less than 2:<br>▪ The compiler generates interpreted code, regardless of PLSQL_CODE_TYPE.<br>▪ If you specify NATIVE, the compiler warns you that NATIVE was ignored.<br>For more information about PLSQL_CODE_TYPE, see *Oracle Database Reference*. |
| PLSQL_OPTIMIZE_LEVEL | Specifies the optimization level at which to compile PL/SQL units (the higher the level, the more optimizations the compiler tries to make).<br>PLSQL_OPTIMIZE_LEVEL=1 instructs the PL/SQL compiler to generate and store code for use by the PL/SQL debugger.<br>For more information about PLSQL_OPTIMIZE_LEVEL, see "PL/SQL Optimizer" on page 12-1 and *Oracle Database Reference*. |
| PLSQL_WARNINGS | Enables or disables the reporting of warning messages by the PL/SQL compiler, and specifies which warning messages to show as errors.<br>For more information about PLSQL_WARNINGS, see "Compile-Time Warnings" on page 11-1 and *Oracle Database Reference*. |
| NLS_LENGTH_SEMANTICS | Enables you to create CHAR and VARCHAR2 columns using either byte-length or character-length semantics.<br>For more information about byte and character length semantics, see "CHAR and VARCHAR2 Data Types" on page 3-8.<br>For more information about NLS_LENGTH_SEMANTICS, see *Oracle Database Reference*. |

---

**Note:** The compiler parameter `PLSQL_DEBUG`, which specifies whether to compile PL/SQL units for debugging, is deprecated. To compile PL/SQL units for debugging, specify `PLSQL_OPTIMIZE_LEVEL=1`.

---

The compile-time values of the parameters in Table 1–1 are stored with the metadata of each stored PL/SQL unit, which means that you can reuse those values when you explicitly recompile the unit. (A **stored PL/SQL unit** is created with one of the "CREATE [OR REPLACE] Statements" on page 14-1. An anonymous block is not a stored PL/SQL unit.)

To explicitly recompile a stored PL/SQL unit and reuse its parameter values, you must use an `ALTER` statement with both the `COMPILE` clause and the `REUSE SETTINGS` clause. For more information about `REUSE SETTINGS`, see "compiler_parameters_clause" on page 14-4. (All `ALTER` statements have this clause. For a list of `ALTER` statements, see "ALTER Statements" on page 14-1.)

# 2

# PL/SQL Language Fundamentals

This chapter explains these aspects of the PL/SQL language:

- Character Sets
- Lexical Units
- Declarations
- References to Identifiers
- Scope and Visibility of Identifiers
- Assigning Values to Variables
- Expressions
- Error-Reporting Functions
- SQL Functions in PL/SQL Expressions
- Pragmas
- Conditional Compilation

## Character Sets

PL/SQL supports two character sets:

- Database Character Set
- National Character Set

### Database Character Set

PL/SQL uses the **database character set** for identifiers (which are described in "Identifiers" on page 2-5) and to store data.

The database character set consists of these characters:

- **Letters** *A* through *Z* and *a* through *z*

  Corresponding uppercase and lowercase letters are equivalent—for example, *A* is equivalent to *a*—except in character literals and string literals.

- **Digits** *0* through *9*
- **Symbols** in Table 2–1
- **Whitespace characters** tab, space, and carriage return

*Table 2–1 Symbols in Database Character Set*

| Symbol | Name |
|---|---|
| ( | Left parenthesis |
| ) | Right parenthesis |
| [ | Left bracket |
| ] | Right bracket |
| < | Left angle bracket |
| > | Right angle bracket |
| { | Left brace |
| } | Right brace |
| + | Plus sign |
| - | Hyphen *or* minus sign |
| * | Asterisk |
| / | Slash |
| = | Equal sign |
| , | Comma |
| ; | Semicolon |
| : | Colon |
| . | Period |
| ! | Exclamation point |
| ? | Question mark |
| ' | Apostrophe *or* single quotation mark |
| " | Quotation mark *or* double quotation mark |
| ~ | Tilde |
| ^ | Caret |
| @ | At sign |
| % | Percent sign |
| # | Number sign |
| $ | Dollar sign |
| & | Ampersand |
| _ | Underscore |
| \| | Vertical bar |

To specify the database character set, use the `CHARACTER SET` clause of the `CREATE DATABASE` statement. For information about this statement, see *Oracle Database SQL Language Reference*.

**See Also:** "CHAR and VARCHAR2 Data Types" on page 3-8

## National Character Set

PL/SQL uses the **national character set** to represent languages that have thousands of characters, each of which requires two or three bytes (Japanese, for example). The national character set represents data as Unicode, using either the `AL16UTF16` or `UTF8` encoding.

Table 2–2 compares `AL16UTF16` and `UTF8` encodings. For maximum reliability, Oracle recommends using the default `AL16UTF16` encoding wherever practical.

***Table 2–2 Comparison of AL16UTF16 and UTF8 Encodings***

| Encoding | Character Size (Bytes) | Advantage | Disadvantage |
|---|---|---|---|
| `AL16UTF16` (default) | 2 | Easy to calculate string lengths, which you must do to avoid truncation errors when mixing programming languages. | Strings composed mostly of ASCII or EBCDIC characters take more space than necessary. |
| `UTF8` | 1, 2, or 3 | If most characters use only one byte, you can fit more characters into a variable or table column. | Possibility of truncation errors when transferring the data to a buffer measured in bytes. |

To determine how many bytes a Unicode string needs, use the SQL function `LENGTHB`. For information about this statement, see *Oracle Database SQL Language Reference*.

To specify the national character set, use the `NATIONAL CHARACTER SET` clause of the `CREATE DATABASE` statement. For information about this statement, see *Oracle Database SQL Language Reference*.

**See Also:**

- *Oracle Database Globalization Support Guide* for more information about the national character set
- "NCHAR and NVARCHAR2 Data Types" on page 3-12

# Lexical Units

The **lexical units** of PL/SQL are its smallest individual components. They fall into these catagories:

- Delimiters
- Identifiers
- Literals
- Comments

You cannot embed whitespace or punctuation characters inside lexical units (except string literals and comments).

Example 2–1 shows what happens when you embed a space in a delimiter (the assignment operator).

***Example 2–1 Delimiter with Embedded Space***

```
BEGIN
  count := count + 1;   -- correct
  count : = count + 1;  -- incorrect
```

```
END;
/
```

Result:

```
  count : = count + 1;  -- incorrect
        *
ERROR at line 3:
ORA-06550: line 3, column 9:
PLS-00103: Encountered the symbol ":" when expecting one of these:
:= . ( @ % ;
```

You can put whitespace characters between lexical units, which often makes your source code easier to read, as in Example 2–2.

***Example 2–2 Whitespace Characters Improving Source Code Readability***

```
DECLARE
  x    NUMBER := 10;
  y    NUMBER := 5;
  max  NUMBER;
BEGIN
  IF x>y THEN max:=x;ELSE max:=y;END IF;  -- correct but hard to read

  -- Easier to read:

  IF x > y THEN
    max:=x;
  ELSE
    max:=y;
  END IF;
END;
/
```

Topics:

- Delimiters
- Identifiers
- Literals
- Comments

## Delimiters

A **delimiter** is a character, or character combination, that has a special meaning in PL/SQL. Table 2–3 summarizes the PL/SQL delimiters.

***Table 2–3 PL/SQL Delimiters***

| Delimiter | Meaning |
|---|---|
| + | Addition operator |
| := | Assignment operator |
| => | Association operator |
| % | Attribute indicator |
| ' | Character string delimiter |
| . | Component indicator |

***Table 2–3 (Cont.) PL/SQL Delimiters***

<table>
<tr><th>Delimiter</th><th>Meaning</th></tr>
<tr><td>||</td><td>Concatenation operator</td></tr>
<tr><td>/</td><td>Division operator</td></tr>
<tr><td>**</td><td>Exponentiation operator</td></tr>
<tr><td>(</td><td>Expression or list delimiter (begin)</td></tr>
<tr><td>)</td><td>Expression or list delimiter (end)</td></tr>
<tr><td>:</td><td>Host variable indicator</td></tr>
<tr><td>,</td><td>Item separator</td></tr>
<tr><td><<</td><td>Label delimiter (begin)</td></tr>
<tr><td>>></td><td>Label delimiter (end)</td></tr>
<tr><td>/*</td><td>Multiline comment delimiter (begin)</td></tr>
<tr><td>*/</td><td>Multiline comment delimiter (end)</td></tr>
<tr><td>*</td><td>Multiplication operator</td></tr>
<tr><td>"</td><td>Quoted identifier delimiter</td></tr>
<tr><td>..</td><td>Range operator</td></tr>
<tr><td>=</td><td>Relational operator (equal)</td></tr>
<tr><td><></td><td>Relational operator (not equal)</td></tr>
<tr><td>!=</td><td>Relational operator (not equal)</td></tr>
<tr><td>~=</td><td>Relational operator (not equal)</td></tr>
<tr><td>^=</td><td>Relational operator (not equal)</td></tr>
<tr><td><</td><td>Relational operator (less than)</td></tr>
<tr><td>></td><td>Relational operator (greater than)</td></tr>
<tr><td><=</td><td>Relational operator (less than or equal)</td></tr>
<tr><td>>=</td><td>Relational operator (greater than or equal)</td></tr>
<tr><td>@</td><td>Remote access indicator</td></tr>
<tr><td>--</td><td>Single-line comment indicator</td></tr>
<tr><td>;</td><td>Statement terminator</td></tr>
<tr><td>-</td><td>Subtraction or negation operator</td></tr>
</table>

## Identifiers

**Identifiers** name PL/SQL objects, which include:

- Constants
- Cursors
- Exceptions
- Keywords
- Labels
- Packages
- Reserved words

- Subprograms
- Variables
- Types

You must separate adjacent identifiers by one or more spaces or a punctuation character.

Example 2–3 shows what happens when the identifiers END and IF are not separated.

***Example 2–3 Identifiers Not Separated by Space or Punctuation***

```
BEGIN
  IF x > y THEN high := x; END IF;  -- correct
  IF x > y THEN high := x; ENDIF;   -- incorrect
END;
/
```

Result:

```
END;
   *
ERROR at line 4:
ORA-06550: line 4, column 4:
PLS-00103: Encountered the symbol ";" when expecting one of these:
if
```

PL/SQL is case-insensitive for identifiers. For example, these identifiers are the same:

```
lastname
LastName
LASTNAME
```

Every character in an identifier, alphabetic or not, is significant. For example, these identifiers are different:

```
lastname
last_name
```

Topics:


- Reserved Words and Keywords
- Predefined Identifiers
- User-Defined Identifiers
- Quoted Identifiers


### Reserved Words and Keywords

**Reserved words** and **keywords** are identifiers that have special meaning in PL/SQL.

You cannot use reserved words as ordinary user-defined identifiers. You can use keywords as ordinary user-defined identifiers, and you can use reserved words as quoted user-defined identifiers, but neither practice is recommended.

For lists of PL/SQL reserved words and keywords, see Table D–1 on page D-1 and Table D–2 on page D-2, respectively.

### Predefined Identifiers

**Predefined identifiers** are declared in the predefined package STANDARD. An example of a predefined identifier is the exception INVALID_NUMBER. You can use predefined

identifiers as user-defined identifiers, but it is not recommended. Your local declaration overrides the global declaration (see "Scope and Visibility of Identifiers" on page 2-18).

For a list of predefined identifiers, query the static data dictionary view `*_TYPES`. For example:

```
SELECT TYPE_NAME, PREDEFINED FROM ALL_TYPES;
```

### User-Defined Identifiers

PL/SQL lets you define identifiers. An ordinary (as opposed to quoted) user-defined identifier:

- Begins with a letter
- Can include digits and these symbols:
  - Dollar sign ($)
  - Number sign (#)
  - Underscore (_)
- Has at most 30 characters

Examples of acceptable ordinary user-defined identifiers:

```
X
t2
phone#
credit_limit
LastName
oracle$number
money$$$tree
SN##
try_again_
```

Examples of unacceptable ordinary user-defined identifiers:

```
mine&yours
debit-amount
on/off
user id
```

> **Tip:** Make user-defined identifiers meaningful. For example, the meaning of `cost_per_thousand` is obvious, but the meaning of `cpt` is not.

### Quoted Identifiers

A **quoted identifier** is a user-defined identifier that is enclosed in double quotation marks. Between the double quotation marks, any characters are allowed except double quotation marks. For example, these identifiers are acceptable:

```
"X+Y"
"last name"
"on/off switch"
"employee(s)"
"*** header info ***"
```

The maximum size of a quoted identifier is 30 characters (excluding the double quotation marks).

## Literals

A **literal** is a value that is not represented by an identifier or an expression.

Topics:

- Numeric Literals
- Character Literals
- String Literals
- BOOLEAN Literals
- Datetime and Interval Literals

For syntax diagrams of literals, see "Literal" on page 13-91.

### Numeric Literals

A **numeric literal** can have a sign (+ or -) and can be either an integer or a real number.

An integer literal is a whole number without a decimal point. For example:

```
030
6
-14
+32767
```

A real literal is a whole or fractional number with a decimal point. For example:

```
6.6667
0.0
-12.0
3.14159
+8300.00
.5
25.
```

In PL/SQL, numbers such as `12.0` and `25.` are real.

If you add the suffix `d` or `D` to a numeric literal, its data type is `BINARY_DOUBLE`; if you add the suffix `f` or `F`, its data type is `BINARY_FLOAT`. For example:

```
2.0f
2D
```

You can write numeric literals in scientific notation, using `e` or `E`. For example:

```
2E5
1.0E-7
3.14159e0
-1E38
-9.5e-3
```

`aEb` means $a(10^b)$—or, in PL/SQL, `a*(10**b)`. Therefore:

```
5E3 = 5 * 10**3 = 5 * 1000 = 5000
```

For information about numeric data types, see "Predefined PL/SQL Numeric Data Types and Subtypes" on page 3-2.

### Character Literals

A **character literal** is a single character enclosed in single quotation marks. For example:

```
'Z'
'%'
'7'
' '
'z'
'('
```

A character literal has the data type CHAR. For information about the CHAR data type, see "CHAR and VARCHAR2 Data Types" on page 3-8.

PL/SQL is case-sensitive for character literals. For example, the literals 'Z' and 'z' are different.

The character literals '0' through '9' are not equivalent to the integer literals 0 through 9, but you can use them in arithmetic expressions, because PL/SQL converts them to integers.

### String Literals

A **string literal** is a string of zero or more characters, enclosed in single quotation marks. For example:

```
'Hello, world!'
'XYZ Corporation'
'10-NOV-91'
'He said, "Life is like licking honey from a thorn."'
'$1,000,000'
```

A string literal has the data type CHAR. For information about the CHAR data type, see "CHAR and VARCHAR2 Data Types" on page 3-8.

A string literal with zero characters is a **null string**. It has the value NULL.

PL/SQL is case-sensitive for string literals. For example, these literals are different:

```
'baker'
'Baker'
```

Blanks are significant in string literals, including leading and trailing blanks. For example, these literals are different:

```
'abc'
' abc'
'abc '
' abc '
```

However, PL/SQL trims trailing blanks when you assign a string literal to a CHAR column.

To put an apostrophe or single quotation mark (') inside a string literal, you have two choices:

- Use two adjacent single quotation marks (which is different from using one double quotation mark):

  ```
  'I''m a string, you''re a string.'
  ```

- Define your own delimiter character for the string:

  ```
  q'delimiter_character string delimiter_character'
  ```

  Where *delimiter_character* is not present in *string*. For example:

  ```
  q'!I'm a string, you're a string.!'
  ```

For Unicode string literals, use nq instead of q. For more information about the Unicode strings, see *Oracle Database Globalization Support Guide*.

### BOOLEAN Literals

A BOOLEAN literal is the predefined value TRUE, FALSE, or NULL. NULL represents an unknown value.

For information about the BOOLEAN data type, see "Predefined PL/SQL BOOLEAN Data Type" on page 3-15.

### Datetime and Interval Literals

The syntax of a datetime or interval literal depends on its data type. Example 2–4 shows some datetime and interval literals. For more information, see "Predefined PL/SQL Datetime and Interval Data Types" on page 3-15.

***Example 2–4 Datetime and Interval Literals***

```
DECLARE
  d1 DATE      := DATE '1998-12-25';
  t1 TIMESTAMP := TIMESTAMP '1997-10-22 13:01:01';

  t2 TIMESTAMP WITH TIME ZONE :=
  TIMESTAMP '1997-01-31 09:26:56.66 +02:00';

  -- Three years and two months
  -- For greater precision, use the day-to-second interval

  i1 INTERVAL YEAR TO MONTH := INTERVAL '3-2' YEAR TO MONTH;

  -- Five days, four hours, three minutes, two and 1/100 seconds

  i2 INTERVAL DAY TO SECOND := INTERVAL '5 04:03:02.01' DAY TO SECOND;

BEGIN
  NULL;
END;
/
```

## Comments

The PL/SQL compiler ignores comments. Adding comments to your program can make it easier to read and understand. Typically, you use comments to describe the purpose and use of each code segment. You can also disable obsolete or unfinished pieces of code by turning them into comments.

Topics:

- Single-Line Comments
- Multiline Comments

**See Also:** "Comment" on page 13-30

### Single-Line Comments

A single-line comment begins with -- and extends to the end of the line.

---

**Caution:** Do not put a single-line comment in a PL/SQL block to be processed dynamically by an Oracle Precompiler program. The Oracle Precompiler program ignores end-of-line characters, which means that a single-line comment ends when the block ends.

---

Example 2–5 has three single-line comments.

***Example 2–5 Single-Line Comments***

```
DECLARE
  howmany     NUMBER;
  num_tables  NUMBER;
BEGIN
  -- Begin processing
  SELECT COUNT(*) INTO howmany
  FROM USER_OBJECTS
  WHERE OBJECT_TYPE = 'TABLE'; -- Check number of tables
  num_tables := howmany;       -- Compute another value
END;
/
```

While testing or debugging a program, you can disable a line of code by making it a comment. For example:

```
-- DELETE FROM employees WHERE comm_pct IS NULL
```

### Multiline Comments

A multiline comment begins with `/*`, ends with `*/`, and can span multiple lines.

Example 2–6 has two multiline comments. (The built-in SQL function `TO_CHAR` returns the character equivalent of its argument. For more information about `TO_CHAR`, see *Oracle Database SQL Language Reference*.)

***Example 2–6 Multiline Comments***

```
DECLARE
  some_condition  BOOLEAN;
  pi              NUMBER := 3.1415926;
  radius          NUMBER := 15;
  area            NUMBER;
BEGIN
  /* Perform some simple tests and assignments */

  IF 2 + 2 = 4 THEN
    some_condition := TRUE;
  /* We expect this THEN to always be performed */
  END IF;

  /* This line computes the area of a circle using pi,
  which is the ratio between the circumference and diameter.
  After the area is computed, the result is displayed. */

  area := pi * radius**2;
  DBMS_OUTPUT.PUT_LINE('The area is: ' || TO_CHAR(area));
END;
/
```

Result:

```
The area is: 706.858335
```

You can use multiline comment delimiters to "comment out" sections of code. When doing so, be careful not to cause nested multiline comments. One multiline comment cannot contain another multiline comment. However, a multiline comment can contain a single-line comment. For example, this causes a syntax error:

```
/*
  IF 2 + 2 = 4 THEN
    some_condition := TRUE;
  /* We expect this THEN to always be performed */
  END IF;
*/
```

This does not cause a syntax error:

```
/*
  IF 2 + 2 = 4 THEN
    some_condition := TRUE;
  -- We expect this THEN to always be performed
  END IF;
*/
```

# Declarations

A declaration allocates storage space for a value of a specified data type, and names the storage location so that you can reference it. You must declare objects before you can reference them. Declarations can appear in the declarative part of any block, subprogram, or package.

Topics:

- Variable Declarations
- Constant Declarations
- Initial Values of Variables and Constants
- NOT NULL Constraint
- %TYPE Attribute
- %ROWTYPE Attribute

For information about declaring objects other than variables and constants, see the syntax of *declare_section* in "Block" on page 13-8.

## Variable Declarations

A variable declaration always specifies the name and data type of the variable. It can also specify an initial value and the NOT NULL constraint.

The name must be a valid user-defined identifier (see "User-Defined Identifiers" on page 2-7).

The data type can be a SQL data type (such as CHAR, DATE, or NUMBER) or a PL/SQL-only data type (such as BOOLEAN or PLS_INTEGER). For information about data types, see Chapter 3, "PL/SQL Data Types".

Example 2–7 declares several variables.

***Example 2–7 Variable Declarations***

```
DECLARE
  part_number       NUMBER(6);     -- SQL data type
  part_name         VARCHAR2(20);  -- SQL data type
  in_stock          BOOLEAN;       -- PL/SQL-only data type
  part_price        NUMBER(6,2);   -- SQL data type
  part_description  VARCHAR2(50);  -- SQL data type
BEGIN
  NULL;
END;
/
```

For variable declaration syntax, see "Variable" on page 13-131.

## Constant Declarations

The information in "Variable Declarations" on page 2-12 also applies to constant declarations, but a constant declaration has two more requirements: the keyword CONSTANT and the initial value of the constant. (The initial value of a constant is its permanent value.)

You can define constants of complex types that have no literal values or predefined constructors, by invoking a function that returns a filled-in value. For example, you can make a constant associative array this way.

Example 2–8 declares three constants.

***Example 2–8 Constant Declarations***

```
DECLARE
  credit_limit     CONSTANT REAL    := 5000.00;  -- SQL data type
  max_days_in_year CONSTANT INTEGER := 366;      -- SQL data type
  urban_legend     CONSTANT BOOLEAN := FALSE     -- PL/SQL-only data type;
BEGIN
  NULL;
END;
/
```

For constant declaration syntax, see "Constant" on page 13-32.

## Initial Values of Variables and Constants

In a variable declaration, the initial value is optional (the default is NULL). In a constant declaration, the initial value is required (and the constant can never have a different value).

The initial value is assigned to the variable or constant every time control passes to the block or subprogram that contains the declaration. If the declaration is in a package specification, the initial value is assigned to the variable or constant once for each session (whether the variable or constant is public or private).

To specify the initial value, use either the assignment operator (:=) or the keyword DEFAULT, followed by an expression. The expression can include previously declared constants and previously initialized variables.

Example 2–9 assigns initial values to the constant and variables that it declares. The initial value of area depends on the previously declared constant pi and the previously initialized variable radius.

***Example 2–9 Variable and Constant Declarations with Initial Values***

```
DECLARE
  hours_worked    INTEGER := 40;
  employee_count  INTEGER := 0;

  pi     CONSTANT REAL := 3.14159;
  radius          REAL := 1;
  area            REAL := (pi * radius**2);
BEGIN
  NULL;
END;
/
```

If you do not specify an initial value for a variable, assign a value to it before using it in any other context.

In Example 2–10, the variable counter has the initial value NULL, by default. As the example shows (using the "IS [NOT] NULL Operator" on page 2-36) NULL is different from zero.

***Example 2–10 Variable Initialized to NULL by Default***

```
DECLARE
  counter INTEGER;  -- initial value is NULL by default
BEGIN
  counter := counter + 1;  -- NULL + 1 is still NULL

  IF counter IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('counter is NULL.');
  END IF;
END;
/
```

Result:

```
counter is NULL.
```

## NOT NULL Constraint

A variable or constant declaration can specify the NOT NULL constraint, which prevents anyone from assigning a null value to the variable or constant.

A variable declaration that specifies NOT NULL must assign an initial value to the variable (because the default initial value for a variable is NULL).

PL/SQL subtypes NATURALN, POSITIVEN, and SIMPLE_INTEGER are predefined as NOT NULL. When declaring a variable of one of these subtypes, you can omit NOT NULL, and you must specify an initial value.

In Example 2–11, all four variables have the NOT NULL constraint.

***Example 2–11 Variable Declaration with NOT NULL Constraint***

```
DECLARE
  acct_id INTEGER(4) NOT NULL := 9999;
  a NATURALN                 := 9999;
  b POSITIVEN                := 9999;
```

```
  c SIMPLE_INTEGER            := 9999;
BEGIN
  NULL;
END;
/
```

PL/SQL treats any zero-length string as a NULL value. This includes values returned by character functions and BOOLEAN expressions.

In Example 2–12, all of the variables are initialized to NULL.

***Example 2–12 Variables Initialized to NULL Values***

```
DECLARE
  null_string  VARCHAR2(80) := TO_CHAR('');
  address      VARCHAR2(80);
  zip_code     VARCHAR2(80) := SUBSTR(address, 25, 0);
  name         VARCHAR2(80);
  valid        BOOLEAN      := (name != '');
BEGIN
  NULL;
END;
/
```

To test for a NULL value, use the "IS [NOT] NULL Operator" on page 2-36.

## %TYPE Attribute

The %TYPE attribute lets you declare a data item to be of the same data type as a previously declared data item or database column. If the declaration of the referenced item changes, the declaration of the referencing item changes accordingly.

The syntax of the declaration is:

```
referencing_item referenced_item%TYPE;
```

For the kinds of data items that can be referencing and referenced items, see "%TYPE Attribute" on page 13-127.

The referencing item inherits the following from the referenced item:

- Data type
- Constraints (unless the referenced item is a database column)

The referencing item does not inherit the initial value of the referenced item. Therefore, if the referencing item specifies or inherits the NOT NULL constraint, you must specify an initial value for it.

The %TYPE attribute is particularly useful when declaring variables to hold database values. The syntax for declaring a variable of the same type as a database column is:

```
variable_name table_name.column_name%TYPE;
```

In Example 2–13, the variable surname inherits the data type of the database column employees.last_name, which has a NOT NULL constraint. Because surname does not inherit the NOT NULL constraint, its declaration does not need an initial value.

***Example 2–13 Declaring Variable of Same Type as Database Column***

```
DECLARE
  surname  employees.last_name%TYPE;
BEGIN
```

```
  DBMS_OUTPUT.PUT_LINE('surname=' || surname);
END;
/
```

Result:

```
surname=
```

In Example 2–14, the variable `surname` inherits the data type and the `NOT NULL` constraint of the variable `name`. Because `surname` does not inherit the initial value of `name`, its declaration needs an initial value.

***Example 2–14 Declaring Variable of Same Type as Another Variable***

```
DECLARE
  name     VARCHAR(25) NOT NULL := 'Smith';
  surname  name%TYPE := 'Jones';
BEGIN
  DBMS_OUTPUT.PUT_LINE('name=' || name);
  DBMS_OUTPUT.PUT_LINE('surname=' || surname);
END;
/
```

Result:

```
name=Smith
surname=Jones
```

## %ROWTYPE Attribute

The `%ROWTYPE` attribute lets you declare a record that represents a row of a database table or view: For every column of the row, the record has a field with the same name and data type. The record can store an entire row that is either selected from the table or view or fetched from a cursor.

The record fields do not inherit the constraints or initial values of the corresponding columns.

If the structure of the row changes, the structure of the record changes accordingly.

The syntax for declaring a record variable to hold a table row is:

```
variable_name table_name%ROWTYPE;
```

For more information about the syntax of `%ROWTYPE`, see "%ROWTYPE Attribute" on page 13-116.

Example 2–15 creates a table, `employees_temp`, and then creates a record, `emprec`, that has a field for every column in the table. Each field (referenced by `emprec.field_name`) inherits the name and data type of the corresponding column, but not the constraints or initial value.

***Example 2–15 Declaring Record that Represents Table Row***

```
CREATE TABLE employees_temp (
  empid    NUMBER(6) NOT NULL PRIMARY KEY
  deptid   NUMBER(6) CONSTRAINT c_employees_temp_deptid
                     CHECK (deptid BETWEEN 100 AND 200),
  deptname VARCHAR2(30) := 'Sales'
);

DECLARE
```

```
  emprec   employees_temp%ROWTYPE;
BEGIN
  emprec.empid := NULL;  -- Null constraint not inherited
  emprec.deptid := 50;   -- Check constraint not inherited
  DBMS_OUTPUT.PUT_LINE ('emprec.deptname: ' || emprec.deptname);
                          -- Initial value not inherited
END;
/
```

Result:

```
emprec.deptname:
```

Example 2–16 declares a cursor, `c1`, which returns two columns of the `departments` table, and then creates a record, `dept_rec`, that has a field for each returned column.

***Example 2–16 Declaring a Record that Represents a Subset of Table Columns***

```
DECLARE
  CURSOR c1 IS
    SELECT department_id, department_name
    FROM departments;

  dept_rec c1%ROWTYPE;  -- Has 2 fields, department_id and department_name

BEGIN
  NULL;
END;
/
```

In Example 2–16, the record `join_rec` has four fields: `employee_id`, `email`, and `manager_id` (which correspond to columns in the `employees` table) and `location_id` (which correspond to a column in the `departments` table).

***Example 2–17 Declaring a Record that Represents a Row from a Join***

```
DECLARE
  CURSOR c2 IS
    SELECT employee_id, email, employees.manager_id, location_id
    FROM employees, departments
    WHERE employees.department_id = departments.department_id;

  join_rec   c2%ROWTYPE;  -- includes columns from two tables

BEGIN
  NULL;
END;
/
```

**Note:** A `%ROWTYPE` declaration cannot include an initial value. For information about assigning values to records, see "Assignments to Records" on page 5-37.

## References to Identifiers

When referencing an identifier, you use a name that is either simple, qualified, remote, or both qualified and remote.

The **simple name** of an identifier is the name in its declaration. For example:

```
DECLARE
  a INTEGER;  -- Declaration
BEGIN
  a := 1;     -- Reference with simple name
END;
/
```

If an identifier is declared in a named PL/SQL unit, you can (and sometimes must) reference it with its **qualified name**. The syntax (called **dot notation**) is:

```
unit_name.simple_identifier_name
```

For example, if package p declares identifier a, you can reference the identifier with the qualified name p.a. The unit name also can (and sometimes must) be qualified. You *must* qualify an identifier when it is not visible (see "Scope and Visibility of Identifiers" on page 2-18).

If the identifier names an object that is stored on a remote database, you must reference it with its **remote name**. The syntax is:

```
simple_identifier_name@link_to_remote_database
```

If the identifier is declared in a PL/SQL unit that is stored on a remote database, you must reference it with its **qualified remote name**. The syntax is:

```
unit_name.simple_identifier_name@link_to_remote_database
```

You can create synonyms for remote schema objects, but you cannot create synonyms for objects declared in PL/SQL subprograms or packages. To create a synonym, use the SQL statement CREATE SYNONYM, described in *Oracle Database SQL Language Reference*.

For information about how PL/SQL resolves ambiguous names, see Appendix B, "PL/SQL Name Resolution".

# Scope and Visibility of Identifiers

The **scope** of an identifier is the region of a PL/SQL unit from which you can reference the identifier. The **visibility** of an identifier is the region of a PL/SQL unit from which you can reference the identifier without qualifying it. An identifier is **local** to the PL/SQL unit that declares it. If that unit has subunits, the identifier is **global** to them.

If a subunit redeclares a global identifier, then inside the subunit, both identifiers are in scope, but only the local identifier is visible. To reference the global identifier, the subunit must qualify it with the name of the unit that declared it. If that unit has no name, then the subunit cannot reference the global identifier.

A PL/SQL unit cannot reference identifiers declared in other units at the same level, because those identifiers are neither local nor global to the block.

Example 2–18 shows the scope and visibility of several identifiers. The first sub-block redeclares the global identifier a. To reference the global variable a, the first sub-block would have to qualify it with the name of the outer block—but the outer block has no name. Therefore, the first sub-block cannot reference the global variable a; it can reference only its local variable a. Because the sub-blocks are at the same level, the first sub-block cannot reference d, and the second sub-block cannot reference c.

***Example 2–18 Scope and Visibility of Identifiers***

```
-- Outer block:
DECLARE
```

```
  a CHAR;  -- Scope of a (CHAR) begins
  b REAL;    -- Scope of b begins
BEGIN
  -- Visible: a (CHAR), b

  -- First sub-block:
  DECLARE
    a INTEGER;  -- Scope of a (INTEGER) begins
    c REAL;       -- Scope of c begins
  BEGIN
    -- Visible: a (INTEGER), b, c
    NULL;
  END;          -- Scopes of a (INTEGER) and c end

  -- Second sub-block:
  DECLARE
    d REAL;     -- Scope of d begins
  BEGIN
    -- Visible: a (CHAR), b, d
    NULL;
  END;          -- Scope of d ends

-- Visible: a (CHAR), b
END;            -- Scopes of a (CHAR) and b end
/
```

Example 2–19 labels the outer block with the name `outer`. Therefore, after the sub-block redeclares the global variable `birthdate`, it can reference that global variable by qualifying its name with the block label. The sub-block can also reference its local variable `birthdate`, by its simple name.

#### *Example 2–19 Qualifying a Redeclared Global Identifier with a Block Label*

```
<<outer>>  -- label
DECLARE
  birthdate DATE := '09-AUG-70';
BEGIN
  DECLARE
    birthdate DATE := '29-SEP-70';
  BEGIN
    IF birthdate = outer.birthdate THEN
      DBMS_OUTPUT.PUT_LINE ('Same Birthday');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('Different Birthday');
    END IF;
  END;
END;
/
```

Result:

```
Different Birthday
```

In Example 2–20, the procedure `check_credit` declares a variable, `rating`, and a function, `check_rating`. The function redeclares the variable. Then the function references the global variable by qualifying it with the procedure name.

#### *Example 2–20 Qualifying an Identifier with a Subprogram Name*

```
CREATE OR REPLACE PROCEDURE check_credit (limit NUMBER) AS
```

```
  rating NUMBER := 3;

  FUNCTION check_rating RETURN BOOLEAN IS
    rating  NUMBER := 1;
    over_limit  BOOLEAN;
  BEGIN
    IF check_credit.rating <= limit THEN  -- reference to global variable
      over_limit := FALSE;
    ELSE
      over_limit := TRUE;
      rating := limit;                    -- reference to local variable
    END IF;
    RETURN over_limit;
  END check_rating;
BEGIN
  IF check_rating THEN
    DBMS_OUTPUT.PUT_LINE
      ('Credit rating over limit (' || TO_CHAR(limit) || ').  '
      || 'Rating: ' || TO_CHAR(rating));
  ELSE
    DBMS_OUTPUT.PUT_LINE
      ('Credit rating OK.  ' || 'Rating: ' || TO_CHAR(rating));
  END IF;
END;
/

BEGIN
  check_credit(1);
END;
/
```

Result:

```
Credit rating over limit (1).  Rating: 3
```

You cannot declare the same identifier twice in the same PL/SQL unit. If you do, an error occurs when you reference the duplicate identifier, as Example 2–21 shows.

***Example 2–21 Duplicate Identifiers in Same Scope***

```
DECLARE
  id  BOOLEAN;
  id  VARCHAR2(5);  -- duplicate identifier
BEGIN
  id := FALSE;
END;
/
```

Result:

```
  id := FALSE;
  *
ERROR at line 5:
ORA-06550: line 5, column 3:
PLS-00371: at most one declaration for 'ID' is permitted
ORA-06550: line 5, column 3:
PL/SQL: Statement ignored
```

You can declare the same identifier in two different units. The two objects represented by the identifier are distinct. Changing one does not affect the other, as Example 2–22 shows.

***Example 2–22 Declaring the Same Identifier in Two Different Units***

```
DECLARE
  PROCEDURE p
  IS
    x VARCHAR2(1);
  BEGIN
    x := 'a';  -- Assign the value 'a' to x
    DBMS_OUTPUT.PUT_LINE('In procedure p, x = ' || x);
  END;

  PROCEDURE q
  IS
    x VARCHAR2(1);
  BEGIN
    x := 'b';  -- Assign the value 'b' to x
    DBMS_OUTPUT.PUT_LINE('In procedure q, x = ' || x);
  END;

BEGIN
  p;
  q;
END;
/
```

Result:

```
In procedure p, x = a
In procedure q, x = b
```

In the same scope, give labels and subprograms unique names to avoid confusion and unexpected results.

In Example 2–23, `echo` is the name of both a block and a subprogram. Both the block and the subprogram declare a variable named `x`. In the subprogram, `echo.x` refers to the local variable `x`, not to the global variable `x`.

***Example 2–23 Label and Subprogram with Same Name in Same Scope***

```
<<echo>>
DECLARE
  x  NUMBER := 5;

  PROCEDURE echo AS
    x  NUMBER := 0;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('x = ' || x);
    DBMS_OUTPUT.PUT_LINE('echo.x = ' || echo.x);
  END;

BEGIN
  echo;
END;
/
```

Result:

```
x = 0
echo.x = 0
```

Example 2–24 has two labels for the outer block, `compute_ratio` and `another_label`. The second label is used again in the inner block. In the inner block,

another_label.denominator refers to the local variable denominator, not to the global variable denominator, which results in the error ZERO_DIVIDE.

***Example 2–24 Block with Multiple and Duplicate Labels***

```
<<compute_ratio>>
<<another_label>>
DECLARE
  numerator   NUMBER := 22;
  denominator NUMBER := 7;
BEGIN
  <<another_label>>
  DECLARE
    denominator NUMBER := 0;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Ratio with compute_ratio.denominator = ');
    DBMS_OUTPUT.PUT_LINE(numerator/compute_ratio.denominator);

    DBMS_OUTPUT.PUT_LINE('Ratio with another_label.denominator = ');
    DBMS_OUTPUT.PUT_LINE(numerator/another_label.denominator);

  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Divide-by-zero error: can''t divide '
        || numerator || ' by ' || denominator);
    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Unexpected error.');
  END another_label;
END compute_ratio;
/
```

Result:

```
Ratio with compute_ratio.denominator =
3.14285714285714285714285714285714285714
Ratio with another_label.denominator =
Divide-by-zero error: cannot divide 22 by 0
```

# Assigning Values to Variables

After declaring a variable, you can assign a value to it in these ways:

- Use the assignment statement to assign it the value of an expression.
- Use the SELECT INTO or FETCH statement to assign it a value from a table.
- Pass it to a subprogram as an OUT or IN OUT parameter, and then assign the value inside the subprogram.

Topics:

- Assigning Values to Variables with the Assignment Statement
- Assigning Values to Variables with the SELECT INTO Statement
- Assigning Values to Variables as Parameters of a Subprogram
- Assigning Values to BOOLEAN Variables

For information about the FETCH statement, see "FETCH Statement" on page 13-67.

## Assigning Values to Variables with the Assignment Statement

To assign the value of an expression to a variable, use this form of the assignment statement:

```
variable_name := expression;
```

For the complete syntax of the assignment statement, see "Assignment Statement" on page 13-3. For the syntax of an expression, see "Expression" on page 13-58.

Example 2–25 declares several variables (specifying initial values for some) and then uses assignment statements to assign the values of expressions to them.

***Example 2–25 Assigning Values to Variables with Assignment Statement***

```
DECLARE  -- You can assign initial values here
  wages          NUMBER;
  hours_worked   NUMBER := 40;
  hourly_salary  NUMBER := 22.50;
  bonus          NUMBER := 150;
  country        VARCHAR2(128);
  counter        NUMBER := 0;
  done           BOOLEAN;
  valid_id       BOOLEAN;
  emp_rec1       employees%ROWTYPE;
  emp_rec2       employees%ROWTYPE;
  TYPE commissions IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
  comm_tab       commissions;

BEGIN  -- You can assign values here too
  wages := (hours_worked * hourly_salary) + bonus;
  country := 'France';
  country := UPPER('Canada');
  done := (counter > 100);
  valid_id := TRUE;
  emp_rec1.first_name := 'Antonio';
  emp_rec1.last_name := 'Ortiz';
  emp_rec1 := emp_rec2;
  comm_tab(5) := 20000 * 0.15;
END;
/
```

## Assigning Values to Variables with the SELECT INTO Statement

A simple form of the `SELECT INTO` statement is:

```
SELECT select_item [, select_item ]...
INTO variable_name [, variable_name ]...
FROM table_name;
```

For each *`select_item`*, there must be a corresponding, type-compatible *`variable_name`*. Because SQL does not have a `BOOLEAN` type, *`variable_name`* cannot be a `BOOLEAN` variable. For the complete syntax of the `SELECT INTO` statement, see "SELECT INTO Statement" on page 13-118.

Example 2–26 uses a `SELECT INTO` statement to assign to the variable `bonus` the value that is 10% of the salary of the employee whose `employee_id` is 100.

***Example 2–26 SELECT INTO Assigns Values to Variables***

```
DECLARE
```

```
  bonus   NUMBER(8,2);
BEGIN
  SELECT salary * 0.10 INTO bonus
  FROM employees
  WHERE employee_id = 100;
END;

DBMS_OUTPUT.PUT_LINE('bonus = ' || TO_CHAR(bonus));
/
```

Result:

```
bonus = 2458.2
```

## Assigning Values to Variables as Parameters of a Subprogram

If you pass a variable to a subprogram as an OUT or IN OUT parameter, and the subprogram assigns a value to the parameter, the variable retains that value after the subprogram finishes running. For more information, see "OUT Mode" on page 8-8 and "IN OUT Mode" on page 8-9.

Example 2–27 passes the variable new_sal to the procedure adjust_salary. The procedure assigns a value to the corresponding formal parameter, sal. Because sal is an IN OUT parameter, the variable new_sal retains the assigned value after the procedure finishes running.

***Example 2–27 Assigning Values to Variables as Parameters of a Subprogram***

```
DECLARE
  new_sal  NUMBER(8,2);
  emp_id   NUMBER(6) := 126;

  PROCEDURE adjust_salary (
    emp_id      NUMBER,
    sal IN OUT NUMBER
  )
  IS
    emp_job  VARCHAR2(10);
    avg_sal  NUMBER(8,2);
  BEGIN
    SELECT job_id INTO emp_job
    FROM employees
    WHERE employee_id = emp_id;

    SELECT AVG(salary) INTO avg_sal
    FROM employees
    WHERE job_id = emp_job;

    DBMS_OUTPUT.PUT_LINE ('The average salary for '
                          || emp_job
                          || ' employees: '
                          || TO_CHAR(avg_sal)
    );

    sal := (sal + avg_sal)/2;
  END;

BEGIN
  -- Assign initial value to new_sal:
```

```
  SELECT AVG(salary)
  INTO new_sal
  FROM employees;

  DBMS_OUTPUT.PUT_LINE ('initial value of new_sal: ' ||
                        TO_CHAR(new_sal)
                       );

  -- Pass new_sal to procedure:

  adjust_salary(emp_id, new_sal);

  DBMS_OUTPUT.PUT_LINE ('final value of new_sal: ' ||
                        TO_CHAR(new_sal)
                       );
END;
/
```

Result:

```
initial value of new_sal: 6461.83
The average salary for ST_CLERK employees: 3281.01
final value of new_sal: 5056.07
```

## Assigning Values to BOOLEAN Variables

The only values that you can assign to a BOOLEAN variable are TRUE, FALSE, and NULL.

Example 2–28 initializes the BOOLEAN variable done to NULL by default, assigns it the literal value FALSE, compares it to the literal value TRUE, and assigns it the value of a BOOLEAN expression.

***Example 2–28 Assigning BOOLEAN Values***

```
DECLARE
  done    BOOLEAN;              -- Initial value is NULL by default
  counter NUMBER := 0;
BEGIN
  done := FALSE;                -- Assign literal value
  WHILE done != TRUE            -- Compare to literal value
    LOOP
      counter := counter + 1;
      done := (counter > 500);  -- Assign value of BOOLEAN expression
    END LOOP;
END;
/
```

For more information about the BOOLEAN data type, see "Predefined PL/SQL BOOLEAN Data Type" on page 3-15.

# Expressions

An expression always returns a single value. The simplest expressions, in order of increasing complexity, are:

1. A single constant or variable (for example, a)
2. A unary operator and its single operand (for example, -a)
3. A binary operator and its two operands (for example, a+b)

An **operand** can be a variable, constant, literal, operator, function call, or placeholder—or another expression. Therefore, expressions can be arbitrarily complex. For expression syntax, see "Expression" on page 13-58.

The data types of the operands determine the data type of the expression. Every time the expression is evaluated, a single value of that data type results.

Topics:

- Concatenation Operator
- Operator Precedence
- Logical Operators
- Short-Circuit Evaluation
- Comparison Operators
- BOOLEAN Expressions
- CASE Expressions
- SQL Functions in PL/SQL Expressions

## Concatenation Operator

The concatenation operator (||) appends one string operand to another, as Example 2–29 shows.

***Example 2–29 Concatenation Operator***

```
DECLARE
  x VARCHAR2(4) := 'suit';
  y VARCHAR2(4) := 'case';
BEGIN
  DBMS_OUTPUT.PUT_LINE (x || y);
END;
/
```

Result:

```
suitcase
```

The concatenation operator ignores null operands, as Example 2–30 shows.

***Example 2–30 Concatenation Operator with NULL Operands***

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('apple' || NULL || NULL || 'sauce');
END;
/
```

Result:

```
applesauce
```

For more information about the syntax of the concatenation operator, see *character_expression ::=* on page 13-60.

## Operator Precedence

An **operation** is either a unary operator and its single operand or a binary operator and its two operands. The operations in an expression are evaluated in order of operator precedence.

Table 2–4 shows operator precedence from highest to lowest. Operators with equal precedence are evaluated in no particular order.

***Table 2–4 Operator Precedence***

| Operator | Operation |
|---|---|
| `**` | exponentiation |
| `+`, `-` | identity, negation |
| `*`, `/` | multiplication, division |
| `+`, `-`, `\|\|` | addition, subtraction, concatenation |
| `=`, `<`, `>`, `<=`, `>=`, `<>`, `!=`, `~=`, `^=`, `IS NULL`, `LIKE`, `BETWEEN`, `IN` | comparison |
| `NOT` | negation |
| `AND` | conjunction |
| `OR` | inclusion |

To control the order of evaluation, enclose operations in parentheses, as in Example 2–31.

***Example 2–31 Controlling Evaluation Order with Parentheses***

```
DECLARE
  a INTEGER := 1+2**2;
  b INTEGER := (1+2)**2;
BEGIN
  DBMS_OUTPUT.PUT_LINE('a = ' || TO_CHAR(a));
  DBMS_OUTPUT.PUT_LINE('b = ' || TO_CHAR(b));
END;
/
```

Result:

```
a = 5
b = 9
```

When parentheses are nested, the most deeply nested operations are evaluated first.

In Example 2–32, the operations (1+2) and (3+4) are evaluated first, producing the values 3 and 7, respectively. Next, the operation 3*7 is evaluated, producing the result 21. Finally, the operation 21/7 is evaluated, producing the final value 3.

***Example 2–32 Expression with Nested Parentheses***

```
DECLARE
  a INTEGER := ((1+2)*(3+4))/7;
BEGIN
  DBMS_OUTPUT.PUT_LINE('a = ' || TO_CHAR(a));
END;
/
```

Result:

```
a = 3
```

You can also use parentheses to improve readability, as in Example 2–33, where the parentheses do not affect evaluation order.

***Example 2–33 Improving Readability with Parentheses***

```
DECLARE
  a INTEGER := 2**2*3**2;
  b INTEGER := (2**2)*(3**2);
BEGIN
  DBMS_OUTPUT.PUT_LINE('a = ' || TO_CHAR(a));
  DBMS_OUTPUT.PUT_LINE('b = ' || TO_CHAR(b));
END;
/
```

Result:

```
a = 36
b = 36
```

Example 2–34 shows the effect of operator precedence and parentheses in several more complex expressions.

***Example 2–34 Operator Precedence***

```
DECLARE
  salary      NUMBER := 60000;
  commission  NUMBER := 0.10;
BEGIN
  -- Division has higher precedence than addition:

  DBMS_OUTPUT.PUT_LINE('5 + 12 / 4 = ' || TO_CHAR(5 + 12 / 4));
  DBMS_OUTPUT.PUT_LINE('12 / 4 + 5 = ' || TO_CHAR(12 / 4 + 5));

 -- Parentheses override default operator precedence:

  DBMS_OUTPUT.PUT_LINE('8 + 6 / 2 = ' || TO_CHAR(8 + 6 / 2));
  DBMS_OUTPUT.PUT_LINE('(8 + 6) / 2 = ' || TO_CHAR((8 + 6) / 2));

  -- Most deeply nested operation is evaluated first:

  DBMS_OUTPUT.PUT_LINE('100 + (20 / 5 + (7 - 3)) = '
                      || TO_CHAR(100 + (20 / 5 + (7 - 3))));

  -- Parentheses, even when unnecessary, improve readability:

  DBMS_OUTPUT.PUT_LINE('(salary * 0.05) + (commission * 0.25) = '
    || TO_CHAR((salary * 0.05) + (commission * 0.25))
  );

  DBMS_OUTPUT.PUT_LINE('salary * 0.05 + commission * 0.25 = '
    || TO_CHAR(salary * 0.05 + commission * 0.25)
  );
END;
/
```

Result:

```
5 + 12 / 4 = 8
12 / 4 + 5 = 8
```

```
8 + 6 / 2 = 11
(8 + 6) / 2 = 7
100 + (20 / 5 + (7 - 3)) = 108
(salary * 0.05) + (commission * 0.25) = 3000.025
salary * 0.05 + commission * 0.25 = 3000.025
```

## Logical Operators

The logical operators AND, OR, and NOT follow the tri-state logic shown in Table 2–5. AND and OR are binary operators; NOT is a unary operator.

***Table 2–5 Logical Truth Table***

| x | y | x AND y | x OR y | NOT x |
|---|---|---|---|---|
| TRUE | TRUE | TRUE | TRUE | FALSE |
| TRUE | FALSE | FALSE | TRUE | FALSE |
| TRUE | NULL | NULL | TRUE | FALSE |
| FALSE | TRUE | FALSE | TRUE | TRUE |
| FALSE | FALSE | FALSE | FALSE | TRUE |
| FALSE | NULL | FALSE | NULL | TRUE |
| NULL | TRUE | NULL | TRUE | NULL |
| NULL | FALSE | FALSE | NULL | NULL |
| NULL | NULL | NULL | NULL | NULL |

As Table 2–5 and Example 2–35 show, AND returns TRUE if and only if both operands are TRUE. (Several examples invoke the print_boolean procedure that Example 2–35 creates. The procedure uses the "IS [NOT] NULL Operator" on page 2-36.)

***Example 2–35 AND Operator***

```
CREATE OR REPLACE PROCEDURE print_boolean (
  name   VARCHAR2,
  value  BOOLEAN
) IS
BEGIN
  IF value IS NULL THEN
    DBMS_OUTPUT.PUT_LINE (name || ' = NULL');
  ELSIF value = TRUE THEN
    DBMS_OUTPUT.PUT_LINE (name || ' = TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE (name || ' = FALSE');
  END IF;
END;
/

DECLARE
  PROCEDURE print_x_and_y (
    x  BOOLEAN,
    y  BOOLEAN
  ) IS
  BEGIN
   print_boolean ('x', x);
   print_boolean ('y', y);
   print_boolean ('x AND y', x AND y);
```

```
  END print_x_and_y;

BEGIN
  print_x_and_y (FALSE, FALSE);
  print_x_and_y (TRUE, FALSE);
  print_x_and_y (FALSE, TRUE);
  print_x_and_y (TRUE, TRUE);

  print_x_and_y (TRUE, NULL);
  print_x_and_y (FALSE, NULL);
  print_x_and_y (NULL, TRUE);
  print_x_and_y (NULL, FALSE);
END;
/
```

Result:

```
x = FALSE
y = FALSE
x AND y = FALSE
x = TRUE
y = FALSE
x AND y = FALSE
x = FALSE
y = TRUE
x AND y = FALSE
x = TRUE
y = TRUE
x AND y = TRUE
x = TRUE
y = NULL
x AND y = NULL
x = FALSE
y = NULL
x AND y = FALSE
x = NULL
y = TRUE
x AND y = NULL
x = NULL
y = FALSE
x AND y = FALSE
```

As Table 2–5 and Example 2–36 show, `OR` returns `TRUE` if either operand is `TRUE`. (Example 2–36 invokes the `print_boolean` procedure from Example 2–35.)

***Example 2–36 OR Operator***

```
DECLARE
  PROCEDURE print_x_or_y (
    x  BOOLEAN,
    y  BOOLEAN
  ) IS
  BEGIN
    print_boolean ('x', x);
    print_boolean ('y', y);
    print_boolean ('x OR y', x OR y);
  END print_x_or_y;

BEGIN
  print_x_or_y (FALSE, FALSE);
  print_x_or_y (TRUE, FALSE);
```

```
  print_x_or_y (FALSE, TRUE);
  print_x_or_y (TRUE, TRUE);

  print_x_or_y (TRUE, NULL);
  print_x_or_y (FALSE, NULL);
  print_x_or_y (NULL, TRUE);
  print_x_or_y (NULL, FALSE);
END;
/
```

Result:

```
x = FALSE
y = FALSE
x OR y = FALSE
x = TRUE
y = FALSE
x OR y = TRUE
x = FALSE
y = TRUE
x OR y = TRUE
x = TRUE
y = TRUE
x OR y = TRUE
x = TRUE
y = NULL
x OR y = TRUE
x = FALSE
y = NULL
x OR y = NULL
x = NULL
y = TRUE
x OR y = TRUE
x = NULL
y = FALSE
x OR y = NULL
```

As Table 2–5 and Example 2–37 show, `NOT` returns the opposite of its operand, unless the operand is `NULL`. `NOT NULL` returns `NULL`, because `NULL` is an indeterminate value. (Example 2–37 invokes the `print_boolean` procedure from Example 2–35.)

***Example 2–37 NOT Operator***

```
DECLARE
  PROCEDURE print_not_x (
    x  BOOLEAN
  ) IS
  BEGIN
    print_boolean ('x', x);
    print_boolean ('NOT x', NOT x);
  END print_not_x;

BEGIN
  print_not_x (TRUE);
  print_not_x (FALSE);
  print_not_x (NULL);
END;
/
```

Result:

```
x = TRUE
NOT x = FALSE
x = FALSE
NOT x = TRUE
x = NULL
NOT x = NULL
```

In Example 2–38, you might expect the sequence of statements to run because x and y seem unequal. But, NULL values are indeterminate. Whether x equals y is unknown. Therefore, the IF condition yields NULL and the sequence of statements is bypassed.

***Example 2–38 NULL Value in Unequal Comparison***

```
DECLARE
  x NUMBER := 5;
  y NUMBER := NULL;
BEGIN
  IF x != y THEN  -- yields NULL, not TRUE
    DBMS_OUTPUT.PUT_LINE('x != y');  -- not run
  ELSIF x = y THEN -- also yields NULL
    DBMS_OUTPUT.PUT_LINE('x = y');
  ELSE
    DBMS_OUTPUT.PUT_LINE
      ('Can''t tell if x and y are equal or not.');
  END IF;
END;
/
```

Result:

```
Can't tell if x and y are equal or not.
```

In Example 2–39, you might expect the sequence of statements to run because a and b seem equal. But, again, that is unknown, so the IF condition yields NULL and the sequence of statements is bypassed.

***Example 2–39 NULL Value in Equal Comparison***

```
DECLARE
  a NUMBER := NULL;
  b NUMBER := NULL;
BEGIN
  IF a = b THEN  -- yields NULL, not TRUE
    DBMS_OUTPUT.PUT_LINE('a = b');  -- not run
  ELSIF a != b THEN  -- yields NULL, not TRUE
    DBMS_OUTPUT.PUT_LINE('a != b');  -- not run
  ELSE
    DBMS_OUTPUT.PUT_LINE('Can''t tell if two NULLs are equal');
  END IF;
END;
/
```

Result:

```
Can't tell if two NULLs are equal
```

In Example 2–40, the two IF statements appear to be equivalent. However, if either x or y is NULL, then the first IF statement assigns the value of y to high and the second IF statement assigns the value of x to high.

***Example 2–40 NOT NULL Equals NULL***

```
DECLARE
  x    INTEGER := 2;
  Y    INTEGER := 5;
  high INTEGER;
BEGIN
  IF (x > y)       -- If x or y is NULL, then (x > y) is NULL
    THEN high := x;  -- run if (x > y) is TRUE
    ELSE high := y;  -- run if (x > y) is FALSE or NULL
  END IF;

  IF NOT (x > y)   -- If x or y is NULL, then NOT (x > y) is NULL
    THEN high := y;  -- run if NOT (x > y) is TRUE
    ELSE high := x;  -- run if NOT (x > y) is FALSE or NULL
  END IF;
END;
/
```

Example 2–41 invokes the `print_boolean` procedure from Example 2–35 three times. The third and first invocation are logically equivalent—the parentheses in the third invocation only improve readability. The parentheses in the second invocation change the order of operation.

***Example 2–41 Changing Evaluation Order of Logical Operators***

```
DECLARE
  x  BOOLEAN := FALSE;
  y  BOOLEAN := FALSE;

BEGIN
  print_boolean ('NOT x AND y', NOT x AND y);
  print_boolean ('NOT (x AND y)', NOT (x AND y));
  print_boolean ('(NOT x) AND y', (NOT x) AND y);
END;
/
```

Result:

```
NOT x AND y = FALSE
NOT (x AND y) = TRUE
(NOT x) AND y = FALSE
```

## Short-Circuit Evaluation

When evaluating a logical expression, PL/SQL uses **short-circuit evaluation**. That is, PL/SQL stops evaluating the expression as soon as it can determine the result. Therefore, you can write expressions that might otherwise cause errors.

In Example 2–42, short-circuit evaluation prevents the `OR` expression from causing a divide-by-zero error. When the value of `on_hand` is zero, the value of the left operand is `TRUE`, so PL/SQL does not evaluate the right operand. If PL/SQL evaluated both operands before applying the `OR` operator, the right operand would cause a division by zero error.

***Example 2–42 Short-Circuit Evaluation***

```
DECLARE
  on_hand  INTEGER := 0;
  on_order INTEGER := 100;
BEGIN
```

```
  -- Does not cause divide-by-zero error;
  -- evaluation stops after first expression

  IF (on_hand = 0) OR ((on_order / on_hand) < 5) THEN
    DBMS_OUTPUT.PUT_LINE('On hand quantity is zero.');
  END IF;
END;
/
```

Result:

```
On hand quantity is zero.
```

## Comparison Operators

Comparison operators compare one expression to another. The result is always either TRUE, FALSE, or NULL. If the value of one expression is NULL, then the result of the comparison is also NULL.

The comparison operators are:

- Relational Operators
- IS [NOT] NULL Operator
- LIKE Operator
- BETWEEN Operator
- IN Operator

> **Note:** Using CLOB values with comparison operators can create temporary LOB values. Ensure that your temporary tablespace is large enough to handle them.

### Relational Operators

Example 2–6 summarizes the relational operators.

***Table 2–6 Relational Operators***

| Operator | Meaning |
|---|---|
| = | equal to |
| <>, !=, ~=, ^= | not equal to |
| < | less than |
| > | greater than |
| <= | less than or equal to |
| >= | greater than or equal to |

Topics:

- Arithmetic Comparisons
- BOOLEAN Comparisons
- Character Comparisons
- Date Comparisons

**Arithmetic Comparisons** One number is greater than another if it represents a larger quantity. Real numbers are stored as approximate values, so Oracle recommends comparing them for equality or inequality.

Example 2–43 invokes the `print_boolean` procedure from Example 2–35 to print the values of expressions that use relational operators to compare arithmetic values.

***Example 2–43 Relational Operators in Expressions***

```
BEGIN
  print_boolean ('(2 + 2 =  4)', 2 + 2 = 4);

  print_boolean ('(2 + 2 <> 4)', 2 + 2 <> 4);
  print_boolean ('(2 + 2 != 4)', 2 + 2 != 4);
  print_boolean ('(2 + 2 ~= 4)', 2 + 2 ~= 4);
  print_boolean ('(2 + 2 ^= 4)', 2 + 2 ^= 4);

  print_boolean ('(1 < 2)', 1 < 2);

  print_boolean ('(1 > 2)', 1 > 2);

  print_boolean ('(1 <= 2)', 1 <= 2);

  print_boolean ('(1 >= 1)', 1 >= 1);
END;
/
```

Result:

```
(2 + 2 =  4) = TRUE
(2 + 2 <> 4) = FALSE
(2 + 2 != 4) = FALSE
(2 + 2 ~= 4) = FALSE
(2 + 2 ^= 4) = FALSE
(1 < 2) = TRUE
(1 > 2) = FALSE
(1 <= 2) = TRUE
(1 >= 1) = TRUE
```

**BOOLEAN Comparisons** By definition, `TRUE` is greater than `FALSE`. Any comparison with `NULL` returns `NULL`.

**Character Comparisons** By default, one character is greater than another if its binary value is larger. For example, this expression is true:

```
'y' > 'r'
```

Strings are compared character by character. For example, this expression is true:

```
'Kathy' > 'Kathryn'
```

If you set the initialization parameter `NLS_COMP=ANSI`, string comparisons use the collating sequence identified by the `NLS_SORT` initialization parameter.

A **collating sequence** is an internal ordering of the character set in which a range of numeric codes represents the individual characters. One character value is greater than another if its internal numeric value is larger. Each language might have different rules about where such characters occur in the collating sequence. For example, an accented letter might be sorted differently depending on the database character set, even though the binary value is the same in each case.

By changing the value of the NLS_SORT parameter, you can perform comparisons that are case-insensitive and accent-insensitive.

A **case-insensitive comparison** treats corresponding uppercase and lowercase letters as the same letter. For example, these expressions are true:

```
'a' = 'A'
'Alpha' = 'ALPHA'
```

To make comparisons case-insensitive, append _CI to the value of the NLS_SORT parameter (for example, BINARY_CI or XGERMAN_CI).

An **accent-insensitive comparison** is case-insensitive, and also treats letters that differ only in accents or punctuation characters as the same letter. For example, these expressions are true:

```
'Cooperate' = 'Co-Operate'
'Co-Operate' = 'coöperate'
```

To make comparisons both case-insensitive and accent-insensitive, append _AI to the value of the NLS_SORT parameter (for example, BINARY_AI or FRENCH_M_AI).

Semantic differences between the CHAR and VARCHAR2 data types affect character comparisons. For more information, see "Comparing Character Values" on page 3-10.

**Date Comparisons** One date is greater than another if it is more recent. For example, this expression is true:

```
'01-JAN-91' > '31-DEC-90'
```

## IS [NOT] NULL Operator

The IS NULL operator returns the BOOLEAN value TRUE if its operand is NULL or FALSE if it is not NULL. The IS NOT NULL operator does the opposite. Comparisons involving NULL values always yield NULL.

To test whether a value is NULL, use IF *value* IS NULL, as in these examples:

- Example 2–10
- Example 2–35
- Example 2–53

## LIKE Operator

The LIKE operator compares a character, string, or CLOB value to a pattern and returns TRUE if the value matches the pattern and FALSE if it does not.

The pattern can include the two **wildcard characters** underscore (_) and percent sign (%). Underscore matches exactly one character. Percent sign (%) matches zero or more characters.

Case is significant. The string 'Johnson' matches the pattern 'J%s_n' but not 'J%S_N', as Example 2–44 shows.

***Example 2–44 LIKE Operator in Expression***

```
DECLARE
  PROCEDURE compare (
    value   VARCHAR2,
    pattern VARCHAR2
  ) IS
  BEGIN
```

```
    IF value LIKE pattern THEN
      DBMS_OUTPUT.PUT_LINE ('TRUE');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('FALSE');
    END IF;
  END;
BEGIN
  compare('Johnson', 'J%s_n');
  compare('Johnson', 'J%S_N');
END;
/
```

Result:

```
TRUE
FALSE
```

To search for the percent sign or underscore, define an escape character and put it before the percent sign or underscore.

Example 2–45 uses the backslash as the escape character, so that the percent sign in the string does not act as a wildcard.

***Example 2–45 Escape Character in Pattern***

```
DECLARE
  PROCEDURE half_off (sale_sign VARCHAR2) IS
  BEGIN
    IF sale_sign LIKE '50\% off!' ESCAPE '\' THEN
      DBMS_OUTPUT.PUT_LINE ('TRUE');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('FALSE');
    END IF;
  END;
BEGIN
  half_off('Going out of business!');
  half_off('50% off!');
END;
/
```

Result:

```
FALSE
TRUE
```

### BETWEEN Operator

The BETWEEN operator tests whether a value lies in a specified range. x BETWEEN a AND b returns the same value as (x>=a) AND (x<=b).

Example 2–46 invokes the print_boolean procedure from Example 2–35 to print the values of expressions that include the BETWEEN operator.

***Example 2–46 BETWEEN Operator in Expressions***

```
BEGIN
  print_boolean ('2 BETWEEN 1 AND 3', 2 BETWEEN 1 AND 3);
  print_boolean ('2 BETWEEN 2 AND 3', 2 BETWEEN 2 AND 3);
  print_boolean ('2 BETWEEN 1 AND 2', 2 BETWEEN 1 AND 2);
  print_boolean ('2 BETWEEN 3 AND 4', 2 BETWEEN 3 AND 4);
END;
/
```

Result:

```
2 BETWEEN 1 AND 3 = TRUE
2 BETWEEN 2 AND 3 = TRUE
2 BETWEEN 1 AND 2 = TRUE
2 BETWEEN 3 AND 4 = FALSE
```

### IN Operator

The IN operator tests set membership. x IN (*set*) returns TRUE only if x equals a member of *set*.

Example 2–47 invokes the print_boolean procedure from Example 2–35 to print the values of expressions that include the IN operator.

***Example 2–47 IN Operator in Expressions***

```
DECLARE
  letter VARCHAR2(1) := 'm';
BEGIN
  print_boolean (
    'letter IN (''a'', ''b'', ''c'')',
    letter IN ('a', 'b', 'c')
  );
  print_boolean (
    'letter IN (''z'', ''m'', ''y'', ''p'')',
    letter IN ('z', 'm', 'y', 'p')
  );
END;
/
```

Result:

```
letter IN ('a', 'b', 'c') = FALSE
letter IN ('z', 'm', 'y', 'p') = TRUE
```

Example 2–48 shows what happens when *set* includes a NULL value. (Example 2–48 invokes the print_boolean procedure from Example 2–35.)

***Example 2–48 IN Operator with Sets with NULL Values***

```
DECLARE
  a INTEGER; -- Initialized to NULL by default
  b INTEGER := 10;
  c INTEGER := 100;
BEGIN
  print_boolean ('100 IN (a, b, c)', 100 IN (a, b, c));
  print_boolean ('100 NOT IN (a, b, c)', 100 NOT IN (a, b, c));

  print_boolean ('100 IN (a, b)', 100 IN (a, b));
  print_boolean ('100 NOT IN (a, b)', 100 NOT IN (a, b));

  print_boolean ('a IN (a, b)', a IN (a, b));
  print_boolean ('a NOT IN (a, b)', a NOT IN (a, b));
END;
/
```

Result:

```
100 IN (a, b, c) = TRUE
100 NOT IN (a, b, c) = FALSE
```

```
100 IN (a, b) = NULL
100 NOT IN (a, b) = NULL
a IN (a, b) = NULL
a NOT IN (a, b) = NULL
```

## BOOLEAN Expressions

A **BOOLEAN expression** is an expression that returns a BOOLEAN value—TRUE, FALSE, or NULL. The simplest BOOLEAN expression is a single BOOLEAN value, constant, or variable. The following are also BOOLEAN expressions:

```
NOT boolean_expression
boolean_expression relational_operator boolean_expression
boolean_expression { AND | OR } boolean_expression
```

For a list of relational operators, see Table 2–6. For the complete syntax of a BOOLEAN expression, see *boolean_expression ::=* on page 13-59.

Typically, you use BOOLEAN expressions as conditions in control statements (which are described in Chapter 4, "PL/SQL Control Statements") and in WHERE clauses of DML statements.

You can use a BOOLEAN variable itself as a condition; you need not compare it to the value TRUE or FALSE. In Example 2–49, the conditions in the loops are equivalent.

***Example 2–49 Equivalent BOOLEAN Expressions as Conditions in Loops***

```
DECLARE
  done BOOLEAN;
BEGIN
  -- These WHILE loops are equivalent

  done := FALSE;
  WHILE done = FALSE
    LOOP
      done := TRUE;
    END LOOP;

  done := FALSE;
  WHILE NOT (done = TRUE)
    LOOP
      done := TRUE;
    END LOOP;

  done := FALSE;
  WHILE NOT done
    LOOP
      done := TRUE;
    END LOOP;
END;
/
```

## CASE Expressions

Topics:

- Simple CASE Expression
- Searched CASE Expression

### Simple CASE Expression

For this explanation, assume that a simple CASE expression has this syntax:

```
CASE selector
WHEN selector_value_1 THEN result_1
WHEN selector_value_2 THEN result_2
...
WHEN selector_value_n THEN result_n
[ ELSE
  else_result ]
END]
```

The *selector* is an expression (typically a single variable). Each *selector_value* and each *result* can be either a literal or an expression.

The simple CASE expression returns the first *result* for which *selector_value* matches *selector*. Remaining expressions are not evaluated. If no *selector_value* matches *selector*, the CASE expression returns *else_result* if it exists and NULL otherwise.

> **See Also:** *simple_case_expression ::=* on page 13-61 for the complete syntax

Example 2–50 assigns the value of a simple CASE expression to the variable appraisal. The *selector* is grade.

***Example 2–50 Simple CASE Expression***

```
DECLARE
  grade CHAR(1) := 'B';
  appraisal VARCHAR2(20);
BEGIN
  appraisal :=
    CASE grade
      WHEN 'A' THEN 'Excellent'
      WHEN 'B' THEN 'Very Good'
      WHEN 'C' THEN 'Good'
      WHEN 'D' THEN 'Fair'
      WHEN 'F' THEN 'Poor'
      ELSE 'No such grade'
    END;
    DBMS_OUTPUT.PUT_LINE ('Grade ' || grade || ' is ' || appraisal);
END;
/
```

Result:

```
Grade B is Very Good
```

If *selector* has the value NULL, it cannot be matched by WHEN NULL, as Example 2–51 shows. Instead, use a searched CASE expression with WHEN *boolean_expression* IS NULL, as in Example 2–53.

***Example 2–51 Simple CASE Expression with WHEN NULL***

```
DECLARE
  grade CHAR(1); -- NULL by default
  appraisal VARCHAR2(20);
BEGIN
  appraisal :=
```

```
  CASE grade
    WHEN NULL THEN 'No grade assigned'
    WHEN 'A' THEN 'Excellent'
    WHEN 'B' THEN 'Very Good'
    WHEN 'C' THEN 'Good'
    WHEN 'D' THEN 'Fair'
    WHEN 'F' THEN 'Poor'
    ELSE 'No such grade'
  END;
  DBMS_OUTPUT.PUT_LINE ('Grade ' || grade || ' is ' || appraisal);
END;
/
```

Result:

```
Grade  is No such grade
```

### Searched CASE Expression

For this explanation, assume that a searched `CASE` expression has this syntax:

```
CASE
WHEN boolean_expression_1 THEN result_1
WHEN boolean_expression_2 THEN result_2
...
WHEN boolean_expression_n THEN result_n
[ ELSE
  else_result ]
END]
```

The searched `CASE` expression returns the first *`result`* for which *`boolean_expression`* is `TRUE`. Remaining expressions are not evaluated. If no *`boolean_expression`* is `TRUE`, the `CASE` expression returns *`else_result`* if it exists and `NULL` otherwise.

> **See Also:** *searched_case_expression ::=* on page 13-62 for the complete syntax

Example 2–52 assigns the value of a searched `CASE` expression to the variable `appraisal`.

***Example 2–52 Searched CASE Expression***

```
DECLARE
  grade      CHAR(1) := 'B';
  appraisal  VARCHAR2(120);
  id         NUMBER  := 8429862;
  attendance NUMBER := 150;
  min_days   CONSTANT NUMBER := 200;

  FUNCTION attends_this_school (id NUMBER)
    RETURN BOOLEAN IS
  BEGIN
    RETURN TRUE;
  END;
BEGIN
  appraisal :=
  CASE
    WHEN attends_this_school(id) = FALSE
      THEN 'Student not enrolled'
```

```
    WHEN grade = 'F' OR attendance < min_days
      THEN 'Poor (poor performance or bad attendance)'
    WHEN grade = 'A' THEN 'Excellent'
    WHEN grade = 'B' THEN 'Very Good'
    WHEN grade = 'C' THEN 'Good'
    WHEN grade = 'D' THEN 'Fair'
    ELSE 'No such grade'
  END;
  DBMS_OUTPUT.PUT_LINE
    ('Result for student ' || id || ' is ' || appraisal);
END;
/
```

Result:

```
Result for student 8429862 is Poor (poor performance or bad attendance)
```

Example 2–53 uses a searched CASE expression to solve the problem in Example 2–51.

***Example 2–53 Searched CASE Expression with WHEN condition IS NULL***

```
DECLARE
  grade CHAR(1); -- NULL by default
  appraisal VARCHAR2(20);
BEGIN
  appraisal :=
    CASE
      WHEN grade IS NULL THEN 'No grade assigned'
      WHEN grade = 'A' THEN 'Excellent'
      WHEN grade = 'B' THEN 'Very Good'
      WHEN grade = 'C' THEN 'Good'
      WHEN grade = 'D' THEN 'Fair'
      WHEN grade = 'F' THEN 'Poor'
      ELSE 'No such grade'
    END;
    DBMS_OUTPUT.PUT_LINE ('Grade ' || grade || ' is ' || appraisal);
END;
/
```

Result:

```
Grade  is No grade assigned
```

## SQL Functions in PL/SQL Expressions

In PL/SQL expressions, you can use all SQL functions except:

- Aggregate functions (such as AVG and COUNT)
- Analytic functions (such as LAG and RATIO_TO_REPORT)
- Collection functions (such as CARDINALITY and SET)
- Data mining functions (such as CLUSTER_ID and FEATURE_VALUE)
- Encoding and decoding functions (such as DECODE and DUMP)
- Model functions (such as ITERATION_NUMBER and PREVIOUS)
- Object reference functions (such as REF and VALUE)
- XML functions (such as APPENDCHILDXML and EXISTSNODE)
- These conversion functions:

- BIN_TO_NUM
- CAST
- RAWTONHEX
- ROWIDTONCHAR

- These miscellaneous functions:
  - CUBE_TABLE
  - DATAOBJ_TO_PARTITION
  - LNNVL
  - SYS_CONNECT_BY_PATH
  - SYS_TYPEID
  - WIDTH_BUCKET

PL/SQL supports an overload of BITAND for which the arguments and result are BINARY_INTEGER.

When used in a PL/SQL expression, the RAWTOHEX function accepts an argument of data type RAW and returns a VARCHAR2 value with the hexadecimal representation of bytes that comprise the value of the argument. Arguments of types other than RAW can be specified only if they can be implicitly converted to RAW. This conversion is possible for CHAR, VARCHAR2, and LONG values that are valid arguments of the HEXTORAW function, and for LONG RAW and BLOB values of up to 16380 bytes.

## Error-Reporting Functions

PL/SQL has two built-in error-reporting functions, SQLCODE and SQLERRM, for use in PL/SQL exception-handling code. For their descriptions, see "SQLCODE Function" on page 13-124 and "SQLERRM Function" on page 13-125.

You cannot use the SQLCODE and SQLERRM functions in SQL statements.

## Pragmas

A **pragma** is an instruction to the compiler. It is processed at compile time, not at run time. The basic syntax is:

```
PRAGMA instruction
```

For the complete syntax, see *pragma ::=* on page 13-10.

## Conditional Compilation

Conditional compilation lets you customize the functionality of a PL/SQL application without removing source code. For example, you can:

- Use new features with the latest database release and disable them when running the application in an older database release.
- Activate debugging or tracing statements in the development environment and hide them when running the application at a production site.

Topics:

- How Conditional Compilation Works

- Conditional Compilation Examples
- Retrieving and Printing Post-Processed Source Text
- Conditional Compilation Directive Restrictions

## How Conditional Compilation Works

---

**Note:** The conditional compilation feature and related PL/SQL packages are available for Oracle Database release 10.1.0.4 and later releases.

---

Conditional compilation uses selection directives, which are similar to IF statements, to select source text for compilation. The condition in a selection directive usually includes an inquiry directive. Error directives raise user-defined errors. All conditional compilation directives are built from preprocessor control tokens and PL/SQL text.

Topics:

- Preprocessor Control Tokens
- Selection Directives
- Error Directives
- Inquiry Directives
- Static Expressions

### Preprocessor Control Tokens

A preprocessor control token identifies code that is processed before the PL/SQL unit is compiled.

#### Syntax

```
$plsql_identifier
```

There cannot be space between $ and *plsql_identifier*. For information about *plsql_identifier*, see "Identifiers" on page 2-5. The character $ can also appear inside *plsql_identifier*, but it has no special meaning there.

These preprocessor control tokens are reserved:

- $IF
- $THEN
- $ELSE
- $ELSIF
- $ERROR

### Selection Directives

A **selection directive** selects source text to compile.

#### Syntax

```
$IF boolean_static_expression $THEN
   text
[ $ELSIF boolean_static_expression $THEN
```

```
   text
]...
[ $ELSE
   text
$END
]
```

For the syntax of *boolean_static_expression*, see "BOOLEAN Static Expressions" on page 2-49. The *text* can be anything, but typically, it is either a statement (described in "statement ::=" on page 13-11) or an error directive (described in "Error Directives" on page 2-45).

The selection directive evaluates the BOOLEAN static expressions in the order that they appear until either one expression has the value TRUE or the list of expressions is exhausted. If one expression has the value TRUE, its text is selected for compilation, the remaining expressions are not evaluated, and their text is not analyzed. If no expression has the value TRUE, then if $ELSE is present, its text is selected for compilation; otherwise, no text is selected for compilation.

For examples of selection directives, see "Conditional Compilation Examples" on page 2-51.

> **See Also:** "Conditional Selection Statements" on page 4-1 for information about the IF statement, which has the same logic as the selection directive

### Error Directives

An **error directive** produces a user-defined error message during compilation.

#### Syntax

```
$ERROR varchar2_static_expression $END
```

It produces this compile-time error message, where *string* is the value of *varchar2_static_expression*:

```
PLS-00179: $ERROR: string
```

For the syntax of *varchar2_static_expression*, see "VARCHAR2 Static Expressions" on page 2-49.

For an example of an error directive, see Example 2–58.

### Inquiry Directives

An **inquiry directive** provides information about the compilation environment.

#### Syntax

```
$$name
```

For information about *name*, which is an unquoted PL/SQL identifier, see "Identifiers" on page 2-5.

An inquiry directive typically appears in the *boolean_static_expression* of a selection directive, but it can appear anywhere that a variable or literal of its type can appear. Moreover, it can appear where regular PL/SQL allows only a literal (not a variable)— for example, to specify the size of a VARCHAR2 variable.

Topics:

- Predefined Inquiry Directives

- Assigning Values to Inquiry Directives
- Unresolvable Inquiry Directives

**Predefined Inquiry Directives** The predefined inquiry directives are:

- `$$PLSQL_LINE`

  A `PLS_INTEGER` literal whose value is the number of the source line on which the directive appears in the current PL/SQL unit. An example of `$$PLSQL_LINE` in a selection directive is:

  ```
  $IF $$PLSQL_LINE = 32 $THEN ...
  ```

- `$$PLSQL_UNIT`

  A `VARCHAR2` literal that contains the name of the current PL/SQL unit. If the current PL/SQL unit is an anonymous block, `$$PLSQL_UNIT` contains a `NULL` value. An example of `$$PLSQL_UNIT` in a selection directive is:

  ```
  $IF $$PLSQL_UNIT IS NULL $THEN ...
  ```

  Because a selection directive needs a `BOOLEAN` static expression, you cannot use a `VARCHAR2` comparison such as:

  ```
  $IF $$PLSQL_UNIT = 'AWARD_BONUS' $THEN ...
  ```

- `$$`*`plsql_compilation_parameter`*

  The name *`plsql_compilation_parameter`* is a PL/SQL compilation parameter (for example, `PLSCOPE_SETTINGS`). For descriptions of these parameters, see Table 1–1.

Example 2–54, a SQL*Plus script, uses the predefined inquiry directives `$$PLSQL_LINE` and `$$PLSQL_UNIT` as ordinary `PLS_INTEGER` and `VARCHAR2` literals, respectively, to show how their values are assigned.

***Example 2–54 Predefined Inquiry Directives $$PLSQL_LINE and $$PLSQL_UNIT***

```
SQL> CREATE OR REPLACE PROCEDURE p
  2  IS
  3    i PLS_INTEGER;
  4  BEGIN
  5    DBMS_OUTPUT.PUT_LINE('Inside p');
  6    i := $$PLSQL_LINE;
  7    DBMS_OUTPUT.PUT_LINE('i = ' || i);
  8    DBMS_OUTPUT.PUT_LINE('$$PLSQL_LINE = ' || $$PLSQL_LINE);
  9    DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT = ' || $$PLSQL_UNIT);
 10  END;
 11  /

Procedure created.

SQL> BEGIN
  2   p;
  3   DBMS_OUTPUT.PUT_LINE('Outside p');
  4   DBMS_OUTPUT.PUT_LINE('$$PLSQL_UNIT = ' || $$PLSQL_UNIT);
  5  END;
  6  /
```

Result:

```
Inside p
```

```
i = 6
$$PLSQL_LINE = 8
$$PLSQL_UNIT = P
Outside p
$$PLSQL_UNIT =

PL/SQL procedure successfully completed.
```

Example 2–55 displays the current values of PL/SQL the compilation parameters.

***Example 2–55 Displaying Values of PL/SQL Compilation Parameters***

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('$$PLSCOPE_SETTINGS = '     || $$PLSCOPE_SETTINGS);
  DBMS_OUTPUT.PUT_LINE('$$PLSQL_CCFLAGS = '        || $$PLSQL_CCFLAGS);
  DBMS_OUTPUT.PUT_LINE('$$PLSQL_CODE_TYPE = '      || $$PLSQL_CODE_TYPE);
  DBMS_OUTPUT.PUT_LINE('$$PLSQL_OPTIMIZE_LEVEL = ' || $$PLSQL_OPTIMIZE_LEVEL);
  DBMS_OUTPUT.PUT_LINE('$$PLSQL_WARNINGS = '       || $$PLSQL_WARNINGS);
  DBMS_OUTPUT.PUT_LINE('$$NLS_LENGTH_SEMANTICS = ' || $$NLS_LENGTH_SEMANTICS);
END;
/
```

Result:

```
$$PLSCOPE_SETTINGS =
$$PLSQL_CCFLAGS =
$$PLSQL_CODE_TYPE = INTERPRETED
$$PLSQL_OPTIMIZE_LEVEL = 2
$$PLSQL_WARNINGS = ENABLE:ALL
$$NLS_LENGTH_SEMANTICS = BYTE
```

> **Note:** In the SQL*Plus environment, you can display the current values of initialization parameters, including the PL/SQL compilation parameters, with the command `SHOW PARAMETERS`. For more information about the `SHOW` command and its `PARAMETERS` option, see *SQL*Plus User's Guide and Reference*.

**Assigning Values to Inquiry Directives** You can assign values to inquiry directives with the `PLSQL_CCFLAGS` compilation parameter. For example:

```
ALTER SESSION SET PLSQL_CCFLAGS =
  'name1:value1, name2:value2, ... namen:valuen'
```

Each *`value`* must be either a `BOOLEAN` literal (`TRUE`, `FALSE`, or `NULL`) a or `PLS_INTEGER` literal. The data type of *`value`* determines the data type of *`name`*.

The same *`name`* can appear multiple times, with values of the same or different data types. Later assignments override earlier assignments. For example, this command sets the value of `$$flag` to 5 and its data type to `PLS_INTEGER`:

```
ALTER SESSION SET PLSQL_CCFLAGS = 'flag:TRUE, flag:5'
```

Oracle recommends against using `PLSQL_CCFLAGS` to assign values to predefined inquiry directives, including compilation parameters. To assign values to compilation parameters, Oracle recommends using the `ALTER SESSION` statement. For more information about the `ALTER SESSION` statement, see *Oracle Database SQL Language Reference*.

Example 2–56 uses `PLSQL_CCFLAGS` to assign a value to the user-defined inquiry directive `$$Some_Flag` and (though not recommended) to itself. Because later

assignments override earlier assignments, the resulting value of `$$Some_Flag` is 2 and the resulting value of `PLSQL_CCFLAGS` is the value that it assigns to itself (99), not the value that the `ALTER SESSION` statement assigns to it (`'Some_Flag:1, Some_Flag:2, PLSQL_CCFlags:99'`).

***Example 2–56 PLSQL_CCFLAGS Assigns Value to Itself***

```
ALTER SESSION SET
PLSQL_CCFlags = 'Some_Flag:1, Some_Flag:2, PLSQL_CCFlags:99'
/
BEGIN
  Print($$Some_Flag);
  Print($$PLSQL_CCFlags);
END;
/
```

Result:

```
2
99
```

---

**Note:** The compile-time value of `PLSQL_CCFLAGS` is stored with the metadata of stored PL/SQL units, which means that you can reuse the value when you explicitly recompile the units. For more information, see "PL/SQL Units and Compilation Parameters" on page 1-11.

---

For more information about `PLSQL_CCFLAGS`, see *Oracle Database Reference*.

**Unresolvable Inquiry Directives** If an inquiry directive (`$$`*`name`*) cannot be resolved (that is, if its value cannot be determined) and the source text is not wrapped, then warning `PLW-6003` is generated and `NULL` is substituted for the value of the unresolved inquiry directive. If the source text is wrapped, the warning message is disabled, so that the unresolved inquiry directive is not revealed. For information about wrapping PL/SQL source text, see Appendix A, "PL/SQL Source Code Wrapping".

## Static Expressions

A **static expression** is an expression whose value can be determined at compilation time (that is, it does not include references to variables or functions). Static expressions are the only expressions that can appear in conditional compilation directives.

Topics:

- PLS_INTEGER Static Expressions
- BOOLEAN Static Expressions
- VARCHAR2 Static Expressions
- Static Constants
- DBMS_DB_VERSION Package

**See Also:** "Expressions" on page 2-25 for general information about expressions

**PLS_INTEGER Static Expressions** `PLS_INTEGER` static expressions are:

- `PLS_INTEGER` literals

For information about numeric literals, see "Numeric Literals" on page 2-8.

- PLS_INTEGER static constants

  For information about static constants, see "Static Constants" on page 2-50.

- NULL

> **See Also:** "PLS_INTEGER and BINARY_INTEGER Data Types" on page 3-3 for information about the PLS_INTEGER data type

**BOOLEAN Static Expressions** BOOLEAN static expressions are:

- BOOLEAN literals (TRUE, FALSE, or NULL)
- BOOLEAN static constants

  For information about static constants, see "Static Constants" on page 2-50.

- Where *x* and *y* are PLS_INTEGER static expressions:
  - *x* > *y*
  - *x* < *y*
  - *x* >= *y*
  - *x* <= *y*
  - *x* = *y*
  - *x* <> *y*

  For information about PLS_INTEGER static expressions, see "PLS_INTEGER Static Expressions" on page 2-48.

- Where *x* and *y* are BOOLEAN expressions:
  - NOT *y*
  - *x* AND *y*
  - *x* OR *y*
  - *x* > *y*
  - *x* >= *y*
  - *x* = *y*
  - *x* <= *y*
  - *x* <> *y*

  For information about BOOLEAN expressions, see "BOOLEAN Expressions" on page 2-39.

- Where x is a static expression:
  - *x* IS NULL
  - *x* IS NOT NULL

  For information about static expressions, see "Static Expressions" on page 2-48.

> **See Also:** "Predefined PL/SQL BOOLEAN Data Type" on page 3-15 for information about the BOOLEAN data type

**VARCHAR2 Static Expressions** VARCHAR2 static expressions are:

- String literal with maximum size of 32,767 bytes

  For information about string literals, see "String Literals" on page 2-9.
- NULL
- TO_CHAR(x), where x is a PLS_INTEGER static expression

  For information about the TO_CHAR function, see *Oracle Database SQL Language Reference*.
- TO_CHAR(x, f, n) where x is a PLS_INTEGER static expression and f and n are VARCHAR2 static expressions

  For information about the TO_CHAR function, see *Oracle Database SQL Language Reference*.
- x || y where x and y are VARCHAR2 or PLS_INTEGER static expressions

  For information about PLS_INTEGER static expressions, see "PLS_INTEGER Static Expressions" on page 2-48.

> **See Also:** "CHAR and VARCHAR2 Data Types" on page 3-8 for information about the VARCHAR2 data type

**Static Constants** A **static constant** is declared in a package specification with this syntax:

```
constant_name CONSTANT data_type := static_expression;
```

The type of *static_expression* must be the same as *data_type* (either BOOLEAN or PLS_INTEGER).

The static constant must always be referenced as *package_name.constant_name*, even in the body of the *package_name* package.

If you use *constant_name* in the BOOLEAN expression in a conditional compilation directive in a PL/SQL unit, then the PL/SQL unit depends on the package *package_name*. If you alter the package specification, the dependent PL/SQL unit might become invalid and need to be recompiled (for information about the invalidation of dependent objects, see *Oracle Database Advanced Application Developer's Guide*).

If you use a package with static constants to control conditional compilation in multiple PL/SQL units, Oracle recommends that you create only the package specification, and dedicate it exclusively to controlling conditional compilation. This practice minimizes invalidations caused by altering the package specification.

To control conditional compilation in a single PL/SQL unit, you can set flags in the PLSQL_CCFLAGS compilation parameter. For information about this parameter, see "Assigning Values to Inquiry Directives" on page 2-47 and *Oracle Database Reference*.

In Example 2–57, the package my_debug defines the static constants debug and trace to control debugging and tracing in multiple PL/SQL units. The procedure my_proc1 uses only debug, and the procedure my_proc2 uses only trace, but both procedures depend on the package. However, the recompiled code might not be different. For example, if you only change the value of debug to FALSE and then recompile the two procedures, the compiled code for my_proc1 changes, but the compiled code for my_proc2 does not.

***Example 2–57 Static Constants***

```
CREATE PACKAGE my_debug IS
  debug CONSTANT BOOLEAN := TRUE;
```

```
  trace CONSTANT BOOLEAN := TRUE;
END my_debug;
/

CREATE PROCEDURE my_proc1 IS
BEGIN
  $IF my_debug.debug $THEN
    DBMS_OUTPUT.put_line('Debugging ON');
  $ELSE
    DBMS_OUTPUT.put_line('Debugging OFF');
  $END
END my_proc1;
/

CREATE PROCEDURE my_proc2 IS
BEGIN
  $IF my_debug.trace $THEN
    DBMS_OUTPUT.put_line('Tracing ON');
  $ELSE
    DBMS_OUTPUT.put_line('Tracing OFF');
  $END
END my_proc2;
/
```

**See Also:**

- "Constant Declarations" on page 2-13 for general information about declaring constants
- Chapter 10, "PL/SQL Packages" for more information about packages
- *Oracle Database Advanced Application Developer's Guide* for more information about schema object dependencies

**DBMS_DB_VERSION Package** The DBMS_DB_VERSION package provides these static constants:

- The PLS_INTEGER constant VERSION identifies the current Oracle Database version.
- The PLS_INTEGER constant RELEASE identifies the current Oracle Database release number.
- Each BOOLEAN constant of the form VER_LE_*v* has the value TRUE if the database version is less than or equal to *v*; otherwise, it has the value FALSE.
- Each BOOLEAN constant of the form VER_LE_*v*_*r* has the value TRUE if the database version is less than or equal to *v* and release is less than or equal to *r*; otherwise, it has the value FALSE.
- All constants representing Oracle Database 10*g* release 1 or earlier have the value FALSE.

For more information about the DBMS_DB_VERSION package, see *Oracle Database PL/SQL Packages and Types Reference*.

## Conditional Compilation Examples

Example 2–58 generates an error message if the database version and release is less than 10.2; otherwise, it displays a message saying that the version and release are supported and uses a COMMIT statement that became available at release 10.2.

***Example 2–58 Code for Checking Database Version***

```
BEGIN
  $IF DBMS_DB_VERSION.VER_LE_10_1 $THEN  -- selection directive begins
    $ERROR 'unsupported database release' $END  -- error directive
  $ELSE
    DBMS_OUTPUT.PUT_LINE (
      'Release ' || DBMS_DB_VERSION.VERSION || '.' ||
      DBMS_DB_VERSION.RELEASE || ' is supported.'
    );
  -- This COMMIT syntax is newly supported in 10.2:
  COMMIT WRITE IMMEDIATE NOWAIT;
  $END  -- selection directive ends
END;
/
```

Result:

```
Release 11.1 is supported.
```

Example 2–59 sets the values of the user-defined inquiry directives `$$my_debug` and `$$my_tracing` and then uses conditional compilation:

- In the specification of package `my_pkg`, to determine the base type of the subtype `my_real` (`BINARY_DOUBLE` is available only for Oracle Database versions 10*g* and later.)
- In the body of package `my_pkg`, to compute the values of `my_pi` and `my_e` differently for different database versions
- In the procedure `circle_area`, to compile some code only if the inquiry directive `$$my_debug` has the value `TRUE`.

***Example 2–59 Compiling Different Code for Different Database Versions***

```
ALTER SESSION SET PLSQL_CCFLAGS = 'my_debug:FALSE, my_tracing:FALSE';

CREATE OR REPLACE PACKAGE my_pkg AS
  SUBTYPE my_real IS
    $IF DBMS_DB_VERSION.VERSION < 10 $THEN
      NUMBER;
    $ELSE
      BINARY_DOUBLE;
    $END

  my_pi my_real;
  my_e  my_real;
END my_pkg;
/

CREATE OR REPLACE PACKAGE BODY my_pkg AS
BEGIN
  $IF DBMS_DB_VERSION.VERSION < 10 $THEN
    my_pi := 3.1415926535897932384626433832795028842O;
    my_e  := 2.7182818284590452353602874713526624977 5;
  $ELSE
    my_pi := 3.14159265358979323846264338327950288420d;
    my_e  := 2.71828182845904523536028747135266249775d;
  $END
END my_pkg;
/
```

```
CREATE OR REPLACE PROCEDURE circle_area(radius my_pkg.my_real) IS
  my_area       my_pkg.my_real;
  my_data_type  VARCHAR2(30);
BEGIN
  my_area := my_pkg.my_pi * (radius**2);

  DBMS_OUTPUT.PUT_LINE
    ('Radius: ' || TO_CHAR(radius) || ' Area: ' || TO_CHAR(my_area));

  $IF $$my_debug $THEN
    SELECT DATA_TYPE INTO my_data_type
    FROM USER_ARGUMENTS
    WHERE OBJECT_NAME = 'CIRCLE_AREA'
    AND ARGUMENT_NAME = 'RADIUS';

    DBMS_OUTPUT.PUT_LINE
      ('Data type of the RADIUS argument is: ' || my_data_type);
  $END
END;
/
```

## Retrieving and Printing Post-Processed Source Text

The DBMS_PREPROCESSOR package provides subprograms that retrieve and print the source text of a PL/SQL unit in its post-processed form. For information about the DBMS_PREPROCESSOR package, see *Oracle Database PL/SQL Packages and Types Reference*.

Example 2–60 invokes the procedure DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE to print the post-processed form of my_pkg (from Example 2–59). Lines of code in Example 2–59 that are not included in the post-processed text appear as blank lines.

***Example 2–60 Displaying Post-Processed Source Code***

```
CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE (
  'PACKAGE', 'HR', 'MY_PKG'
);
```

Result:

```
PACKAGE my_pkg AS
SUBTYPE my_real IS
BINARY_DOUBLE;
my_pi my_real;
my_e my_real;
END my_pkg;
```

## Conditional Compilation Directive Restrictions

A conditional compilation directive cannot appear in the specification of a schema-level user-defined type (created with the "CREATE TYPE Statement" on page 14-69). This type specification specifies the attribute structure of the type, which determines the attribute structure of dependent types and the column structure of dependent tables.

---

**Caution:** Using a conditional compilation directive to change the attribute structure of a type can cause dependent objects to "go out of sync" or dependent tables to become inaccessible. Oracle recommends that you change the attribute structure of a type only with the "ALTER TYPE Statement" on page 14-17. The `ALTER TYPE` statement propagates changes to dependent objects.

---

The SQL parser imposes these restrictions on the location of the first conditional compilation directive in a stored PL/SQL unit or anonymous block:

- In a package specification, a package body, a type body, and in a schema-level subprogram with no formal parameters, the first conditional compilation directive cannot appear before the keyword `IS` or `AS`.
- In a schema-level subprogram with at least one formal parameter, the first conditional compilation directive cannot appear before the left parenthesis that follows the subprogram name.

  This example is correct:

  ```
  CREATE OR REPLACE PROCEDURE my_proc (
    $IF $$xxx $THEN i IN PLS_INTEGER $ELSE i IN INTEGER $END
  ) IS BEGIN NULL; END my_proc;
  /
  ```

- In a trigger or an anonymous block, the first conditional compilation directive cannot appear before the keyword `DECLARE` or `BEGIN`, whichever comes first.

The SQL parser also imposes this restriction: If an anonymous block uses a placeholder, the placeholder cannot appear in a conditional compilation directive. For example:

```
BEGIN
  :n := 1; -- valid use of placeholder
  $IF ... $THEN
    :n := 1; -- invalid use of placeholder
$END
```

# 3

# PL/SQL Data Types

Every constant, variable, and parameter has a **data type** (often called a **type**) that determines its storage format, constraints, valid range of values, and operations that can be performed on it. PL/SQL provides many predefined data types and subtypes, and lets you define your own PL/SQL subtypes.

Table 3–1 lists the categories of predefined PL/SQL data types, describes the data they store, and tells where to find information about the specialized data types.

***Table 3–1 Categories of Predefined PL/SQL Data Types***

| Data Type Category | Data Description |
|---|---|
| Scalar | Single values with no internal components. |
| Composite | Data items that have internal components that can be accessed individually. Explained in Chapter 5, "PL/SQL Collections and Records." |
| Reference | Pointers to other data items. Explained in "Cursor Variables" on page 6-25. |
| Large Object (LOB) | Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms. |

Topics:

- Subtypes
- Predefined PL/SQL Scalar Data Types and Subtypes
- Predefined PL/SQL Large Object (LOB) Data Types
- User-Defined PL/SQL Subtypes
- PL/SQL Data Type Conversion

**See Also:**

- Chapter 5, "PL/SQL Collections and Records" for information about defining collection and record types
- "CREATE TYPE Statement" on page 14-69 for information about creating ADTs and other schema-level user-defined data types

## Subtypes

A **subtype** is a subset of another data type, which is called its **base type**. A subtype has the same valid operations as its base type, but only a subset of its valid values.

Subtypes can increase reliability, provide compatibility with ANSI/ISO types, and improve readability by indicating the intended use of constants and variables.

An **unconstrained subtype** has the same set of values as its base type. For example:

```
SUBTYPE CHARACTER IS CHAR;
```

A **constrained subtype** has only a subset of the values of its base type. For example:

```
SUBTYPE INTEGER IS NUMBER(38,0);  -- only whole numbers
```

PL/SQL predefines the subtypes CHAR and INTEGER (and many others) in the package STANDARD.

# Predefined PL/SQL Scalar Data Types and Subtypes

Scalar data types store single values with no internal components. Table 3–2 lists the predefined PL/SQL scalar data types and describes the data they store.

***Table 3–2 Categories of Predefined PL/SQL Scalar Data Types***

| Category | Data Description |
|---|---|
| Numeric | Numeric values, on which you can perform arithmetic operations. |
| Character | Alphanumeric values that represent single characters or strings of characters, which you can manipulate. |
| BOOLEAN | Logical values, on which you can perform logical operations. |
| Datetime | Dates and times, which you can manipulate. |
| Interval | Time intervals, which you can manipulate. |

Topics:

- Predefined PL/SQL Numeric Data Types and Subtypes
- Predefined PL/SQL Character Data Types and Subtypes
- Predefined PL/SQL BOOLEAN Data Type
- Predefined PL/SQL Datetime and Interval Data Types

## Predefined PL/SQL Numeric Data Types and Subtypes

Numeric data types let you store numeric data, represent quantities, and perform calculations. Table 3–3 lists the predefined PL/SQL numeric types and describes the data they store.

***Table 3–3 Predefined PL/SQL Numeric Data Types***

| Data Type | Data Description |
|---|---|
| PLS_INTEGER or BINARY_INTEGER | Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits |
| BINARY_FLOAT | Single-precision IEEE 754-format floating-point number |
| BINARY_DOUBLE | Double-precision IEEE 754-format floating-point number |
| NUMBER | Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0. |

Topics:

- PLS_INTEGER and BINARY_INTEGER Data Types
- SIMPLE_INTEGER Subtype of PLS_INTEGER
- BINARY_FLOAT and BINARY_DOUBLE Data Types
- NUMBER Data Type

### PLS_INTEGER and BINARY_INTEGER Data Types

The PLS_INTEGER and BINARY_INTEGER data types are identical. For simplicity, this document uses "PLS_INTEGER" to mean both PLS_INTEGER and BINARY_INTEGER.

The PLS_INTEGER data type stores signed integers in the range -2,147,483,648 through 2,147,483,647, represented in 32 bits.

The PLS_INTEGER data type has these advantages over the NUMBER data type and NUMBER subtypes:

- PLS_INTEGER values require less storage.
- PLS_INTEGER operations use hardware arithmetic, so they are faster than NUMBER operations, which use library arithmetic.

For efficiency, use PLS_INTEGER values for all calculations that fall in its range. For calculations outside the PLS_INTEGER range, use INTEGER, a predefined subtype of the NUMBER data type.

---

**Note:** When a calculation with two PLS_INTEGER data types overflows the PLS_INTEGER range, an overflow exception is raised even if the result is assigned to a NUMBER data type.

---

Table 3–4 lists the predefined subtypes of the PLS_INTEGER data type and describes the data they store.

***Table 3–4 Predefined Subtypes of PLS_INTEGER Data Type***

| Data Type | Data Description |
|---|---|
| NATURAL | Nonnegative PLS_INTEGER value |
| NATURALN | Nonnegative PLS_INTEGER value with NOT NULL constraint |
| POSITIVE | Positive PLS_INTEGER value |
| POSITIVEN | Positive PLS_INTEGER value with NOT NULL constraint |
| SIGNTYPE | PLS_INTEGER value -1, 0, or 1 (useful for programming tri-state logic) |
| SIMPLE_INTEGER | PLS_INTEGER value with NOT NULL constraint |

### SIMPLE_INTEGER Subtype of PLS_INTEGER

SIMPLE_INTEGER is a predefined subtype of the PLS_INTEGER data type that has the same range as PLS_INTEGER (-2,147,483,648 through 2,147,483,647) and has a NOT NULL constraint. It differs significantly from PLS_INTEGER in its overflow semantics.

You can use SIMPLE_INTEGER when the value will never be NULL and overflow checking is unnecessary. Without the overhead of checking for nullness and overflow, SIMPLE_INTEGER provides significantly better performance than PLS_INTEGER when PLSQL_CODE_TYPE='NATIVE', because arithmetic operations on SIMPLE_INTEGER values are done directly in the hardware. When PLSQL_CODE_TYPE='INTERPRETED', the performance improvement is smaller.

Topics:

- Overflow Semantics
- Overloading Rules
- Integer Literals
- Cast Operations
- Compiler Warnings

**Overflow Semantics** The overflow semantics of SIMPLE_INTEGER differ significantly from those of PLS_INTEGER. An arithmetic operation that increases a PLS_INTEGER value to greater than 2,147,483,647 or decrease it to less than -2,147,483,648 causes error ORA-01426. In contrast, when this PL/SQL block is run from SQL*Plus, it runs without error:

```
DECLARE
  n SIMPLE_INTEGER := 2147483645;
BEGIN
  FOR j IN 1..4 LOOP
    n := n + 1;
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
  END LOOP;
  FOR j IN 1..4 LOOP
   n := n - 1;
   DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S9999999999'));
  END LOOP;
END;
/
```

Result:

```
+2147483646
+2147483647
-2147483648
-2147483647
-2147483648
+2147483647
+2147483646
+2147483645
```

#### Overloading Rules

- In overloaded subprograms, SIMPLE_INTEGER and PLS_INTEGER actual parameters can be substituted for each other.
- If all of their operands or arguments have the data type SIMPLE_INTEGER, the following produce SIMPLE_INTEGER results, using two's complement arithmetic and ignoring overflows:
  - Operators:
    - Addition (+)
    - Subtraction (-)
    - Multiplication (*)
  - Built-in functions:
    - MAX
    - MIN

    * ROUND
    * SIGN
    * TRUNC
- CASE expression

If some but not all operands or arguments have the data type SIMPLE_INTEGER, those of the data type SIMPLE_INTEGER are implicitly cast to PLS_INTEGER NOT NULL.

**Integer Literals** Integer literals in the SIMPLE_INTEGER range have the data type SIMPLE_INTEGER. This relieves you from explicitly casting each integer literal to SIMPLE_INTEGER in arithmetic expressions computed using two's complement arithmetic.

If and only if all operands and arguments have the data type SIMPLE_INTEGER, PL/SQL uses two's complement arithmetic and ignores overflows. Because overflows are ignored, values can wrap from positive to negative or from negative to positive; for example:

$2^{30} + 2^{30}$ = 0x40000000 + 0x40000000 = 0x80000000 = $-2^{31}$

$-2^{31} + -2^{31}$ = 0x80000000 + 0x80000000 = 0x00000000 = 0

To ensure backward compatibility, when all operands in an arithmetic expression are integer literals, PL/SQL treats the integer literals as if they were cast to PLS_INTEGER.

**Cast Operations** A cast operation that coerces a PLS_INTEGER value to the SIMPLE_INTEGER data type makes no conversion if the source value is not NULL. If the source value is NULL, a run-time exception is raised.

A cast operation that coerces a SIMPLE_INTEGER value to the PLS_INTEGER data type makes no conversion. This operation always succeeds (no exception is raised).

**Compiler Warnings** The compiler issues a warning in these cases:

- An operation mixes SIMPLE_INTEGER values with values of other numeric types.
- A SIMPLE_INTEGER value is passed as a parameter, a bind, or a define where a PLS_INTEGER is expected.

### BINARY_FLOAT and BINARY_DOUBLE Data Types

The BINARY_FLOAT and BINARY_DOUBLE data types represent single-precision and double-precision IEEE 754-format floating-point numbers, respectively.

A BINARY_FLOAT literal ends with f or F (for example, 2.07f). A BINARY_DOUBLE literal ends with d or D (for example, 3.000094D).

BINARY_FLOAT and BINARY_DOUBLE computations do not raise exceptions; therefore, you must check the values that they produce for conditions such as overflow and underflow, using the predefined constants listed and described in Table 3–5. For example:

```
SELECT COUNT(*)
  FROM employees
    WHERE salary < BINARY_FLOAT_INFINITY;
```

*Table 3–5 Predefined PL/SQL BINARY_FLOAT and BINARY_DOUBLE Constants*[1]

| Constant | Description |
|---|---|
| BINARY_FLOAT_NAN[1] | BINARY_FLOAT value for which the condition IS NAN (not a number) is true |
| BINARY_FLOAT_INFINITY[1] | Single-precision positive infinity |
| BINARY_FLOAT_MAX_NORMAL[1] | Maximum normal BINARY_FLOAT value |
| BINARY_FLOAT_MIN_NORMAL[1] | Minimum normal BINARY_FLOAT value |
| BINARY_FLOAT_MAX_SUBNORMAL[1] | Maximum subnormal BINARY_FLOAT value |
| BINARY_FLOAT_MIN_SUBNORMAL[1] | Minimum subnormal BINARY_FLOAT value |
| BINARY_DOUBLE_NAN[1] | BINARY_DOUBLE value for which the condition IS NAN (not a number) is true |
| BINARY_DOUBLE_INFINITY[1] | Double-precision positive infinity |
| BINARY_DOUBLE_MAX_NORMAL | Maximum normal BINARY_DOUBLE value |
| BINARY_DOUBLE_MIN_NORMAL | Minimum normal BINARY_DOUBLE value |
| BINARY_DOUBLE_MAX_SUBNORMAL | Maximum subnormal BINARY_DOUBLE value |
| BINARY_DOUBLE_MIN_SUBNORMAL | Minimum subnormal BINARY_DOUBLE value |

[1] Also predefined by SQL

In the IEEE-754 standard, **subnormal** ranges of values are intended to reduce problems caused by underflow to zero.

BINARY_FLOAT and BINARY_DOUBLE data types are primarily for high-speed scientific computation, as explained in "Computation-Intensive PL/SQL Programs" on page 12-33.

> **See Also:** "Subprograms that You Can Overload" on page 8-14, for information about writing libraries that accept different numeric types

SIMPLE_FLOAT and SIMPLE_DOUBLE are predefined subtypes of the BINARY_FLOAT and BINARY_DOUBLE data types, respectively. Each subtype has the same range as its base type and has a NOT NULL constraint.

You can use SIMPLE_FLOAT and SIMPLE_DOUBLE when the value will never be NULL. Without the overhead of checking for nullness, SIMPLE_FLOAT and SIMPLE_DOUBLE provide significantly better performance than BINARY_FLOAT and BINARY_DOUBLE when PLSQL_CODE_TYPE='NATIVE', because arithmetic operations on SIMPLE_FLOAT and SIMPLE_DOUBLE values are done directly in the hardware. When PLSQL_CODE_TYPE='INTERPRETED', the performance improvement is smaller.

### NUMBER Data Type

The NUMBER data type stores fixed-point or floating-point numbers with absolute values in the range 1E-130 up to (but not including) 1.0E126. A NUMBER variable can also represent 0.

Oracle recommends using only NUMBER literals and results of NUMBER computations that are in the specified range. Otherwise:

- Any value that is too small is rounded to zero.
- A literal value that is too large causes a compilation error.

- A computation result that is too large is undefined, causing unreliable results and possibly run-time errors.

A `NUMBER` value has both **precision** (its total number of digits) and **scale** (the number of digits to the right of the decimal point).

The syntax for specifying a fixed-point `NUMBER` is:

```
NUMBER(precision, scale)
```

For example:

```
NUMBER(8,2)
```

For an integer, the scale is zero. The syntax for specifying an integer `NUMBER` is:

```
NUMBER(precision)
```

For example:

```
NUMBER(2)
```

In a floating-point number, the decimal point can float to any position. The syntax for specifying a floating-point `NUMBER` is:

```
NUMBER
```

Both *`precision`* and *`scale`* must be integer literals, not constants or variables.

For *`precision`*, the maximum value is 38. The default value is 39 or 40, or the maximum for your system, whichever is least.

For *`scale`*, the minimum and maximum values are -84 and 127, respectively. The default value is zero.

Scale determines where rounding occurs. For example, a value whose scale is 2 is rounded to the nearest hundredth (3.454 becomes 3.45 and 3.456 becomes 3.46). A negative scale causes rounding to the left of the decimal point. For example, a value whose scale is -3 is rounded to the nearest thousand (34462 becomes 34000 and 34562 becomes 35000). A value whose scale is 0 is rounded to the nearest integer (3.4562 becomes 3 and 3.56 becomes 4).

For more information about the `NUMBER` data type, see *Oracle Database SQL Language Reference*.

Table 3–6 lists the predefined subtypes of the `NUMBER` data type and describes the data they store.

***Table 3–6 Predefined Subtypes of NUMBER Data Type***

| Data Type | Description |
|---|---|
| `DEC`, `DECIMAL`, or `NUMERIC` | Fixed-point `NUMBER` with maximum precision of 38 decimal digits |
| `DOUBLE PRECISION` or `FLOAT` | Floating-point `NUMBER` with maximum precision of 126 binary digits (approximately 38 decimal digits) |
| `INT`, `INTEGER`, or `SMALLINT` | Integer with maximum precision of 38 decimal digits |
| `REAL` | Floating-point `NUMBER` with maximum precision of 63 binary digits (approximately 18 decimal digits) |

## Predefined PL/SQL Character Data Types and Subtypes

Character data types let you store alphanumeric values that represent single characters or strings of characters, which you can manipulate. Table 3–7 describes the predefined PL/SQL character types and describes the data they store.

***Table 3–7 Predefined PL/SQL Character Data Types[1]***

| Data Type | Data Description |
|---|---|
| CHAR | Fixed-length character string with maximum size of 32,767 bytes |
| VARCHAR2 | Variable-length character string with maximum size of 32,767 bytes |
| RAW | Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL |
| NCHAR | Fixed-length national character string with maximum size of 32,767 bytes |
| NVARCHAR2 | Variable-length national character string with maximum size of 32,767 bytes |
| LONG[1] | Variable-length character string with maximum size of 32,760 bytes |
| LONG RAW[1] | Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL |
| ROWID[1] | Physical row identifier, the address of a row in an ordinary table |
| UROWID | Universal row identifier (physical, logical, or foreign row identifier) |

[1] Supported only for backward compatibility with existing applications

Topics:

- CHAR and VARCHAR2 Data Types
- RAW Data Type
- NCHAR and NVARCHAR2 Data Types
- LONG and LONG RAW Data Types
- ROWID and UROWID Data Types

### CHAR and VARCHAR2 Data Types

The CHAR and VARCHAR2 data types store fixed-length and variable-length character strings, respectively. All string literals have data type CHAR.

How CHAR and VARCHAR2 data is represented internally depends on the database character set. For information about the database character set, see "Database Character Set" on page 2-1.

The syntax for specifying a CHAR or VARCHAR2 data item is:

```
[ CHAR | VARCHAR2 ] [( maximum_size [ CHAR | BYTE ] )]
```

For example:

```
CHAR
VARCHAR2
CHAR(10 CHAR)
VARCHAR2(32 BYTE)
```

The *maximum_size* must be an integer literal in the range 1..32767, not a constant or variable. The default value is one.

The default size unit (CHAR or BYTE) is determined by the NLS_LENGTH_SEMANTICS initialization parameter. When a PL/SQL subprogram is compiled, the setting of this parameter is recorded, so that the same setting is used when the subprogram is recompiled after being invalidated. For more information about NLS_LENGTH_SEMANTICS, see *Oracle Database Reference*.

The maximum size of a CHAR or VARCHAR2 data item is 32,767 bytes, whether you specify *maximum_size* in characters or bytes. The maximum number of characters in a CHAR or VARCHAR2 data item depends on how the character set is encoded. For a single-byte character set, the maximum size of a CHAR or VARCHAR2 data item is 32,767 characters. For an *n*-byte character set, the maximum size of a CHAR or VARCHAR2 data item is 32,767/*n* characters, rounded down to the nearest integer. For a multiple-byte character set, specify *maximum_size* in characters to ensure that a CHAR(*n*) or VARCHAR2(*n*) variable can store *n* multiple-byte characters.

If the character value that you assign to a character variable is longer than the maximum size of the variable, PL/SQL does not truncate the value or strip trailing blanks; it stops the assignment and raises the predefined exception VALUE_ERROR.

For example, given the declaration:

```
acronym CHAR(4);
```

this assignment raises VALUE_ERROR:

```
acronym := 'SPCA ';  -- note trailing blank
```

If the character value that you insert into a database column is longer than the defined width of the column, PL/SQL does not truncate the value or strip trailing blanks; it stops the insertion and raises an exception.

To strip trailing blanks from a character value before assigning it to a variable or inserting it into a database column, use the built-in function RTRIM. For example, given the preceding declaration, this assignment does not raise an exception:

```
acronym := RTRIM('SPCA ');  -- note trailing blank
```

For the syntax of RTRIM, see *Oracle Database SQL Language Reference*.

### Differences Between CHAR and VARCHAR2 Data Types

CHAR and VARCHAR2 data types differ in:

- Predefined Subtypes of Character Data Types
- Memory Allocation for Character Variables
- Blank-Padding Shorter Character Values
- Comparing Character Values
- Maximum Sizes of Values Inserted into Character Database Columns

**Predefined Subtypes of Character Data Types** The CHAR data type has one predefined subtype, CHARACTER. The VARCHAR2 data type has two predefined subtypes, VARCHAR and STRING. Each of these subtypes has the same range of values as its base type, and can be used instead of its base type for compatibility with ANSI/ISO and IBM types.

---

**Note:** In a future PL/SQL release, to accommodate emerging SQL standards, VARCHAR might become a separate data type, no longer synonymous with VARCHAR2.

---

**Memory Allocation for Character Variables** For a CHAR variable, or for a VARCHAR2 variable whose maximum size is less than 2,000 bytes, PL/SQL allocates enough memory for the maximum size at compile time. For a VARCHAR2 whose maximum size is 2,000 bytes or more, PL/SQL allocates enough memory to store the actual value at run time. In this way, PL/SQL optimizes smaller VARCHAR2 variables for performance and larger ones for efficient memory use.

For example, if you assign the same 500-byte value to VARCHAR2(1999 BYTE) and VARCHAR2(2000 BYTE) variables, PL/SQL allocates 1999 bytes for the former variable at compile time and 500 bytes for the latter variable at run time.

**Blank-Padding Shorter Character Values** In each of these situations, whether or not PL/SQL blank-pads the character value depends on the data type of the receiver:

- The character value that you assign to a PL/SQL character variable is shorter than the maximum size of the variable.
- The character value that you insert into a character database column is shorter than the defined width of the column.
- The value that you retrieve from a character database column into a PL/SQL character variable is shorter than the maximum length of the variable.

If the data type of the receiver is CHAR, PL/SQL blank-pads the value to the maximum size. Information about trailing blanks in the original value is lost.

For example, the value assigned to last_name in this statement has six trailing blanks, not only one:

```
last_name CHAR(10) := 'CHEN ';  -- note trailing blank
```

If the data type of the receiver is VARCHAR2, PL/SQL neither blank-pads the value nor strips trailing blanks. Character values are assigned intact, and no information is lost.

**Comparing Character Values** You can use relational operators in Table 2–6 on page 2-34 to compare character values. One character value is greater than another if it follows it in the collating sequence used for the database character set. In this example, the IF condition is TRUE:

```
DECLARE
  last_name1 VARCHAR2(10) := 'COLES';
  last_name2 VARCHAR2(10) := 'COLEMAN';
BEGIN
  IF last_name1 > last_name2 THEN
    DBMS_OUTPUT.PUT_LINE
      (last_name1 || ' is greater than ' || last_name2);
  ELSE
    DBMS_OUTPUT.PUT_LINE
      (last_name2 || ' is greater than ' || last_name1);
  END IF;
END;
/
```

Result:

```
COLES is greater than COLEMAN
```

To be equal, two character values must have the same length.

If both values have data type CHAR, PL/SQL blank-pads the shorter value to the length of the longer value before comparing them. In Example 3–1, the IF condition is TRUE.

If either value has data type VARCHAR2, PL/SQL does not adjust their lengths before comparing them. In both Example 3–2 and Example 3–3, the IF condition is FALSE.

***Example 3–1 Comparing Two CHAR Values***

```
DECLARE
  last_name1 CHAR(5)  := 'BELLO';     -- no trailing blanks
  last_name2 CHAR(10) := 'BELLO   ';  -- trailing blanks
BEGIN
  IF last_name1 = last_name2 THEN
    DBMS_OUTPUT.PUT_LINE
      (last_name1 || ' equals ' || last_name2);
  ELSE
    DBMS_OUTPUT.PUT_LINE
      (last_name2 || ' does not equal ' || last_name1);
  END IF;
END;
/
```

Result:

```
BELLO equals BELLO
```

***Example 3–2 Comparing Two VARCHAR2 Values***

```
DECLARE
  last_name1 VARCHAR2(10) := 'DOW';     -- no trailing blanks
  last_name2 VARCHAR2(10) := 'DOW   ';  -- trailing blanks
BEGIN
  IF last_name1 = last_name2 THEN
    DBMS_OUTPUT.PUT_LINE
      (last_name1 || ' equals ' || last_name2 );
  ELSE
    DBMS_OUTPUT.PUT_LINE
      (last_name2 || ' does not equal ' || last_name1);
  END IF;
END;
/
```

Result:

```
DOW    does not equal DOW
```

***Example 3–3 Comparing CHAR Value and VARCHAR2 Value***

```
DECLARE
  last_name1 VARCHAR2(10) := 'STAUB';
  last_name2 CHAR(10)     := 'STAUB';  -- PL/SQL blank-pads value
BEGIN
  IF last_name1 = last_name2 THEN
    DBMS_OUTPUT.PUT_LINE
      (last_name1 || ' equals ' || last_name2);
  ELSE
    DBMS_OUTPUT.PUT_LINE
      (last_name2 || ' does not equal ' || last_name1 );
```

```
  END IF;
END;
/
```

Result:

```
STAUB      does not equal STAUB
```

**Maximum Sizes of Values Inserted into Character Database Columns** The largest CHAR value that you can insert into a CHAR database column is 2,000 bytes.

The largest VARCHAR2 value that you can insert into a VARCHAR2 database column is 4,000 bytes.

You can insert any CHAR or VARCHAR2 value into a LONG database column, because the maximum width of a LONG column is 2,147,483,648 bytes (2 GB). However, you cannot retrieve a value longer than 32,767 bytes from a LONG column into a CHAR or VARCHAR2 variable. (The LONG data type is supported only for backward compatibility with existing applications. For more information, see "LONG and LONG RAW Data Types" on page 3-14.)

### RAW Data Type

The RAW data type stores binary or byte strings, such as sequences of graphics characters or digitized pictures. Raw data is like VARCHAR2 data, except that PL/SQL does not interpret raw data. Oracle Net does no character set conversions when you transmit raw data from one system to another.

The syntax for specifying a RAW data item is:

```
RAW (maximum_size)
```

For example:

```
RAW(256)
```

The *maximum_size*, in bytes, must be an integer literal in the range 1..32767, not a constant or variable. The default value is one.

The largest RAW value that you can insert into a RAW database column is 2,000 bytes.

You can insert any RAW value into a LONG RAW database column, because the maximum width of a LONG RAW column is 2,147,483,648 bytes (2 GB). However, you cannot retrieve a value longer than 32,767 bytes from a LONG RAW column into a RAW variable. (The LONG RAW data type is supported only for backward compatibility with existing applications. For more information, see "LONG and LONG RAW Data Types" on page 3-14.)

### NCHAR and NVARCHAR2 Data Types

The NCHAR and NVARCHAR2 data types store fixed-length and variable-length national character strings, respectively.

How NCHAR and NVARCHAR2 data is represented internally depends on the national character set. For information about the national character set, see "National Character Set" on page 2-3.

Topics:

- NCHAR Data Type
- NVARCHAR2 Data Type

**NCHAR Data Type** The NCHAR data type stores fixed-length national character strings. Because this type can always accommodate multiple-byte characters, you can use it to store any Unicode character data.

The syntax for specifying an NCHAR data item is:

```
NCHAR [(maximum_size)]
```

For example:

```
NCHAR
NCHAR(100)
```

The *maximum_size* must be an integer literal, not a constant or variable. It represents the maximum number of characters, not the maximum number of bytes, which is 32,767. The largest *maximum_size* you can specify is 32767/2 with AL16UTF16 encoding and 32767/3 with UTF8 encoding. The default value is one.

The largest NCHAR value that you can insert into an NCHAR database column is 2,000 bytes.

If the NCHAR value is shorter than the defined width of the NCHAR column, PL/SQL blank-pads the value to the defined width.

You can interchange CHAR and NCHAR values in statements and expressions. It is always safe to convert a CHAR value to an NCHAR value, but converting an NCHAR value to a CHAR value might cause data loss if the character set for the CHAR value cannot represent all the characters in the NCHAR value. Such data loss usually results in characters that look like question marks (?).

**NVARCHAR2 Data Type** The NVARCHAR2 data type stores variable-length national character strings. Because this type can always accommodate multiple-byte characters, you can use it to store any Unicode character data.

The syntax for specifying an NVARCHAR2 data item is:

```
NVARCHAR2 (maximum_size)
```

For example:

```
NVARCHAR2(300)
```

The *maximum_size* must be an integer literal, not a constant or variable. It represents the maximum number of characters, not the maximum number of bytes, which is 32,767. The largest *maximum_size* you can specify is 32767/2 with AL16UTF16 encoding and 32767/3 with UTF8 encoding. The default value is one.

The largest NVARCHAR2 value that you can insert into an NVARCHAR2 database column is 4,000 bytes.

You can interchange VARCHAR2 and NVARCHAR2 values in statements and expressions. It is always safe to convert a VARCHAR2 value to an NVARCHAR2 value, but converting an NVARCHAR2 value to a VARCHAR2 value might cause data loss if the character set for the VARCHAR2 value cannot represent all the characters in the NVARCHAR2 value. Such data loss usually results in characters that look like question marks (?).

### LONG and LONG RAW Data Types

---

**Note:** The LONG and LONG RAW data types are supported only for backward compatibility with existing applications. For new applications, use CLOB or NCLOB instead of LONG, and BLOB or BFILE instead of LONG RAW. Oracle recommends that you also replace existing LONG and LONG RAW data types with LOB data types. See "Predefined PL/SQL Large Object (LOB) Data Types" on page 3-21.

---

The LONG data type stores variable-length character strings. The LONG data type is like the VARCHAR2 data type, except that the maximum size of a LONG value is 32,760 bytes (as opposed to 32,767 bytes).

The LONG RAW data type stores binary or byte strings. LONG RAW data is like LONG data, except that LONG RAW data is not interpreted by PL/SQL. The maximum size of a LONG RAW value is 32,760 bytes.

Because the maximum width of a LONG or LONG RAW database column is 2,147,483,648 bytes (2 GB), you can insert any LONG value into a LONG column and any LONG RAW value into a LONG RAW column. However, you cannot retrieve a value longer than 32,760 bytes from a LONG column into a LONG variable, or from a LONG RAW column into a LONG RAW variable.

LONG database columns can store text, arrays of characters, and even short documents.

**See Also:** *Oracle Database SQL Language Reference* for information about referencing LONG columns in SQL statements

### ROWID and UROWID Data Types

Internally, every database table has a ROWID pseudocolumn, which returns binary values called rowids. Each **rowid** represents the storage address of a row. A **physical rowid** identifies a row in an ordinary table. A **logical rowid** identifies a row in an index-organized table. The ROWID data type can store only physical rowids, while the UROWID (**universal rowid**) data type can store physical, logical, or foreign (not database) rowids.

---

**Note:** The ROWID data type is supported only for backward compatibility with existing applications. For new applications, use the UROWID data type.

---

Physical rowids are useful for working around the inability to fetch from a FOR UPDATE cursor after a COMMIT or ROLLBACK statement, as in Example 6–47 on page 6-46.

When you retrieve a rowid into a ROWID variable, you can use the built-in function ROWIDTOCHAR, which converts the binary value into an 18-byte character string. Conversely, the function CHARTOROWID converts a ROWID character string into a rowid. If the conversion fails because the character string does not represent a valid rowid, PL/SQL raises the predefined exception SYS_INVALID_ROWID. This also applies to implicit conversions.

To convert between UROWID variables and character strings, use regular assignment statements without any function call. The values are implicitly converted between UROWID and character types.

**See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the package DBMS_ROWID, whose subprograms enable you to manipulate rowids

## Predefined PL/SQL BOOLEAN Data Type

The BOOLEAN data type stores logical values, which you can use in logical operations. The logical values are the Boolean values TRUE and FALSE and the value NULL. NULL represents an unknown value.

The syntax for specifying an BOOLEAN data item is:

```
BOOLEAN
```

Because SQL has no data type equivalent to BOOLEAN, you cannot:

- Assign a BOOLEAN expression to a database table column
- Select or fetch the value of a database table column into a BOOLEAN variable
- Use a BOOLEAN expression in a SQL statement, built-in SQL function, or PL/SQL function invoked from a SQL statement

To represent BOOLEAN values in output, use IF or CASE statements to translate BOOLEAN values into another type (for example, 0 or 1, 'Y' or 'N', 'true' or 'false').

## Predefined PL/SQL Datetime and Interval Data Types

Datetime data types let you store dates and times. Interval data types let you store periods of time. A datetime or interval value has fields that determine its value—see Table 3–8.

All datetime and interval data types except TIMESTAMP WITH LOCAL TIMEZONE are part of the SQL standard. For information about datetime and interval format models, literals, time-zone names, and SQL functions, see *Oracle Database SQL Language Reference*.

***Table 3–8 Fields of Datetime and Interval Data Types***

| Field Name | Valid Datetime Values | Valid Interval Values |
|---|---|---|
| YEAR | -4712 to 9999 (excluding year 0) | Any nonzero integer |
| MONTH | 01 to 12 | 0 to 11 |
| DAY | 01 to 31 (limited by the values of MONTH and YEAR, according to the rules of the calendar for the locale) | Any nonzero integer |
| HOUR | 00 to 23 | 0 to 23 |
| MINUTE | 00 to 59 | 0 to 59 |
| SECOND | 00 to 59.9(n), where 9(n) is the precision of time fractional seconds | 0 to 59.9(n), where 9(n) is the precision of interval fractional seconds |
| TIMEZONE_HOUR | -12 to 14 (range accommodates daylight savings time changes) | Not applicable |
| TIMEZONE_MINUTE | 00 to 59 | Not applicable |
| TIMEZONE_REGION | Found in the dynamic performance view V$TIMEZONE_NAMES | Not applicable |

*Table 3–8 (Cont.) Fields of Datetime and Interval Data Types*

| Field Name | Valid Datetime Values | Valid Interval Values |
|---|---|---|
| TIMEZONE_ABBR | Found in the dynamic performance view V$TIMEZONE_NAMES | Not applicable |

Topics:

- DATE Data Type
- TIMESTAMP Data Type
- TIMESTAMP WITH TIME ZONE Data Type
- TIMESTAMP WITH LOCAL TIME ZONE Data Type
- INTERVAL YEAR TO MONTH Data Type
- INTERVAL DAY TO SECOND Data Type
- Datetime and Interval Arithmetic
- Avoiding Truncation Problems with Date and Time Subtypes

### DATE Data Type

You use the DATE data type to store fixed-length datetimes, which include the time of day in seconds since midnight. The date portion defaults to the first day of the current month; the time portion defaults to midnight. The date function SYSDATE returns the current date and time.

To compare dates for equality, regardless of the time portion of each date, use the function result TRUNC(*date_variable*) in comparisons, GROUP BY operations, and so on.

To find just the time portion of a DATE variable, subtract the date portion: date_variable - TRUNC(*date_variable*).

Valid dates range from January 1, 4712 BC to December 31, 9999 AD. A Julian date is the number of days since January 1, 4712 BC. Julian dates allow continuous dating from a common reference. You can use the date format model 'J' with the date functions TO_DATE and TO_CHAR to convert between DATE values and their Julian equivalents.

In date expressions, PL/SQL automatically converts character values in the default date format to DATE values. The default date format is set by the Oracle initialization parameter NLS_DATE_FORMAT. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year.

You can use the addition and subtraction operators to increment or decrement date values. For example:

```
same_time_tomorrow := SYSDATE + 1;
hire_date := '10-MAY-95';
hire_date := hire_date + 1;  -- makes hire_date '11-MAY-95'
hire_date := hire_date - 5;  -- makes hire_date '06-MAY-95'
```

### TIMESTAMP Data Type

The TIMESTAMP data type, which extends the data type DATE, stores the year, month, day, hour, minute, and second. The syntax is:

```
TIMESTAMP[(precision)
```

where the optional parameter *precision* specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..9. The default is 6.

The default timestamp format is set by the Oracle initialization parameter NLS_TIMESTAMP_FORMAT.

Example 3–4 declares a variable of type TIMESTAMP and assigns a literal value to it. The fractional part of the seconds field is 0.275.

***Example 3–4 Assigning a Literal Value to a TIMESTAMP Variable***

```
DECLARE
  checkout TIMESTAMP(3);
BEGIN
  checkout := '22-JUN-2004 07:48:53.275';
  DBMS_OUTPUT.PUT_LINE( TO_CHAR(checkout));
END;
/
```

Result:

```
22-JUN-04 07.48.53.275 AM
```

In Example 3–5, the SCN_TO_TIMESTAMP and TIMESTAMP_TO_SCN functions are used to manipulate TIMESTAMPs.

***Example 3–5 SCN_TO_TIMESTAMP and TIMESTAMP_TO_SCN Functions***

```
DECLARE
  right_now  TIMESTAMP;
  yesterday  TIMESTAMP;
  sometime   TIMESTAMP;
  scn1       INTEGER;
  scn2       INTEGER;
  scn3       INTEGER;
BEGIN
  right_now := SYSTIMESTAMP;
  scn1 := TIMESTAMP_TO_SCN(right_now);
  DBMS_OUTPUT.PUT_LINE('Current SCN is ' || scn1);

  yesterday := right_now - 1;
  scn2 := TIMESTAMP_TO_SCN(yesterday);
  DBMS_OUTPUT.PUT_LINE('SCN from yesterday is ' || scn2);

  -- Find arbitrary SCN between yesterday and today

  scn3 := (scn1 + scn2) / 2;
  sometime := SCN_TO_TIMESTAMP(scn3);
  DBMS_OUTPUT.PUT_LINE
    ('SCN ' || scn3 || ' was in effect at ' || TO_CHAR(sometime));
END;
/
```

Result:

```
Current SCN is 10216101
SCN from yesterday is 10105627
SCN 10160864 was in effect at 17-MAR-09 12.16.02.000000 AM
```

### TIMESTAMP WITH TIME ZONE Data Type

The TIMESTAMP WITH TIME ZONE data type, which extends the data type TIMESTAMP, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC,) formerly Greenwich Mean Time (GMT). The syntax is:

```
TIMESTAMP[(precision)] WITH TIME ZONE
```

where the optional parameter *precision* specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..9. The default is 6.

The default timestamp with time zone format is set by the Oracle initialization parameter NLS_TIMESTAMP_TZ_FORMAT.

Example 3–6 declares a variable of type TIMESTAMP WITH TIME ZONE and assign a literal value to it. The time-zone displacement is +02:00.

***Example 3–6 Assigning a Literal to a TIMESTAMP WITH TIME ZONE Variable***

```
DECLARE
  logoff TIMESTAMP(3) WITH TIME ZONE;
BEGIN
  logoff := '10-OCT-2004 09:42:37.114 AM +02:00';
  DBMS_OUTPUT.PUT_LINE (TO_CHAR(logoff));
END;
/
```

Result:

```
10-OCT-04 09.42.37.114 AM +02:00
```

You can also specify the time zone by using a symbolic name. The specification can include a long form such as 'US/Pacific', an abbreviation such as 'PDT', or a combination. For example, these literals all represent the same time:

```
TIMESTAMP '15-APR-2004 8:00:00 -8:00'
TIMESTAMP '15-APR-2004 8:00:00 US/Pacific'
TIMESTAMP '31-OCT-2004 01:30:00 US/Pacific PDT'
```

Of the preceding forms, the third form is most reliable, because it specifies the rules to follow at the point when switching to daylight savings time.

You can find the available names for time zones in the TIMEZONE_REGION and TIMEZONE_ABBR columns of the static data dictionary view V$TIMEZONE_NAMES.

Two TIMESTAMP WITH TIME ZONE values are considered identical if they represent the same instant in UTC, regardless of their time-zone displacements. For example, these two values are considered identical because, in UTC, 8:00 AM Pacific Standard Time equals 11:00 AM Eastern Standard Time:

```
'29-AUG-2004 08:00:00 -8:00'
'29-AUG-2004 11:00:00 -5:00'
```

### TIMESTAMP WITH LOCAL TIME ZONE Data Type

The TIMESTAMP WITH LOCAL TIME ZONE data type, which extends the data type TIMESTAMP, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC)—formerly Greenwich Mean Time. You can also use named time zones, as with TIMESTAMP WITH TIME ZONE.

The syntax is:

```
TIMESTAMP[(precision)] WITH LOCAL TIME ZONE
```

where the optional parameter *precision* specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..9. The default is 6.

This data type differs from TIMESTAMP WITH TIME ZONE in that when you insert a value into a database column, the value is normalized to the database time zone, and the time-zone displacement is not stored in the column. When you retrieve the value, Oracle returns it in your local session time zone.

Both Example 3–7 and Example 3–8 declare a variable of type TIMESTAMP WITH LOCAL TIME ZONE and assign it a value. The value in Example 3–7 is an appropriate local time, but the value in Example 3–8 includes a time zone displacement, which causes an error.

***Example 3–7 Correct Assignment to TIMESTAMP WITH LOCAL TIME ZONE***

```
DECLARE
  logoff  TIMESTAMP(3) WITH LOCAL TIME ZONE;
BEGIN
  logoff := '10-OCT-2004 09:42:37.114 AM ';
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(logoff));
END;
/
```

Result:

```
10-OCT-04 09.42.37.114 AM
```

***Example 3–8 Incorrect Assignment to TIMESTAMP WITH LOCAL TIME ZONE***

```
DECLARE
  logoff  TIMESTAMP(3) WITH LOCAL TIME ZONE;
BEGIN
  logoff := '10-OCT-2004 09:42:37.114 AM +02:00';
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-01830: date format picture ends before converting entire input string
ORA-06512: at line 4
```

### INTERVAL YEAR TO MONTH Data Type

Use the INTERVAL YEAR TO MONTH data type to store and manipulate intervals of years and months. The syntax is:

```
INTERVAL YEAR[(precision)] TO MONTH
```

where *precision* specifies the number of digits in the years field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..4. The default is 2.

Example 3–9 declares a variable of type INTERVAL YEAR TO MONTH and assigns a value of 101 years and 3 months to it, in three different ways.

***Example 3–9 Assigning Literals to an INTERVAL YEAR TO MONTH Variable***

```
DECLARE
  lifetime  INTERVAL YEAR(3) TO MONTH;
BEGIN
  lifetime := INTERVAL '101-3' YEAR TO MONTH;  -- Interval literal

  lifetime := '101-3';  -- Implicit conversion from character type

  lifetime := INTERVAL '101' YEAR;  -- Specify only years
  lifetime := INTERVAL '3' MONTH;   -- Specify only months
END;
/
```

### INTERVAL DAY TO SECOND Data Type

You use the INTERVAL DAY TO SECOND data type to store and manipulate intervals of days, hours, minutes, and seconds. The syntax is:

```
INTERVAL DAY[(leading_precision)
  TO SECOND (fractional_seconds_precision)
```

where *leading_precision* and *fractional_seconds_precision* specify the number of digits in the days field and seconds field, respectively. In both cases, you cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..9. The defaults are 2 and 6, respectively.

Example 3–10 declares a variable of type INTERVAL DAY TO SECOND and assigns a value to it.

***Example 3–10 Assigning Literals to an INTERVAL DAY TO SECOND Variable***

```
DECLARE
  lag_time  INTERVAL DAY(3) TO SECOND(3);
BEGIN
  lag_time := '7 09:24:30';

  IF lag_time > INTERVAL '6' DAY THEN
    DBMS_OUTPUT.PUT_LINE ('Greater than 6 days');
  ELSE
    DBMS_OUTPUT.PUT_LINE ('Less than 6 days');
  END IF;
END;
/
```

Result:

```
Greater than 6 days
```

### Datetime and Interval Arithmetic

PL/SQL lets you construct datetime and interval expressions. This table shows the operators that you can use in such expressions:

| Operand 1 | Operator | Operand 2 | Result Type |
|---|---|---|---|
| datetime | + | interval | datetime |
| datetime | – | interval | datetime |
| interval | + | datetime | datetime |
| datetime | – | datetime | interval |

| Operand 1 | Operator | Operand 2 | Result Type |
|---|---|---|---|
| interval | + | interval | interval |
| interval | – | interval | interval |
| interval | * | numeric | interval |
| numeric | * | interval | interval |
| interval | / | numeric | interval |

**See Also:** *Oracle Database SQL Language Reference* for information about using SQL functions to perform arithmetic operations on datetime values

### Avoiding Truncation Problems with Date and Time Subtypes

The default precisions for some date and time types are less than the maximum precision. For example, the default for DAY TO SECOND is DAY(2) TO SECOND(6), while the highest precision is DAY(9) TO SECOND(9). To avoid truncation when assigning variables and passing subprogram parameters of these types, you can declare variables and subprogram parameters of these subtypes, which use the maximum values for precision:

```
TIMESTAMP_UNCONSTRAINED
TIMESTAMP_TZ_UNCONSTRAINED
TIMESTAMP_LTZ_UNCONSTRAINED
YMINTERVAL_UNCONSTRAINED
DSINTERVAL_UNCONSTRAINED
```

## Predefined PL/SQL Large Object (LOB) Data Types

Large object (LOB) data types reference large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms. LOB data types allow efficient, random, piecewise access to this data. Predefined PL/SQL LOB data types are listed and described in Table 3–9.

***Table 3–9 Predefined PL/SQL Large Object (LOB) Data Types***

| Data Type | Description | Size |
|---|---|---|
| BFILE | Used to store large binary objects in operating system files outside the database. | System-dependent. Cannot exceed 4 gigabytes (GB). |
| BLOB | Used to store large binary objects in the database. | 8 to 128 terabytes (TB) |
| CLOB | Used to store large blocks of character data in the database. | 8 to 128 TB |
| NCLOB | Used to store large blocks of NCHAR data in the database. | 8 to 128 TB |

### LOB Locators

To reference a large object that is stored in an external file, a LOB data type uses a **LOB locator**, which is stored in an external file, either inside the row (**inline**) or outside the row (**out-of-line**). In the external file, LOB locators are in columns of the types BFILE, BLOB, CLOB, and NCLOB.

PL/SQL operates on large objects through their LOB locators. For example, when you select a BLOB column value, PL/SQL returns only its locator. If PL/SQL returned the locator during a transaction, the locator includes a transaction ID, so you cannot use

that locator to update that large object in another transaction. Likewise, you cannot save a locator during one session and then use it in another session.

#### Differences Between LOB Data Types and LONG and LONG RAW Data Types

LOB data types differ from LONG and LONG RAW data types in these ways:

| Difference | LOB Data Types | LONG and LONG RAW Data Types |
|---|---|---|
| Support | Functionality enhanced in every release. | Functionality static. Supported only for backward compatibility with existing applications. |
| Maximum size | 8 to 128 TB | 2 GB |
| Access | Random | Sequential |
| Can be ADT attribute | BFILE, BLOB, CLOB: Yes. NCLOB: No | No |

**See Also:**

- "LONG and LONG RAW Data Types" on page 3-14
- *Oracle Database SecureFiles and Large Objects Developer's Guide*for more information about LOBs

Topics:

- BFILE Data Type
- BLOB Data Type
- CLOB Data Type
- NCLOB Data Type

## BFILE Data Type

You use the BFILE data type to store large binary objects in operating system files outside the database. Every BFILE variable stores a file locator, which points to a large binary file on the server. The locator includes a directory alias, which specifies a full path name. Logical path names are not supported.

BFILEs are read-only, so you cannot modify them. Your DBA makes sure that a given BFILE exists and that Oracle has read permissions on it. The underlying operating system maintains file integrity.

BFILEs do not participate in transactions, are not recoverable, and cannot be replicated. The maximum number of open BFILEs is set by the Oracle initialization parameter SESSION_MAX_OPEN_FILES, which is system dependent.

## BLOB Data Type

You use the BLOB data type to store large binary objects in the database, inline or out-of-line. Every BLOB variable stores a locator, which points to a large binary object.

BLOBs participate fully in transactions, are recoverable, and can be replicated. Changes made by package DBMS_LOB can be committed or rolled back. BLOB locators can span transactions (for reads only), but they cannot span sessions.

## CLOB Data Type

You use the CLOB data type to store large blocks of character data in the database, inline or out-of-line. Both fixed-width and variable-width character sets are supported. Every CLOB variable stores a locator, which points to a large block of character data.

CLOBs participate fully in transactions, are recoverable, and can be replicated. Changes made by package DBMS_LOB can be committed or rolled back. CLOB locators can span transactions (for reads only), but they cannot span sessions.

## NCLOB Data Type

You use the NCLOB data type to store large blocks of NCHAR data in the database, inline or out-of-line. Both fixed-width and variable-width character sets are supported. Every NCLOB variable stores a locator, which points to a large block of NCHAR data.

NCLOBs participate fully in transactions, are recoverable, and can be replicated. Changes made by package DBMS_LOB can be committed or rolled back. NCLOB locators can span transactions (for reads only), but they cannot span sessions.

# User-Defined PL/SQL Subtypes

You can define your own subtypes in the declarative part of any PL/SQL block, subprogram, or package using this syntax:

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

where *subtype_name* is an identifier and *base_type* is a predefined or previously defined data type. Specify a *constraint* only if *base_type* can specify precision and scale or a maximum size.

***Example 3–11 Defining Subtypes***

```
DECLARE
  SUBTYPE BirthDate IS DATE NOT NULL;          -- Based on DATE type
  SUBTYPE Counter IS NATURAL;                  -- Based on NATURAL subtype

  TYPE NameList IS TABLE OF VARCHAR2(10);
  SUBTYPE DutyRoster IS NameList;              -- Based on TABLE type

  TYPE TimeRec IS RECORD (minutes INTEGER, hours INTEGER);
  SUBTYPE FinishTime IS TimeRec;                 -- Based on RECORD type
  SUBTYPE ID_Num IS employees.employee_id%TYPE; -- Based on column type
BEGIN
  NULL;
END;
/
```

After defining a subtype, you can declare items of that type. The subtype name indicates the intended use of the variable. You can constrain a user-defined subtype when declaring variables of that type, as in Example 3–12.

***Example 3–12 Declaring Variables of User-Defined Subtype***

```
DECLARE
  SUBTYPE Counter IS NATURAL;
  rows Counter;

  SUBTYPE Accumulator IS NUMBER;
  total Accumulator(7,2);
```

```
BEGIN
  NULL;
END;
/
```

Subtypes can increase reliability by detecting out-of-range values. Example 3–13 restricts the subtype `pinteger` to storing integers in the range -9..9. When the program tries to store a number outside that range in a `pinteger` variable, PL/SQL raises an exception.

#### *Example 3–13 Ranges with Subtypes*

```
DECLARE
  v_sqlerrm VARCHAR2(64);

  SUBTYPE pinteger IS PLS_INTEGER RANGE -9..9;
  y_axis pinteger;

  PROCEDURE p (x IN pinteger) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE (x);
  END p;

BEGIN
  y_axis := 9;
  p(10);

EXCEPTION
  WHEN OTHERS THEN
    v_sqlerrm := SUBSTR(SQLERRM, 1, 64);
    DBMS_OUTPUT.PUT_LINE('Error: ' || v_sqlerrm);
END;
/
```

Result:

```
Error: ORA-06502: PL/SQL: numeric or value error
```

An unconstrained subtype is interchangeable with its base type. Example 3–14 assigns the value of `amount` to `total` without conversion.

#### *Example 3–14 Type Compatibility with the NUMBER Data Type*

```
DECLARE
  SUBTYPE Accumulator IS NUMBER;
  amount  NUMBER(7,2);
  total   Accumulator;
BEGIN
  amount := 10000.50;
  total  := amount;
END;
/
```

Different subtypes are interchangeable if they have the same base type:

```
DECLARE
  SUBTYPE b1 IS BOOLEAN;
  SUBTYPE b2 IS BOOLEAN;
  finished  b1;
  debugging b2;
BEGIN
```

```
  finished  := FALSE;
  debugging := finished;
END;
/
```

Different subtypes are also interchangeable if their base types are in the same data type family. For example, the value of `verb` can be assigned to `sentence`:

```
DECLARE
  SUBTYPE Word IS CHAR(15);
  SUBTYPE Text IS VARCHAR2(1500);
  verb     Word;
  sentence Text(150);
BEGIN
  verb := 'program';
  sentence := verb;
END;
/
```

Example 3–15 assigns an initial value to a constrained subtype variable.

***Example 3–15 Assigning Initial Value to Subtype Variable***

```
DECLARE
  SUBTYPE v_word IS VARCHAR2(10) NOT NULL;
  verb  v_word  := 'verb';
  noun  v_word  := 'noun';
BEGIN
  DBMS_OUTPUT.PUT_LINE (UPPER(verb));
  DBMS_OUTPUT.PUT_LINE (UPPER(noun));
END;
/
```

Result:

```
VERB
NOUN
```

In Example 3–16, the procedure enforces the `NOT NULL` constraint, but not the size constraint.

***Example 3–16 Subtype Constraints Inherited by Subprograms***

```
DECLARE
  SUBTYPE v_word IS VARCHAR2(10) NOT NULL;
  verb    v_word       := 'run';
  noun    VARCHAR2(10) := NULL;

  PROCEDURE word_to_upper (w IN v_word) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE (UPPER(w));
  END word_to_upper;
BEGIN
 12    word_to_upper('more than ten characters');
 13    word_to_upper(noun);
END;
/
```

Result:

```
MORE THAN TEN CHARACTERS
DECLARE
```

```
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 13
```

As Example 3–17 shows, subtypes do not inherit the column constraints NOT NULL or CHECK, but they do inherit column size constraints.

***Example 3–17 Column Constraints Inherited by Subtypes***

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp (
  empid NUMBER(6) NOT NULL PRIMARY KEY,
  deptid NUMBER(6) CONSTRAINT c_employees_temp_deptid
    CHECK (deptid BETWEEN 100 AND 200),
  deptname VARCHAR2(30)
);

DECLARE
  SUBTYPE v_empid_subtype    IS employees_temp.empid%TYPE;
  SUBTYPE v_deptid_subtype   IS employees_temp.deptid%TYPE;
  SUBTYPE v_deptname_subtype IS employees_temp.deptname%TYPE;
  SUBTYPE v_emprec_subtype   IS employees_temp%ROWTYPE;

  v_empid    v_empid_subtype;
  v_deptid   v_deptid_subtype;
  v_deptname v_deptname_subtype;
  v_emprec   v_emprec_subtype;
BEGIN
  v_empid  := NULL;         -- NULL constraint not inherited
  v_deptid := 50;           -- CHECK constraint not inherited

  v_emprec.empid  := NULL;  -- NULL constraint not inherited
  v_emprec.deptid := 50;    -- CHECK constraint not inherited

 DBMS_OUTPUT.PUT_LINE
   ('v_deptname: ' || v_deptname);  -- Initial value not inherited

 DBMS_OUTPUT.PUT_LINE
   ('v_emprec.deptname: ' || v_emprec.deptname);
   -- Initial value not inherited
  v_empid := 10000002;  -- NUMBER(6) constraint inherited
END;
/
```

Result:

```
v_deptname:
v_emprec.deptname:
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 24
```

# PL/SQL Data Type Conversion

Sometimes it is necessary to convert a value from one data type to another. For example, to use a DATE value in a report, you must convert it to a character string. PL/SQL supports both explicit and implicit data type conversion.

For best reliability and maintainability, use explicit conversion. Implicit conversion is context-sensitive and not always predictable, and its rules might change in later software releases. Implicit conversion can also be slower than explicit conversion.

Topics:

- Explicit Conversion
- Implicit Conversion

## Explicit Conversion

To explicitly convert values from one data type to another, you use built-in functions, which are described in *Oracle Database SQL Language Reference*. For example, to convert a CHAR value to a DATE or NUMBER value, you use the function TO_DATE or TO_NUMBER, respectively. Conversely, to convert a DATE or NUMBER value to a CHAR value, you use the function TO_CHAR.

Explicit conversion can prevent errors or unexpected results. For example:

- Using the concatenation operator (||) to concatenate a string and an arithmetic expression can produce an error, which you can prevent by using the TO_CHAR function to convert the arithmetic expression to a string before concatenation.
- Relying on language settings in the database for the format of a DATE value can produce unexpected results, which you can prevent by using the TO_CHAR function and specifying the desired format.

## Implicit Conversion

Sometimes PL/SQL can convert a value from one data type to another automatically. This is called implicit conversion, and the data types are called **compatible**. When two data types are compatible, you can use a value of one type where a value of the other type is expected. For example, you can pass a numeric literal to a subprogram that expects a string value, and the subprogram receives the string representation of the number.

In Example 3–18, the CHAR variables start_time and finish_time store string values representing the number of seconds past midnight. The difference between those values can be assigned to the NUMBER variable elapsed_time, because PL/SQL converts the CHAR values to NUMBER values automatically.

***Example 3–18 Implicit Conversion***

```
DECLARE
  start_time   CHAR(5);
  finish_time  CHAR(5);
  elapsed_time NUMBER(5);
BEGIN
  -- Get system time as seconds past midnight:
  SELECT TO_CHAR(SYSDATE,'SSSSS') INTO start_time FROM sys.DUAL;
  -- Processing done here
  -- Get system time again:
  SELECT TO_CHAR(SYSDATE,'SSSSS') INTO finish_time FROM sys.DUAL;
  -- Compute and report elapsed time in seconds:
  elapsed_time := finish_time - start_time;
  DBMS_OUTPUT.PUT_LINE ('Elapsed time: ' || TO_CHAR(elapsed_time));
END;
/
```

Result:

```
Elapsed time: 0
```

If you select a value from a column of one data type, and assign that value to a variable of another data type, PL/SQL converts the value to the data type of the variable. This happens, for example, when you select a DATE column value into a VARCHAR2 variable.

If you assign the value of a variable of one database type to a column of another database type, PL/SQL converts the value of the variable to the data type of the column.

If PL/SQL cannot determine which implicit conversion is needed, you get a compilation error. In such cases, you must use explicit conversion.

Table 3–10 shows which implicit conversions PL/SQL can do. However:

- Table 3–10 lists only data types that have different representations.

  Types that have the same representation, such as PLS_INTEGER and BINARY_INTEGER, CLOB and NCLOB, CHAR and NCHAR, and VARCHAR and NVARCHAR2, can be substituted for each other.

- It is your responsibility to ensure that specific values are convertible.

  For example, PL/SQL can convert the CHAR value '02-JUN-92' to a DATE value but cannot convert the CHAR value 'YESTERDAY' to a DATE value. Similarly, PL/SQL cannot convert a VARCHAR2 value containing alphabetic characters to a NUMBER value.

- Regarding date, time, and interval data types:

  – Conversion rules for the DATE data type also apply to the datetime data types. However, because of their different internal representations, these types cannot always be converted to each other. For details about implicit conversions between datetime data types, see *Oracle Database SQL Language Reference*.

  – To implicitly convert a DATE value to a CHAR or VARCHAR2 value, PL/SQL invokes the function TO_CHAR, which returns a character string in the default date format. To get other information, such as the time or Julian date, invoke TO_CHAR explicitly with a format mask.

  – When you insert a CHAR or VARCHAR2 value into a DATE column, PL/SQL implicitly converts the CHAR or VARCHAR2 value to a DATE value by invoking the function TO_DATE, which expects its parameter to be in the default date format. To insert dates in other formats, invoke TO_DATE explicitly with a format mask.

- Regarding LOB data types:

  – Converting between CLOB and NCLOB values can be expensive. To make clear that you intend this conversion, use the explicit conversion functions TO_CLOB and TO_NCLOB.

  – Implicit conversion between CLOB values and CHAR and VARCHAR2 values, and between BLOB values and RAW values, lets you use LOB data types in most SQL and PL/SQL statements and functions. However, to read, write, and do piecewise operations on LOB values, you must use DBMS_LOB package subprograms, which are described in *Oracle Database PL/SQL Packages and Types Reference*.

- Regarding RAW and LONG RAW data types:

- `LONG RAW` is supported only for backward compatibility with existing applications. For more information, see "LONG and LONG RAW Data Types" on page 3-14.
- When you select a `RAW` or `LONG RAW` column value into a `CHAR` or `VARCHAR2` variable, PL/SQL must convert the internal binary value to a character value. PL/SQL does this by returning each binary byte of `RAW` or `LONG RAW` data as a pair of characters. Each character represents the hexadecimal equivalent of a **nibble** (half a byte). For example, PL/SQL returns the binary byte 11111111 as the pair of characters `'FF'`. The function `RAWTOHEX` does the same conversion.
- Conversion is also necessary when you insert a `CHAR` or `VARCHAR2` value into a `RAW` or `LONG RAW` column. Each pair of characters in the variable must represent the hexadecimal equivalent of a binary byte; otherwise, PL/SQL raises an exception.
- When a `LONG` value appears in a SQL statement, PL/SQL binds the `LONG` value as a `VARCHAR2` value. However, if the length of the bound `VARCHAR2` value exceeds the maximum width of a `VARCHAR2` column (4,000 bytes), Oracle converts the bind type to `LONG` automatically, and then issues an error message because you cannot pass `LONG` values to a SQL function.

***Table 3–10 Possible Implicit PL/SQL Data Type Conversions***

| **From:** | **To:** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | BLOB | CHAR | CLOB | DATE | LONG | NUMBER | PLS_INTEGER | RAW | UROWID | VARCHAR2 |
| BLOB | | | | | | | | Yes | | |
| CHAR | | | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes |
| CLOB | | Yes | | | | | | | | Yes |
| DATE | | Yes | | | Yes | | | | | Yes |
| LONG | | Yes | | | | | | Yes | | Yes |
| NUMBER | | Yes | | | Yes | | Yes | | | Yes |
| PLS_INTEGER | | Yes | | | Yes | Yes | | | | Yes |
| RAW | Yes | Yes | | | Yes | | | | | Yes |
| UROWID | | Yes | | | | | | | | Yes |
| VARCHAR2 | | Yes | Yes | Yes | Yes | Yes | Yes | Yes | Yes | |

4

# PL/SQL Control Statements

This chapter explains how to use the PL/SQL statements that control the flow of execution in a PL/SQL program.

Topics:


- Overview of PL/SQL Control Statements
- Conditional Selection Statements
- LOOP Statements
- Sequential Control Statements


## Overview of PL/SQL Control Statements

PL/SQL has three categories of control statements:

- **Conditional selection statements**, which run different statements for different data values.

  The conditional selection statements are `IF` and and `CASE`.

- **Loop statements**, which run the same statements with a series of different data values.

  The loop statements are the `FOR LOOP`, `WHILE LOOP` and basic `LOOP`.

  The `EXIT` statement transfers control to the end of a loop. The `CONTINUE` statement exits the current iteration of a loop and transfers control to the next iteration. Both `EXIT` and `CONTINUE` have an optional `WHEN` clause, where you can specify a condition.

- **Sequential control statements**, which are not crucial to PL/SQL programming.

  The sequential control statements are `GOTO`, which goes to a specified statement, and `NULL`, which does nothing.

## Conditional Selection Statements

The **conditional selection statements**, `IF` and `CASE`, run different statements for different data values.

The `IF` statement either runs or skips a sequence of one or more statements, depending on a condition. The `IF` statement has these forms:

- `IF THEN`
- `IF THEN ELSE`

- IF THEN ELSIF

The CASE statement chooses from a sequence of conditions, and runs the corresponding statement. The CASE statement has these forms:

- Simple, which evaluates a single expression and compares it to several potential values.
- Searched, which evaluates multiple conditions and chooses the first one that is true.

The CASE statement is appropriate when a different action is to be taken for each alternative.

Topics:

- IF THEN Statement
- IF THEN ELSE Statement
- IF THEN ELSIF Statement
- Simple CASE Statement
- Searched CASE Statement

## IF THEN Statement

The IF THEN statement has this structure:

```
IF condition THEN
  statements
END IF;
```

If the *condition* is true, the *statements* run; otherwise, the IF statement does nothing. (For complete syntax, see "IF Statement" on page 13-82.)

In Example 4–1, the statements between THEN and END IF run if and only if the value of sales is greater than quota+200.

***Example 4–1 IF THEN Statement***

```
DECLARE
  PROCEDURE p (
    sales  NUMBER,
    quota  NUMBER,
    emp_id NUMBER
  )
  IS
    bonus    NUMBER := 0;
    updated  VARCHAR2(3) := 'No';
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;

      UPDATE employees
      SET salary = salary + bonus
      WHERE employee_id = emp_id;

      updated := 'Yes';
    END IF;

    DBMS_OUTPUT.PUT_LINE (
      'Table updated?  ' || updated || ', ' ||
```

```
      'bonus = ' || bonus || '.'
    );
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
END;
/
```

Result:

```
Table updated?  No, bonus = 0.
Table updated?  Yes, bonus = 125.
```

> **Tip:** Avoid clumsy IF statements such as:
>
> ```
> IF new_balance < minimum_balance THEN
>   overdrawn := TRUE;
> ELSE
>   overdrawn := FALSE;
> END IF;
> ```
>
> Instead, assign the value of the BOOLEAN expression directly to a BOOLEAN variable:
>
> ```
> overdrawn := new_balance < minimum_balance;
> ```
>
> A BOOLEAN variable is either TRUE, FALSE, or NULL. Do not write:
>
> ```
> IF overdrawn = TRUE THEN
>   RAISE insufficient_funds;
> END IF;
> ```
>
> Instead, write:
>
> ```
> IF overdrawn THEN
>   RAISE insufficient_funds;
> END IF;
> ```

## IF THEN ELSE Statement

The IF THEN ELSE statement has this structure:

```
IF condition THEN
  statements
ELSE
  else_statements
END IF;
```

If the value of *condition* is true, the *statements* run; otherwise, the *else_statements* run. (For complete syntax, see "IF Statement" on page 13-82.)

In Example 4–2, the statement between THEN and ELSE runs if and only if the value of sales is greater than quota+200; otherwise, the statement between ELSE and END IF runs.

***Example 4–2 IF THEN ELSE Statement***

```
DECLARE
  PROCEDURE p (
    sales  NUMBER,
    quota  NUMBER,
```

```
    emp_id NUMBER
  )
  IS
    bonus  NUMBER := 0;
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;
    ELSE
      bonus := 50;
    END IF;

    DBMS_OUTPUT.PUT_LINE('bonus = ' || bonus);

    UPDATE employees
    SET salary = salary + bonus
    WHERE employee_id = emp_id;
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
END;
/
```

Result:

```
bonus = 50
bonus = 125
```

`IF` statements can be nested, as in Example 4–3.

***Example 4–3 Nested IF THEN ELSE Statements***

```
DECLARE
  PROCEDURE p (
    sales  NUMBER,
    quota  NUMBER,
    emp_id NUMBER
  )
  IS
    bonus  NUMBER := 0;
  BEGIN
    IF sales > (quota + 200) THEN
      bonus := (sales - quota)/4;
    ELSE
      IF sales > quota THEN
        bonus := 50;
      ELSE
        bonus := 0;
      END IF;
    END IF;

    DBMS_OUTPUT.PUT_LINE('bonus = ' || bonus);

    UPDATE employees
    SET salary = salary + bonus
    WHERE employee_id = emp_id;
  END p;
BEGIN
  p(10100, 10000, 120);
  p(10500, 10000, 121);
  p(9500, 10000, 122);
```

```
END;
/
```

Result:

```
bonus = 50
bonus = 125
bonus = 0
```

## IF THEN ELSIF Statement

The IF THEN ELSIF statement has this structure:

```
IF condition_1 THEN
  statements_1
ELSIF condition_2 THEN
  statements_2
[ ELSIF condition_3 THEN
    statements_3
]...
[ ELSE
    else_statements
]
END IF;
```

The IF THEN ELSIF statement runs the first *statements* for which *condition* is true. Remaining conditions are not evaluated. If no *condition* is true, the *else_statements* run, if they exist; otherwise, the IF THEN ELSIF statement does nothing. (For complete syntax, see "IF Statement" on page 13-82.)

In Example 4–4, when the value of sales is larger than 50000, both the first and second conditions are true. However, because the first condition is true, bonus is assigned the value 1500, and the second condition is never tested. After bonus is assigned the value 1500, control passes to the DBMS_OUTPUT.PUT_LINE invocation.

***Example 4–4 IF THEN ELSIF Statement***

```
DECLARE
  PROCEDURE p (sales NUMBER)
  IS
    bonus  NUMBER := 0;
  BEGIN
    IF sales > 50000 THEN
      bonus := 1500;
    ELSIF sales > 35000 THEN
      bonus := 500;
    ELSE
      bonus := 100;
    END IF;

    DBMS_OUTPUT.PUT_LINE (
      'Sales = ' || sales || ', bonus = ' || bonus || '.'
    );
  END p;
BEGIN
  p(55000);
  p(40000);
  p(30000);
END;
/
```

Result:

```
Sales = 55000, bonus = 1500.
Sales = 40000, bonus = 500.
Sales = 30000, bonus = 100.
```

A single `IF THEN ELSIF` statement is easier to understand than a logically equivalent nested `IF THEN ELSE` statement:

```
-- IF THEN ELSIF statement

IF condition_1 THEN statements_1;
  ELSIF condition_2 THEN statements_2;
  ELSIF condition_3 THEN statement_3;
END IF;

-- Logically equivalent nested IF THEN ELSE statements

IF condition_1 THEN
  statements_1;
ELSE
  IF condition_2 THEN
    statements_2;
  ELSE
    IF condition_3 THEN
      statements_3;
    END IF;
  END IF;
END IF;
```

Example 4–5 uses an `IF THEN ELSIF` statement with many `ELSIF` clauses to compare a single value to many possible values. For this purpose, a simple `CASE` statement is clearer—see Example 4–6.

***Example 4–5 IF THEN ELSIF Statement that Simulates Simple CASE Statement***

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  IF grade = 'A' THEN
    DBMS_OUTPUT.PUT_LINE('Excellent');
  ELSIF grade = 'B' THEN
    DBMS_OUTPUT.PUT_LINE('Very Good');
  ELSIF grade = 'C' THEN
    DBMS_OUTPUT.PUT_LINE('Good');
  ELSIF grade = 'D' THEN
    DBMS_OUTPUT. PUT_LINE('Fair');
  ELSIF grade = 'F' THEN
    DBMS_OUTPUT.PUT_LINE('Poor');
  ELSE
    DBMS_OUTPUT.PUT_LINE('No such grade');
  END IF;
END;
/
```

Result:

```
Very Good
```

## Simple CASE Statement

The simple CASE statement has this structure:

```
CASE selector
WHEN selector_value_1 THEN statements_1
WHEN selector_value_2 THEN statements_2
...
WHEN selector_value_n THEN statements_n
[ ELSE
  else_statements ]
END CASE;]
```

The *selector* is an expression (typically a single variable). Each *selector_value* can be either a literal or an expression. (For complete syntax, see "CASE Statement" on page 13-18.)

The simple CASE statement runs the first *statements* for which *selector_value* equals *selector*. Remaining conditions are not evaluated. If no *selector_value* equals *selector*, the CASE statement runs *else_statements* if they exist and raises the predefined exception CASE_NOT_FOUND otherwise.

Example 4–6 uses a simple CASE statement to compare a single value to many possible values. The CASE statement in Example 4–6 is logically equivalent to the IF THEN ELSIF statement in Example 4–5.

***Example 4–6 Simple CASE Statement***

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  CASE grade
    WHEN 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/
```

Result:

```
Very Good
```

---

**Note:** As in a simple CASE expression, if the selector in a simple CASE statement has the value NULL, it cannot be matched by WHEN NULL (see Example 2–51, "Simple CASE Expression with WHEN NULL"). Instead, use a searched CASE statement with WHEN *condition* IS NULL (see Example 2–53, "Searched CASE Expression with WHEN condition IS NULL").

---

## Searched CASE Statement

The searched CASE statement has this structure:

```
CASE
```

```
WHEN condition_1 THEN statements_1
WHEN condition_2 THEN statements_2
...
WHEN condition_n THEN statements_n
[ ELSE
  else_statements ]
END CASE;]
```

The searched CASE statement runs the first *statements* for which *condition* is true. Remaining conditions are not evaluated. If no *condition* is true, the CASE statement runs *else_statements* if they exist and raises the predefined exception CASE_NOT_FOUND otherwise. (For complete syntax, see "CASE Statement" on page 13-18.)

The searched CASE statement in Example 4–7 is logically equivalent to the simple CASE statement in Example 4–6.

***Example 4–7 Searched CASE Statement***

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  CASE
    WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
    ELSE DBMS_OUTPUT.PUT_LINE('No such grade');
  END CASE;
END;
/
```

Result:

```
Very Good
```

In both Example 4–7 and Example 4–6, the ELSE clause can be replaced by an EXCEPTION part. Example 4–8 is logically equivalent to Example 4–7.

***Example 4–8 EXCEPTION Instead of ELSE Clause in CASE Statement***

```
DECLARE
  grade CHAR(1);
BEGIN
  grade := 'B';

  CASE
    WHEN grade = 'A' THEN DBMS_OUTPUT.PUT_LINE('Excellent');
    WHEN grade = 'B' THEN DBMS_OUTPUT.PUT_LINE('Very Good');
    WHEN grade = 'C' THEN DBMS_OUTPUT.PUT_LINE('Good');
    WHEN grade = 'D' THEN DBMS_OUTPUT.PUT_LINE('Fair');
    WHEN grade = 'F' THEN DBMS_OUTPUT.PUT_LINE('Poor');
  END CASE;
EXCEPTION
  WHEN CASE_NOT_FOUND THEN
    DBMS_OUTPUT.PUT_LINE('No such grade');
END;
/
```

Result:

```
Very Good
```

# LOOP Statements

**Loop statements** run the same statements with a series of different data values. The loop statements are:

- Basic LOOP
- WHILE LOOP
- FOR LOOP
- Cursor FOR LOOP

The statements that exit a loop are:

- EXIT
- EXIT WHEN

The statements that exit the current iteration of a loop are:

- CONTINUE
- CONTINUE WHEN

EXIT, EXIT WHEN, CONTINUE, and CONTINUE WHEN and can appear anywhere inside a loop, but not outside a loop. These statements are recommended over the "GOTO Statement" on page 4-21, which can exit a loop or the current iteration of a loop by transferring control to a statement outside the loop. A raised exception also exits a loop.

LOOP statements can be labeled. LOOP statements can nested. Labels are recommended for nested loops to improve readability. You must ensure that the label at the end of a LOOP statement matches one of the labels at the beginning of the same statement (the compiler does not check).

Topics:

- Basic LOOP Statement
- EXIT Statement
- EXIT WHEN Statement
- CONTINUE Statement
- CONTINUE WHEN Statement
- WHILE LOOP Statement
- FOR LOOP Statement

For information about the cursor FOR LOOP, see "Looping Through Multiple Rows (Cursor FOR LOOP Statement)" on page 6-21.

## Basic LOOP Statement

The basic LOOP statement has this structure:

```
[ label ] LOOP
  statements
END LOOP [ label ];
```

With each iteration of the loop, the *statements* run and control returns to the top of the loop. To prevent an infinite loop, one of the *statements* or a raised exception must exit the loop.

**See Also:** "Basic LOOP Statement" on page 13-16

## EXIT Statement

The EXIT statement exits the current iteration of a loop unconditionally and transfers control to the end of either the current loop or an enclosing labeled loop.

In Example 4–9, the EXIT statement inside the basic LOOP statement transfers control unconditionally to the end of the current loop.

***Example 4–9 Basic LOOP Statement with EXIT Statement***

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x > 3 THEN
      EXIT;
    END IF;
  END LOOP;
  -- After EXIT, control resumes here
  DBMS_OUTPUT.PUT_LINE(' After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4
```

**See Also:** "EXIT Statement" on page 13-52

## EXIT WHEN Statement

The EXIT WHEN statement exits the current iteration of a loop when the condition in its WHEN clause is true, and transfers control to the end of either the current loop or an enclosing labeled loop.

Each time control reaches the EXIT WHEN statement, the condition in its WHEN clause is evaluated. If the condition is not true, the EXIT WHEN statement does nothing. To prevent an infinite loop, a statement inside the loop must make the condition true, as in Example 4–10.

In Example 4–10, the EXIT WHEN statement inside the basic LOOP statement transfers control to the end of the current loop when x is greater than 3. Example 4–10 is logically equivalent to Example 4–9.

***Example 4–10 Basic LOOP Statement with EXIT WHEN Statement***

```
DECLARE
```

```
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;  -- prevents infinite loop
    EXIT WHEN x > 3;
  END LOOP;
  -- After EXIT statement, control resumes here
  DBMS_OUTPUT.PUT_LINE('After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop:  x = 3
After loop:  x = 4
```

**See Also:** "EXIT Statement" on page 13-52

In Example 4–11, one basic LOOP statement is nested inside the other, and both have labels. The inner loop has two EXIT WHEN statements; one that exits the inner loop and one that exits the outer loop.

***Example 4–11 Nested, Labeled Basic LOOP Statements with EXIT WHEN Statements***

```
DECLARE
  s  PLS_INTEGER := 0;
  i  PLS_INTEGER := 0;
  j  PLS_INTEGER;
BEGIN
  <<outer_loop>>
  LOOP
    i := i + 1;
    j := 0;
    <<inner_loop>>
    LOOP
      j := j + 1;
      s := s + i * j; -- Sum several products
      EXIT inner_loop WHEN (j > 5);
      EXIT outer_loop WHEN ((i * j) > 15);
    END LOOP inner_loop;
  END LOOP outer_loop;
  DBMS_OUTPUT.PUT_LINE
    ('The sum of products equals: ' || TO_CHAR(s));
END;
/
```

Result:

```
The sum of products equals: 166
```

## CONTINUE Statement

The CONTINUE statement exits the current iteration of a loop unconditionally and transfers control to the next iteration of either the current loop or an enclosing labeled loop.

In Example 4–12, the CONTINUE statement inside the basic LOOP statement transfers control unconditionally to the next iteration of the current loop.

***Example 4–12 CONTINUE Statement in Basic LOOP Statement***

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    IF x < 3 THEN
      CONTINUE;
    END IF;
    DBMS_OUTPUT.PUT_LINE
      ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Inside loop, after CONTINUE:  x = 5
After loop:  x = 5
```

**See Also:** "CONTINUE Statement" on page 13-35

## CONTINUE WHEN Statement

The CONTINUE WHEN statement exits the current iteration of a loop when the condition in its WHEN clause is true, and transfers control to the next iteration of either the current loop or an enclosing labeled loop.

Each time control reaches the CONTINUE WHEN statement, the condition in its WHEN clause is evaluated. If the condition is not true, the CONTINUE WHEN statement does nothing.

In Example 4–13, the CONTINUE WHEN statement inside the basic LOOP statement transfers control to the next iteration of the current loop when x is less than 3. Example 4–13 is logically equivalent to Example 4–12.

***Example 4–13 CONTINUE WHEN Statement in Basic LOOP Statement***

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP -- After CONTINUE statement, control resumes here
    DBMS_OUTPUT.PUT_LINE ('Inside loop:  x = ' || TO_CHAR(x));
    x := x + 1;
    CONTINUE WHEN x < 3;
    DBMS_OUTPUT.PUT_LINE
```

```
      ('Inside loop, after CONTINUE:  x = ' || TO_CHAR(x));
    EXIT WHEN x = 5;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE (' After loop:  x = ' || TO_CHAR(x));
END;
/
```

Result:

```
Inside loop:  x = 0
Inside loop:  x = 1
Inside loop:  x = 2
Inside loop, after CONTINUE:  x = 3
Inside loop:  x = 3
Inside loop, after CONTINUE:  x = 4
Inside loop:  x = 4
Inside loop, after CONTINUE:  x = 5
After loop:  x = 5
```

**See Also:** "CONTINUE Statement" on page 13-35

## WHILE LOOP Statement

The WHILE LOOP statement runs one or more statements while a condition is true. It has this structure:

```
[ label ] WHILE condition LOOP
  statements
END LOOP [ label ];
```

If the *condition* is true, the *statements* run and control returns to the top of the loop, where *condition* is evaluated again. If the *condition* is not true, control transfers to the statement after the WHILE LOOP statement. To prevent an infinite loop, a statement inside the loop must make the condition false or null. For complete syntax, see "WHILE LOOP Statement" on page 13-133.

In Example 4–14, the statements in the first WHILE LOOP statement never run, and the statements in the second WHILE LOOP statement run once.

***Example 4–14 WHILE LOOP Statements***

```
DECLARE
  done  BOOLEAN := FALSE;
BEGIN
  WHILE done LOOP
    DBMS_OUTPUT.PUT_LINE ('This line does not print.');
    done := TRUE;  -- This assignment is not made.
  END LOOP;

  WHILE NOT done LOOP
    DBMS_OUTPUT.PUT_LINE ('Hello, world!');
    done := TRUE;
  END LOOP;
END;
/
```

Result:

```
Hello, world!
```

Some languages have a LOOP UNTIL or REPEAT UNTIL structure, which tests a condition at the bottom of the loop instead of at the top, so that the statements run at least once. To simulate this structure in PL/SQL, use a basic LOOP statement with an EXIT WHEN statement:

```
LOOP
  statements
  EXIT WHEN condition;
END LOOP;
```

## FOR LOOP Statement

The FOR LOOP statement runs one or more statements while an index is in a specified range. The statement has this structure:

```
[ label ] FOR index IN [ REVERSE ] lower_bound..upper_bound LOOP
  statements
END LOOP [ label ];
```

Without REVERSE, the value of *index* starts at *lower_bound* and increases by one with each iteration of the loop until it reaches *upper_bound*.

With REVERSE, the value of *index* starts at *upper_bound* and decreases by one with each iteration of the loop until it reaches *lower_bound*.

For the complete syntax and semantics of the FOR LOOP statement, see "FOR LOOP Statement" on page 13-70.

> **Tip:** To process the rows of a query result set, use a cursor FOR LOOP, which has a query instead of a range of integers. For details, see "Looping Through Multiple Rows (Cursor FOR LOOP Statement)" on page 6-21.

In Example 4–15, *index* is i, *lower_bound* is 1, and *upper_bound* is 3. The loop prints the numbers from 1 to 3.

***Example 4–15 FOR LOOP Statements***

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('lower_bound < upper_bound');

  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('lower_bound = upper_bound');

  FOR i IN 2..2 LOOP
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('lower_bound > upper_bound');

  FOR i IN 3..1 LOOP
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;
END;
/
```

Result:

```
lower_bound < upper_bound
1
2
3
lower_bound = upper_bound
2
lower_bound > upper_bound
```

The FOR LOOP statement in Example 4–16 is the reverse of the one in Example 4–15: It prints the numbers from 3 to 1.

***Example 4–16 Reverse FOR LOOP Statements***

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('upper_bound > lower_bound');

  FOR i IN REVERSE 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('upper_bound = lower_bound');

  FOR i IN REVERSE 2..2 LOOP
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('upper_bound > lower_bound');

  FOR i IN REVERSE 3..1 LOOP
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;
END;
/
```

Result:

```
upper_bound > lower_bound
3
2
1
upper_bound = lower_bound
2
upper_bound > lower_bound
```

Topics:

- FOR LOOP Index
- Lower Bound and Upper Bound
- EXIT Statement in FOR LOOP Statement

## FOR LOOP Index

The index of a FOR LOOP statement is implicitly declared as a variable of type INTEGER that is local to the loop. The statements in the loop can read the value of the index, but cannot change it. Statements outside the loop cannot reference the index. After the FOR LOOP statement runs, the index is undefined. (A loop index is sometimes called a loop counter.)

In Example 4–17, the FOR LOOP statement tries to change the value of its index, causing an error.

***Example 4–17 FOR LOOP Statement Tries to Change Index Value***

```
BEGIN
  FOR i IN 1..3 LOOP
    IF i < 3 THEN
      DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
    ELSE
      i := 2;
    END IF;
  END LOOP;
END;
/
```

Result:

```
       i := 2;
       *
ERROR at line 6:
ORA-06550: line 6, column 8:
PLS-00363: expression 'I' cannot be used as an assignment target
ORA-06550: line 6, column 8:
PL/SQL: Statement ignored
```

In Example 4–18, a statement outside the FOR LOOP statement references the loop index, causing an error.

***Example 4–18 Statement Outside FOR LOOP Tries to Reference Index***

```
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
END;
/
```

Result:

```
  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
                                                         *
ERROR at line 6:
ORA-06550: line 6, column 58:
PLS-00201: identifier 'I' must be declared
ORA-06550: line 6, column 3:
PL/SQL: Statement ignored
```

If the index of a FOR LOOP statement has the same name as a variable declared in an enclosing block, the local implicit declaration hides the other declaration, as Example 4–19 shows.

***Example 4–19 FOR LOOP Index with Same Name as Declared Variable***

```
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop, i is ' || TO_CHAR(i));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('Outside loop, i is ' || TO_CHAR(i));
END;
```

```
/
```

Result:

```
Inside loop, i is 1
Inside loop, i is 2
Inside loop, i is 3
Outside loop, i is 5
```

Example 4–20 shows how to change Example 4–19 to allow the statement inside the loop to reference the variable declared in the enclosing block.

***Example 4–20 FOR LOOP References Declared Variable with Same Name as Index***

```
<<main>>  -- Label block.
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (
      'local: ' || TO_CHAR(i) || ', global: ' ||
      TO_CHAR(main.i)  -- Qualify reference with block label.
    );
  END LOOP;
END main;
/
```

Result:

```
local: 1, global: 5
local: 2, global: 5
local: 3, global: 5
```

In Example 4–21, the indexes of the nested `FOR LOOP` statements have the same name. The inner loop references the index of the outer loop by qualifying the reference with the label of the outer loop. For clarity only, the inner loop also qualifies the reference to its own index with its own label.

***Example 4–21 Nested FOR LOOP Statements with Same Index Name***

```
BEGIN
  <<outer_loop>>
  FOR i IN 1..3 LOOP
    <<inner_loop>>
    FOR i IN 1..3 LOOP
      IF outer_loop.i = 2 THEN
        DBMS_OUTPUT.PUT_LINE
          ('outer: ' || TO_CHAR(outer_loop.i) || ' inner: '
           || TO_CHAR(inner_loop.i));
      END IF;
    END LOOP inner_loop;
  END LOOP outer_loop;
END;
/
```

Result:

```
outer: 2 inner: 1
outer: 2 inner: 2
outer: 2 inner: 3
```

### Lower Bound and Upper Bound

The bounds of a loop range can be either literals, variables, or expressions, but they must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception VALUE_ERROR.

***Example 4–22 Several Types of FOR LOOP Bounds***

```
DECLARE
  first  INTEGER := 1;
  last   INTEGER := 10;
  high   INTEGER := 100;
  low    INTEGER := 12;
BEGIN
  -- Bounds are numeric literals:
  FOR j IN -5..5 LOOP
    NULL;
  END LOOP;

  -- Bounds are numeric variables:
  FOR k IN REVERSE first..last LOOP
    NULL;
  END LOOP;

 -- Lower bound is numeric literal,
 -- Upper bound is numeric expression:
  FOR step IN 0..(TRUNC(high/low) * 2) LOOP
    NULL;
  END LOOP;
END;
/
```

Some languages provide a STEP clause, which lets you specify a different increment (5 instead of 1, for example). PL/SQL has no such structure, but you can easily build one. Inside the FOR loop, simply multiply each reference to the loop counter by the increment.

Example 4–23 assigns today's date to elements 5, 10, and 15 of an associative array.

***Example 4–23 Changing the Increment of the Counter in a FOR LOOP Statement***

```
DECLARE
  TYPE DateList IS TABLE OF DATE INDEX BY PLS_INTEGER;
  dates DateList;
BEGIN
  FOR j IN 1..3 LOOP
    dates(j*5) := SYSDATE;
  END LOOP;
END;
/
```

PL/SQL lets you specify the loop range at run time by using variables for bounds as shown in Example 4–24.

***Example 4–24 Specifying a LOOP Range at Run Time***

```
DROP TABLE temp;
CREATE TABLE temp (
  emp_no      NUMBER,
  email_addr  VARCHAR2(50)
);
```

```
DECLARE
  emp_count  NUMBER;
BEGIN
  SELECT COUNT(employee_id) INTO emp_count
  FROM employees;

  FOR i IN 1..emp_count LOOP
    INSERT INTO temp (emp_no, email_addr)
    VALUES(i, 'to be added later');
  END LOOP;
END;
/
```

If the lower bound of a loop range is larger than the upper bound, the loop body does not run and control passes to the next statement, as Example 4–25 shows.

***Example 4–25 FOR LOOP with Lower Bound > Upper Bound***

```
CREATE OR REPLACE PROCEDURE p (limit_ IN INTEGER) IS
  BEGIN
    FOR i IN 2..limit_ LOOP
      DBMS_OUTPUT.PUT_LINE ('Inside loop, limit_ is ' || i);
    END LOOP;

    DBMS_OUTPUT.PUT_LINE ('Outside loop, limit_ is ' || TO_CHAR(limit_));
END;
/
```

Invoke `p` with `limit_` 3:

```
BEGIN
  p(3);
END;
/
```

Result:

```
Inside loop, limit_ is 2
Inside loop, limit_ is 3
Outside loop, limit_ is 3

PL/SQL procedure successfully completed.
```

Invoke `p` with `limit_` 1:

```
BEGIN
  p(1);
END;
/
```

Result:

```
Outside loop, limit_ is 1
```

## EXIT Statement in FOR LOOP Statement

The `EXIT` statement lets a `FOR` loop complete early. In Example 4–26, the loop normally runs ten times, but as soon as the `FETCH` statement fails to return a row, the loop completes no matter how many times it has run.

***Example 4–26 EXIT in FOR LOOP***

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;
  -- Fetch entire row into v_employees record:
  FOR i IN 1..10 LOOP
    FETCH c1 INTO v_employees;
    EXIT WHEN c1%NOTFOUND;
    -- Process data here
  END LOOP;
  CLOSE c1;
END;
/
```

Suppose you must exit early from a nested FOR loop. To complete not only the current loop, but also any enclosing loop, label the enclosing loop and use the label in an EXIT statement as shown in Example 4–27. To complete the current iteration of the labeled loop and exit any enclosed loops, use a label in a CONTINUE statement.

***Example 4–27 EXIT with Label in FOR LOOP***

```
DECLARE
  v_employees employees%ROWTYPE;
  CURSOR c1 is SELECT * FROM employees;
BEGIN
  OPEN c1;

  -- Fetch entire row into v_employees record:
  <<outer_loop>>
  FOR i IN 1..10 LOOP
    -- Process data here
    FOR j IN 1..10 LOOP
      FETCH c1 INTO v_employees;
      EXIT outer_loop WHEN c1%NOTFOUND;
      -- Process data here
    END LOOP;
  END LOOP outer_loop;

  CLOSE c1;
END;
/
```

# Sequential Control Statements

Unlike the IF and LOOP statements, the **sequential control statements** GOTO and NULL are not crucial to PL/SQL programming.

The GOTO statement, which goes to a specified statement, is seldom needed. Occasionally, it simplifies logic enough to warrant its use.

The NULL statement, which does nothing, can improve readability by making the meaning and action of conditional statements clear.

Topics:

- GOTO Statement
- NULL Statement

## GOTO Statement

The GOTO statement transfers control to a label unconditionally. The label must be unique in its scope and must precede an executable statement or a PL/SQL block. When run, the GOTO statement transfers control to the labeled statement or block. For GOTO statement restrictions, see "GOTO Statement" on page 13-80.

Use GOTO statements sparingly—overusing them results in code that is hard to understand and maintain. Do not use a GOTO statement to transfer control from a deeply nested structure to an exception handler. Instead, raise an exception. For information about the PL/SQL exception-handling mechanism, see Chapter 11, "PL/SQL Error Handling."

***Example 4–28 GOTO Statement***

```
DECLARE
  p  VARCHAR2(30);
  n  PLS_INTEGER := 37;
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
    IF n MOD j = 0 THEN
      p := ' is not a prime number';
      GOTO print_now;
    END IF;
  END LOOP;

  p := ' is a prime number';

  <<print_now>>
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n) || p);
END;
/
```

Result:

```
37 is a prime number
```

A label can appear only before a block (as in Example 4–20) or before a statement (as in Example 4–28), not in a statement, as in Example 4–29.

***Example 4–29 Incorrect Label Placement***

```
DECLARE
  done  BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
    IF done THEN
       GOTO end_loop;
    END IF;
    <<end_loop>>
  END LOOP;
END;
/
```

Result:

```
  END LOOP;
  *
ERROR at line 9:
ORA-06550: line 9, column 3:
PLS-00103: Encountered the symbol "END" when expecting one of the following:
```

```
( begin case declare exit for goto if loop mod null raise
return select update while with <an identifier>
<a double-quoted delimited-identifier> <a bind variable> <<
continue close current delete fetch lock insert open rollback
savepoint set sql run commit forall merge pipe purge
```

To correct Example 4–29, add a NULL statement, as in Example 4–30.

***Example 4–30 NULL Statement Allows GOTO to Label***

```
DECLARE
  done  BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
    IF done THEN
      GOTO end_loop;
    END IF;
    <<end_loop>>
    NULL;
  END LOOP;
END;
/
```

A GOTO statement can transfer control to an enclosing block from the current block, as in Example 4–31.

***Example 4–31 GOTO Statement Transfers Control to Enclosing Block***

```
DECLARE
  v_last_name  VARCHAR2(25);
  v_emp_id     NUMBER(6) := 120;
BEGIN
  <<get_name>>
  SELECT last_name INTO v_last_name
  FROM employees
  WHERE employee_id = v_emp_id;

  BEGIN
    DBMS_OUTPUT.PUT_LINE (v_last_name);
    v_emp_id := v_emp_id + 5;

    IF v_emp_id < 120 THEN
      GOTO get_name;
    END IF;
  END;
END;
/
```

Result:

```
Weiss
```

The GOTO statement transfers control to the first enclosing block in which the referenced label appears.

The GOTO statement in Example 4–32 transfers control into an IF statement, causing an error.

***Example 4–32 GOTO Statement Cannot Transfer Control into IF Statement***

```
DECLARE
```

```
  valid BOOLEAN := TRUE;
BEGIN
  GOTO update_row;

  IF valid THEN
  <<update_row>>
    NULL;
  END IF;
END;
/
```

Result:

```
  GOTO update_row;
  *
ERROR at line 4:
ORA-06550: line 4, column 3:
PLS-00375: illegal GOTO statement; this GOTO cannot transfer control to label
'UPDATE_ROW'
ORA-06550: line 6, column 12:
PL/SQL: Statement ignored
```

## NULL Statement

The NULL statement only passes control to the next statement. Some languages refer to such an instruction as a no-op (no operation).

Some uses for the NULL statement are:

- To provide a target for a GOTO statement, as in Example 4–30.
- To improve readability by making the meaning and action of conditional statements clear, as in Example 4–33
- To create placeholders and stub subprograms, as in Example 4–34
- To show that you are aware of a possibility, but that no action is necessary, as in Example 4–35

In Example 4–33, the NULL statement emphasizes that only salespersons receive commissions.

***Example 4–33 NULL Statement Showing No Action***

```
DECLARE
  v_job_id  VARCHAR2(10);
   v_emp_id  NUMBER(6) := 110;
BEGIN
  SELECT job_id INTO v_job_id
  FROM employees
  WHERE employee_id = v_emp_id;

  IF v_job_id = 'SA_REP' THEN
    UPDATE employees
    SET commission_pct = commission_pct * 1.2;
  ELSE
    NULL;  -- Employee is not a sales rep
  END IF;
END;
/
```

In Example 4–34, the NULL statement lets you compile this subprogram and fill in the real body later.

---

**Note:** Using the NULL statement might raise an unreachable code warning if warnings are enabled. For information about warnings, see "Compile-Time Warnings" on page 11-1.

---

***Example 4–34 NULL Statement as Placeholder During Subprogram Creation***

```
DROP PROCEDURE award_bonus;
CREATE OR REPLACE PROCEDURE award_bonus (
  emp_id NUMBER,
  bonus NUMBER
) AS
BEGIN    -- Executable part starts here
  NULL;  -- Placeholder
  -- (raises "unreachable code" if warnings enabled)
END award_bonus;
/
```

In Example 4–35, the NULL statement shows that you have chosen not to take any action for unnamed exceptions.

***Example 4–35 NULL Statement in WHEN OTHER Clause***

```
CREATE OR REPLACE FUNCTION f (
  a INTEGER,
  b INTEGER
) RETURN INTEGER
AS
BEGIN
  RETURN (a/b);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Attempted division by zero.');
  WHEN OTHERS THEN
    NULL;
END;
/
```

# 5

# PL/SQL Collections and Records

This chapter explains how to create and use PL/SQL collection and record variables. These **composite variables** have internal components that you can treat as individual variables. You can pass composite variables to subprograms as parameters.

To create a collection or record variable, you first define a collection or record type, and then you declare a variable of that type. In this book, *collection* or *record* means both the type and the variables of that type, unless otherwise noted.

In a **collection**, the internal components are always of the same data type, and are called **elements**. You access each element by its unique subscript. Lists and arrays are classic examples of collections.

In a **record**, the internal components can be of different data types, and are called **fields**. You access each field with this syntax: `record_name.field_name`. A record variable can hold a table row, or some columns from a table row. Each record field corresponds to a table column.

#### Collection Topics

- PL/SQL Collection Type Characteristics
- Collection Type Definitions
- Collection Variable Declarations
- Collection Initialization
- Collection Element References
- Assignments to Collections
- Collection Comparisons
- Multidimensional Collections
- Collection Methods
- Collection Exceptions

#### Record Topics

- Record Definitions and Declarations
- Records as Subprogram Parameters and Function Return Values
- Assignments to Records
- Record Comparisons
- Inserting Records into Tables

- Updating Rows with Records
- Returning Rows into Records
- Restrictions on Record Inserts and Updates
- Assignments to Collections of Records

## PL/SQL Collection Type Characteristics

PL/SQL has three collection types, whose characteristics are summarized in Table 5–1.

***Table 5–1 Characteristics of PL/SQL Collection Types***

| Collection Type | Number of Elements | Subscript Type | Dense or Sparse | Where Created | Can Be ADT Attribute |
|---|---|---|---|---|---|
| Associative array (or index-by table) | Unbounded | String or integer | Either | Only in PL/SQL block | No |
| Nested table | Unbounded | Integer | Starts dense, can become sparse | Either in PL/SQL block or at schema level | Yes |
| Variable-size array (varray) | Bounded | Integer | Always dense | Either in PL/SQL block or at schema level | Yes |

**Unbounded** means that, theoretically, there is no limit to the number of elements in the collection. Actually, there is a limit, but it is very high—for details, see "Collection Element References" on page 5-12.

**Dense** means that the collection has no gaps between elements—every element between the first and last element is defined and has a value (which can be `NULL`).

A collection that is created in a PL/SQL block (with the syntax in "Collection" on page 13-23) is available only in that block. A nested table type or varray type that is created at schema level (with the "CREATE TYPE Statement" on page 14-69) is stored in the database, and you can manipulate it with SQL statements.

A collection has only one dimension, but you can model a multidimensional collection by creating a collection whose elements are also collections. For examples, see "Multidimensional Collections" on page 5-20.

If you have code or business logic that uses another language, you can usually translate the array and set types of that language directly to PL/SQL collection types. For example:

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

Topics:

- Associative Arrays
- Nested Tables
- Variable-Size Arrays (Varrays)

> **See Also:** *Oracle Database SQL Language Reference* for information about the CAST function, which converts one SQL data type or collection-typed value into another SQL data type or collection-typed value.

## Associative Arrays

An associative array (formerly called an index-by table) is a set of key-value pairs. Each key is unique, and is used to locate the corresponding value. The key can be either an integer or a string.

Using a key-value pair for the first time adds that pair to the associative array. Using the same key with a different value changes the value.

Example 5–1 declares an associative array that is indexed by a string, populates it, and prints it.

***Example 5–1 Declaring and Populating Associative Array Indexed by String***

```
DECLARE
  -- Associative array indexed by string:

  TYPE population IS TABLE OF NUMBER  -- Associative array type
    INDEX BY VARCHAR2(64);

  city_population  population;        -- Associative array variable
  i                VARCHAR2(64);

BEGIN
  -- Add elements to associative array:

  city_population('Smallville')  := 2000;
  city_population('Midland')     := 750000;
  city_population('Megalopolis') := 1000000;

  -- Change value associated with key 'Smallville':

  city_population('Smallville') := 2001;

  -- Print associative array:

  i := city_population.FIRST;

  WHILE i IS NOT NULL LOOP
    DBMS_Output.PUT_LINE
      ('Population of ' || i || ' is ' || TO_CHAR(city_population(i)));
    i := city_population.NEXT(i);
   END LOOP;
END;
/
```

Result:

```
Population of Megalopolis is 1000000
Population of Midland is 750000
Population of Smallville is 2001
```

Like a database table, an associative array holds a data set of arbitrary size, and you can access its elements without knowing their positions in the array. An associative array does not need the disk space or network operations of a database table, but an

associative array cannot be manipulated by SQL statements (such as INSERT and DELETE).

An associative array is intended for temporary data storage. To make an associative array persistent for the life of a database session, declare the associative array (the type and the variable of that type) in a package, and assign values to its elements in the package body.

An associative array is appropriate for:

- A relatively small lookup table, where the collection can be constructed in memory each time a subprogram is invoked or a package is initialized
- Passing collections to and from the database server

  PL/SQL automatically converts between host arrays and associative arrays that use numeric key values. The most efficient way to pass collections to and from the database server is to set up data values in associative arrays, and then use those associative arrays with bulk constructs (the FORALL statement or BULK COLLECT clause).

With the Oracle Call Interface (OCI) or the Oracle Precompilers, you can bind host arrays to associative arrays declared as the formal parameters of a subprogram. That lets you pass host arrays to stored functions and procedures.

#### NLS Parameter Values Can Affect String Keys of Associative Arrays

Associative arrays that are indexed by strings can be affected by National Language Support (NLS) parameters such as NLS_SORT, NLS_COMP, and NLS_DATE_FORMAT.

As Example 5–1 shows, string keys of an associative array are not stored in creation order, but in sorted order. Sorted order is determined by the initialization parameters NLS_SORT and NLS_COMP. If you change the setting of either of these parameters after populating an associated array, and then try to traverse the array, you might get an error when using a collection method such as NEXT or PRIOR. If you must change these settings during your session, set them back to their original values before performing further operations on associative arrays that are indexed by strings.

When you declare an associative array that is indexed by strings, the string type in the declaration must be VARCHAR2 or one of its subtypes. However, the key values with which you populate the array can be of any data type that can be converted to VARCHAR2 by the TO_CHAR function.

If you use key values of data types other than VARCHAR2 and its subtypes, ensure thatthese key values will be consistent and unique even if the settings of initialization parameters change. For example:

- Do not use TO_CHAR(SYSDATE) as a key value. If the NLS_DATE_FORMAT initialization parameter setting changes, *array_element*(TO_CHAR(SYSDATE)) might return a different result.
- Two different NVARCHAR2 values might be converted to the same VARCHAR2 value (containing question marks instead of certain national characters), in which case *array_element*(*national_string1*) and *array_element*(*national_string2*) would refer to the same element.
- Two CHAR or VARCHAR2 values that differ only in case, accented characters, or punctuation characters might also be considered the same if the value of the NLS_SORT initialization parameter ends in _CI (case-insensitive comparisons) or _AI (accent- and case-insensitive comparisons).

When you pass an associative array as a parameter to a remote database using a database link, the two databases can have different globalization settings. When the remote database uses a collection method such as `FIRST` or `NEXT`, it uses its own character order, which might be different from the order where the collection originated. If character set differences mean that two keys that were unique are not unique on the remote database, the program raises a `VALUE_ERROR` exception.

> **See Also:** *Oracle Database Globalization Support Guide* for information about linguistic sort parameters

## Nested Tables

Conceptually, a nested table is like a one-dimensional array with an arbitrary number of elements.

In the database, a nested table is a column type that holds a set of values. The database stores the rows of a nested table in no particular order. When you retrieve a nested table from the database into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. These subscripts give you array-like access to individual rows.

A nested table differs from an array in these important ways:

- An array has a declared number of elements, but a nested table does not. The size of a nested table can increase dynamically (however, a maximum limit is imposed—see "Collection Element References" on page 5-12).
- An array is always dense (that is, it always has consecutive subscripts). A nested array is dense initially, but it can become sparse, because you can delete elements from it.

Figure 5–1 shows the important differences between a nested table and an array.

***Figure 5–1 Array and Nested Table***

**Array of Integers**

| 321 | 17 | 99 | 407 | 83 | 622 | 105 | 19 | 67 | 278 |
|---|---|---|---|---|---|---|---|---|---|
| **x(1)** | **x(2)** | **x(3)** | **x(4)** | **x(5)** | **x(6)** | **x(7)** | **x(8)** | **x(9)** | **x(10)** |

**Fixed Upper Bound**

**Nested Table after Deletions**

| 321 | | 99 | 407 | | 622 | 105 | 19 | | 278 |
|---|---|---|---|---|---|---|---|---|---|
| **x(1)** | | **x(3)** | **x(4)** | | **x(6)** | **x(7)** | **x(8)** | | **x(10)** |

**Unbounded**

A nested table can be stored in a database column; therefore, you can use a nested table to simplify SQL operations in which you join a single-column table with a larger table.

You cannot rely on the order and subscripts of a nested table remaining stable as the nested table is stored in and retrieved from the database, because the order and subscripts are not preserved in the database.

A nested table is appropriate when:

- Index values are not consecutive.
- There is no set number of index values.
- You must delete or update some elements, but not all elements simultaneously.

Nested table data is stored in a separate store table, a system-generated database table. When you access a nested table, the database joins the nested table with its store table. This makes nested tables suitable for queries and updates that affect only some elements of the collection.

- You would create a separate lookup table, with multiple entries for each row of the main table, and access it through join queries.

The amount of memory allocated to a nested table can increase or decrease dynamically. As you delete elements, memory is freed page by page. If you delete the entire table, all the memory is freed.

## Variable-Size Arrays (Varrays)

A variable-size array (varray) is an item of the data type `VARRAY`. A varray has a maximum size, which you specify in its type definition. A varray can contain a varying number of elements, from zero (when empty) to the maximum size. A varray index has a fixed lower bound of 1 and an extensible upper bound. To access an element of a varray, you use standard subscripting syntax.

Figure 5–2 shows a varray named `Grades`, which has maximum size 10 and contains seven elements. The current upper bound for `Grades` is 7, but you can increase it to the maximum of 10. `Grades(`*n*`)` references the *n*th element of `Grades`.

***Figure 5–2 Varray of Size 10***

**Varray *Grades***

| B | C | A | A | C | D | B | | | |
|---|---|---|---|---|---|---|---|---|---|
| (1) | (2) | (3) | (4) | (5) | (6) | (7) | | | |

**Maximum Size = 10**

When stored in the database, a varray keeps its ordering and subscripts.

A varray is stored as a single object. If a varray is less than 4 KB, it is stored inside the table of which it is a column; otherwise, it is stored outside the table but in the same tablespace.

You must store or retrieve all elements of a varray at the same time, which is appropriate when operating on all the elements simultaneously, but might be impractical for large numbers of elements.

A varray is appropriate when:

- You know the number of elements in advance.
- The elements are usually accessed sequentially.

# Collection Type Definitions

To create a collection, you define a collection type and then declare variables of that type.

You can define a collection type either at schema level, inside a package, or inside a PL/SQL block. A collection type created at schema level is a **standalone stored type**. You create it with the `CREATE TYPE` statement. It is stored in the database until you drop it with the `DROP TYPE` statement.

A collection type created inside a package is a **packaged type**. It is stored in the database until you drop the package with the `DROP PACKAGE` statement.

A type created inside a PL/SQL block is available only inside that block, and is stored in the database only if that block is nested in a standalone or packaged subprogram.

Collections follow the same scoping and instantiation rules as other types and variables. Collections are instantiated when you enter a block or subprogram, and cease to exist when you exit. In a package, collections are instantiated when you first reference the package and cease to exist when you end the database session.

You can define TABLE and VARRAY types in the declarative part of any PL/SQL block, subprogram, or package using a TYPE definition.

For nested tables and varrays declared in PL/SQL, the element type of the table or varray can be any PL/SQL data type except REF CURSOR.

When defining a VARRAY type, you must specify its maximum size with a positive integer. In this example, you define a type that stores up to 366 dates:

```
DECLARE
   TYPE Calendar IS VARRAY(366) OF DATE;
```

Associative arrays let you insert elements using arbitrary key values. The keys need not be consecutive.

The key data type can be PLS_INTEGER, VARCHAR2, or one of VARCHAR2 subtypes VARCHAR, STRING, or LONG.

You must specify the length of a VARCHAR2-based key, except for LONG which is equivalent to declaring a key type of VARCHAR2(32760). The types RAW, LONG RAW, ROWID, CHAR, and CHARACTER are not allowed as keys for an associative array. The LONG and LONG RAW data types are supported only for backward compatibility; see "LONG and LONG RAW Data Types" on page 3-14 for more information.

An initialization clause is not allowed. There is no constructor notation for associative arrays. When you reference an element of an associative array that uses a VARCHAR2-based key, you can use other types, such as DATE or TIMESTAMP, if they can be converted to VARCHAR2 with the TO_CHAR function.

Associative arrays can store data using a primary key value as the index, where the key values are not sequential. Example 5–2 creates a single element in an associative array, with a subscript of 100 rather than 1.

***Example 5–2 Declaring an Associative Array***

```
DECLARE
   TYPE EmpTabTyp IS TABLE OF employees%ROWTYPE
      INDEX BY PLS_INTEGER;
   emp_tab EmpTabTyp;
BEGIN
   /* Retrieve employee record. */
   SELECT * INTO emp_tab(100) FROM employees
     WHERE employee_id = 100;
END;
/
```

**See Also:**


- "Collection" on page 13-23
- "CREATE TYPE Statement" on page 14-69

## Collection Variable Declarations

After defining a collection type, you declare variables of that type. You use the type name in the declaration, the same as with predefined types such as NUMBER.

***Example 5–3 Declaring Nested Tables, Varrays, and Associative Arrays***

```
DECLARE
   TYPE nested_type IS TABLE OF VARCHAR2(30);
   TYPE varray_type IS VARRAY(5) OF INTEGER;
   TYPE assoc_array_num_type
     IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
   TYPE assoc_array_str_type
     IS TABLE OF VARCHAR2(32) INDEX BY PLS_INTEGER;
   TYPE assoc_array_str_type2
     IS TABLE OF VARCHAR2(32) INDEX BY VARCHAR2(64);
   v1 nested_type;
   v2 varray_type;
   v3 assoc_array_num_type;
   v4 assoc_array_str_type;
   v5 assoc_array_str_type2;
BEGIN
-- an arbitrary number of strings can be inserted v1
   v1 := nested_type('Shipping','Sales','Finance','Payroll');
   v2 := varray_type(1, 2, 3, 4, 5); -- Up to 5 integers
   v3(99) := 10; -- Just start assigning to elements
   v3(7) := 100; -- Subscripts can be any integer values
   v4(42) := 'Smith'; -- Just start assigning to elements
   v4(54) := 'Jones'; -- Subscripts can be any integer values
   v5('Canada') := 'North America';
     -- Just start assigning to elements
   v5('Greece') := 'Europe';
     -- Subscripts can be string values
END;
/
```

As shown in Example 5–4, you can use %TYPE to specify the data type of a previously declared collection, so that changing the definition of the collection automatically updates other variables that depend on the number of elements or the element type.

***Example 5–4 Declaring Collections with %TYPE***

```
DECLARE
  TYPE few_depts  IS VARRAY(10)  OF VARCHAR2(30);
  TYPE many_depts IS VARRAY(100) OF VARCHAR2(64);
  some_depts few_depts;

  /* If the type of some_depts changes from few_depts to many_depts,
     local_depts and global_depts will use the same type
     when this block is recompiled */

  local_depts  some_depts%TYPE;
  global_depts some_depts%TYPE;
BEGIN
  NULL;
END;
/
```

You can declare collections as the formal parameters of subprograms. That way, you can pass collections to stored subprograms and from one subprogram to another. Example 5–5 declares a nested table as a parameter of a packaged subprogram.

***Example 5–5 Declaring a Procedure Parameter as a Nested Table***

```
CREATE PACKAGE personnel AS
   TYPE staff_list IS TABLE OF employees.employee_id%TYPE;
   PROCEDURE award_bonuses (empleos_buenos IN staff_list);
END personnel;
/

CREATE PACKAGE BODY personnel AS
 PROCEDURE award_bonuses (empleos_buenos staff_list) IS
  BEGIN
    FOR i IN empleos_buenos.FIRST..empleos_buenos.LAST
    LOOP
     UPDATE employees SET salary = salary + 100
         WHERE employees.employee_id = empleos_buenos(i);
   END LOOP;
  END;
 END;
/
```

To invoke `personnel.award_bonuses` from outside the package, you declare a variable of type `personnel.staff_list` and pass that variable as the parameter.

***Example 5–6 Invoking a Procedure with a Nested Table Parameter***

```
DECLARE
  good_employees personnel.staff_list;
BEGIN
  good_employees :=  personnel.staff_list(100, 103, 107);
  personnel.award_bonuses (good_employees);
END;
/
```

You can also specify a collection type in the `RETURN` clause of a function specification.

To specify the element type, you can use `%TYPE`, which provides the data type of a variable or database column. Also, you can use `%ROWTYPE`, which provides the rowtype of a cursor or database table. See Example 5–7 and Example 5–8.

***Example 5–7 Specifying Collection Element Types with %TYPE and %ROWTYPE***

```
DECLARE
-- Nested table type that can hold an arbitrary number
--   of employee IDs.
-- The element type is based on a column from the EMPLOYEES table.
-- You need not know whether the ID is a number or a string.
   TYPE EmpList IS TABLE OF employees.employee_id%TYPE;
-- Declare a cursor to select a subset of columns.
   CURSOR c1 IS SELECT employee_id FROM employees;
-- Declare an Array type that can hold information
--   about 10 employees.
-- The element type is a record that contains all the same
-- fields as the EMPLOYEES table.
   TYPE Senior_Salespeople IS VARRAY(10) OF employees%ROWTYPE;
-- Declare a cursor to select a subset of columns.
   CURSOR c2 IS SELECT first_name, last_name FROM employees;
```

```
-- Array type that can hold a list of names. The element type
-- is a record that contains the same fields as the cursor
-- (that is, first_name and last_name).
   TYPE NameList IS VARRAY(20) OF c2%ROWTYPE;
BEGIN
   NULL;
END;
/
```

Example 5–8 uses a RECORD type to specify the element type. See "Record Definitions and Declarations" on page 5-34.

***Example 5–8 VARRAY of Records***

```
DECLARE TYPE name_rec
  IS RECORD ( first_name VARCHAR2(20), last_name VARCHAR2(25));
   TYPE names IS VARRAY(250) OF name_rec;
BEGIN
   NULL;
END;
/
```

You can also impose a NOT NULL constraint on the element type, as shown in Example 5–9.

***Example 5–9 NOT NULL Constraint on Collection Elements***

```
DECLARE TYPE EmpList
  IS TABLE OF employees.employee_id%TYPE NOT NULL;
  v_employees EmpList := EmpList(100, 150, 160, 200);
BEGIN
   v_employees(3) := NULL; -- assigning NULL raises an exception
END;
/
```

Result:

```
   v_employees(3) := NULL; -- assigning NULL raises an exception
                     *
ERROR at line 5:
ORA-06550: line 5, column 22:
PLS-00382: expression is of wrong type
ORA-06550: line 5, column 4:
PL/SQL: Statement ignored
```

## Collection Initialization

Until you initialize it, a nested table or varray is atomically null; the collection itself is null, not its elements. To initialize a nested table or varray, you use a constructor, a system-defined function with the same name as the collection type. This function constructs collections from the elements passed to it.

You must explicitly call a constructor for each varray and nested table variable. Associative arrays, the third kind of collection, do not use constructors. Constructor calls are allowed wherever function calls are allowed.

Example 5–10 initializes a nested table using a constructor, which looks like a function with the same name as the collection type.

***Example 5–10 Constructor for a Nested Table***

```
DECLARE
   TYPE dnames_tab IS TABLE OF VARCHAR2(30);
   dept_names dnames_tab;
BEGIN
   dept_names := dnames_tab('Shipping','Sales','Finance','Payroll');
END;
/
```

Because a nested table does not have a declared size, you can put as many elements in the constructor as necessary.

Example 5–11 initializes a varray using a constructor, which looks like a function with the same name as the collection type.

***Example 5–11 Constructor for a Varray***

```
DECLARE
-- In the varray, put an upper limit on the number of elements
   TYPE dnames_var IS VARRAY(20) OF VARCHAR2(30);
   dept_names dnames_var;
BEGIN
-- Because dnames is declared as VARRAY(20),
-- you can put up to 20 elements in the constructor
   dept_names := dnames_var('Shipping','Sales','Finance','Payroll');
END;
/
```

Unless you impose the NOT NULL constraint in the type declaration, you can pass null elements to a constructor as in Example 5–12.

***Example 5–12 Collection Constructor Including Null Elements***

```
DECLARE
   TYPE dnames_tab IS TABLE OF VARCHAR2(30);
   dept_names dnames_tab;
   TYPE dnamesNoNulls_type IS TABLE OF VARCHAR2(30) NOT NULL;
BEGIN
   dept_names := dnames_tab('Shipping', NULL,'Finance', NULL);
-- If dept_names were of type dnamesNoNulls_type,
--  you could not include null values in the constructor
END;
/
```

You can initialize a collection in its declaration, which is a good programming practice, as shown in Example 5–13. In this case, you can invoke the collection's EXTEND method to add elements later.

***Example 5–13 Combining Collection Declaration and Constructor***

```
DECLARE
   TYPE dnames_tab IS TABLE OF VARCHAR2(30);
   dept_names dnames_tab :=
     dnames_tab('Shipping','Sales','Finance','Payroll');
BEGIN
   NULL;
END;
/
```

If you call a constructor without arguments, you get an empty but non-null collection as shown in Example 5–14.

***Example 5–14 Empty Varray Constructor***

```
DECLARE
   TYPE dnames_var IS VARRAY(20) OF VARCHAR2(30);
   dept_names dnames_var;
BEGIN
   IF dept_names IS NULL THEN
      DBMS_OUTPUT.PUT_LINE
        ('Before initialization, the varray is null.');
-- While the varray is null, you cannot check its COUNT attribute.
--   DBMS_OUTPUT.PUT_LINE
--     ('It has ' || dept_names.COUNT || ' elements.');
   ELSE
      DBMS_OUTPUT.PUT_LINE
        ('Before initialization, the varray is not null.');
   END IF;
   dept_names := dnames_var(); -- initialize empty varray
   IF dept_names IS NULL THEN
      DBMS_OUTPUT.PUT_LINE
        ('After initialization, the varray is null.');
   ELSE
      DBMS_OUTPUT.PUT_LINE
        ('After initialization, the varray is not null.');
      DBMS_OUTPUT.PUT_LINE
        ('It has ' || dept_names.COUNT || ' elements.');
   END IF;
END;
/
```

# Collection Element References

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

```
collection_name (subscript)
```

where *subscript* is an expression that usually yields an integer, or a VARCHAR2 for associative arrays declared with strings as keys.

The allowed subscript ranges are:

- For nested tables, 1..2147483647 (the upper limit of PLS_INTEGER).
- For varrays, 1.. *size_limit*, where you specify the limit in the declaration (*size_limit* cannot exceed 2147483647).
- For associative arrays with a numeric key, -2147483648..2147483647.
- For associative arrays with a string key, the length of the key and number of possible values depends on the VARCHAR2 length limit in the type declaration, and the database character set.

Example 5–15 shows how to reference an element in a nested table.

***Example 5–15 Referencing a Nested Table Element***

```
DECLARE
  TYPE Roster IS TABLE OF VARCHAR2(15);
```

```
  names Roster :=
    Roster('D Caruso', 'J Hamil', 'D Piro', 'R Singh');
  PROCEDURE verify_name(the_name VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(the_name);
  END;
BEGIN
  FOR i IN names.FIRST .. names.LAST
  LOOP
      IF names(i) = 'J Hamil' THEN
        DBMS_OUTPUT.PUT_LINE(names(i));
          -- reference to nested table element
      END IF;
  END LOOP;
  verify_name(names(3));
    -- procedure call that references element
END;
/
```

Example 5–16 shows how you can reference the elements of an associative array in a function call.

***Example 5–16 Referencing an Element of an Associative Array***

```
DECLARE
  TYPE sum_multiples IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
  n  PLS_INTEGER := 5;   -- number of multiples to sum for display
  sn PLS_INTEGER := 10;  -- number of multiples to sum
  m  PLS_INTEGER := 3;   -- multiple
FUNCTION get_sum_multiples
  (multiple IN PLS_INTEGER, num IN PLS_INTEGER)
  RETURN sum_multiples IS
  s sum_multiples;
  BEGIN
      FOR i IN 1..num LOOP
        s(i) := multiple * ((i * (i + 1)) / 2);
           -- sum of multiples
      END LOOP;
    RETURN s;
  END get_sum_multiples;
BEGIN
-- invoke function to retrieve
-- element identified by subscript (key)
  DBMS_OUTPUT.PUT_LINE
    ('Sum of the first ' || TO_CHAR(n) || ' multiples of ' ||
     TO_CHAR(m) || ' is ' || TO_CHAR(get_sum_multiples (m, sn)(n)));
END;
/
```

# Assignments to Collections

One collection can be assigned to another by an `INSERT`, `UPDATE`, `FETCH`, or `SELECT` statement, an assignment statement, or a subprogram call. You can assign the value of an expression to a specific element in a collection using the syntax:

```
collection_name (subscript) := expression;
```

where *`expression`* yields a value of the type specified for elements in the collection type definition.

You can use operators such as SET, MULTISET UNION, MULTISET INTERSECT, and MULTISET EXCEPT to transform nested tables as part of an assignment statement.

Assigning a value to a collection element can raise exceptions, for example:

- If the subscript is NULL or is not convertible to the right data type, PL/SQL raises the predefined exception VALUE_ERROR. Usually, the subscript must be an integer. Associative arrays can also be declared to have VARCHAR2 subscripts.
- If the subscript refers to an uninitialized element, PL/SQL raises SUBSCRIPT_BEYOND_COUNT.
- If the collection is atomically null, PL/SQL raises COLLECTION_IS_NULL.

For more information about collection exceptions, see "Collection Exceptions" on page 5-31, Example 5–38 on page 5-31, and "Predefined Exceptions" on page 11-7.

Example 5–17 shows that collections must have the same data type for an assignment to work. Having the same element type is not enough.

***Example 5–17 Data Type Compatibility for Collection Assignment***

```
DECLARE
  TYPE last_name_typ IS VARRAY(3) OF VARCHAR2(64);
  TYPE surname_typ IS VARRAY(3) OF VARCHAR2(64);

  -- These first two variables have the same data type.
  group1 last_name_typ := last_name_typ('Jones','Wong','Marceau');
  group2 last_name_typ := last_name_typ('Klein','Patsos','Singh');

  -- This third variable has a similar declaration,
  -- but is a different type.
  group3 surname_typ := surname_typ('Trevisi','Macleod','Marquez');
BEGIN
  -- Allowed because they have the same data type
  group1 := group2;
  -- Not allowed because they have different data types
  --   group3 := group2; -- raises an exception
END;
/
```

If you assign an atomically null nested table or varray to a second nested table or varray, the second collection must be reinitialized, as shown in Example 5–18. In the same way, assigning a NULL value to a collection makes it atomically null.

***Example 5–18 Assigning a Null Value to a Nested Table***

```
DECLARE
  TYPE dnames_tab IS TABLE OF VARCHAR2(30);

  -- This nested table has some values
  dept_names dnames_tab :=
    dnames_tab('Shipping','Sales','Finance','Payroll');

  -- This nested table is not initialized ("atomically null").
  empty_set dnames_tab;
BEGIN
  -- At first, the initialized variable is not null.
  IF dept_names IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE('OK, at first dept_names is not null.');
  END IF;
```

```
  -- Then assign a null nested table to it.
  dept_names := empty_set;

  -- Now it is null.
  IF dept_names IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('OK, now dept_names has become null.');
  END IF;

  -- Use another constructor to give it some values.
  dept_names := dnames_tab('Shipping','Sales','Finance','Payroll');
END;
/
```

Result:

```
OK, at first dept_names is not null.
OK, now dept_names has become null.
```

Example 5–19 shows some ANSI-standard operators that you can apply to nested tables.

***Example 5–19 Assigning Nested Tables with Set Operators***

```
DECLARE
  TYPE nested_typ IS TABLE OF NUMBER;
  nt1 nested_typ := nested_typ(1,2,3);
  nt2 nested_typ := nested_typ(3,2,1);
  nt3 nested_typ := nested_typ(2,3,1,3);
  nt4 nested_typ := nested_typ(1,2,4);
  answer nested_typ;

  -- The results might be in a different order than you expect.
  -- Do not rely on the order of elements in nested tables.

  PROCEDURE print_nested_table(the_nt nested_typ) IS
    output VARCHAR2(128);
  BEGIN
    IF the_nt IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Result: <NULL>');
      RETURN;
    END IF;
    IF the_nt.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE('Result: empty set');
      RETURN;
    END IF;
    FOR i IN the_nt.FIRST .. the_nt.LAST
    LOOP
      output := output || the_nt(i) || ' ';
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Result: ' || output);
  END;
BEGIN
  answer := nt1 MULTISET UNION nt4; -- (1,2,3,1,2,4)
  print_nested_table(answer);
  answer := nt1 MULTISET UNION nt3; -- (1,2,3,2,3,1,3)
  print_nested_table(answer);
  answer := nt1 MULTISET UNION DISTINCT nt3; -- (1,2,3)
  print_nested_table(answer);
  answer := nt2 MULTISET INTERSECT nt3; -- (3,2,1)
  print_nested_table(answer);
  answer := nt2 MULTISET INTERSECT DISTINCT nt3; -- (3,2,1)
```

```
  print_nested_table(answer);
  answer := SET(nt3); -- (2,3,1)
  print_nested_table(answer);
  answer := nt3 MULTISET EXCEPT nt2; -- (3)
  print_nested_table(answer);
  answer := nt3 MULTISET EXCEPT DISTINCT nt2; -- ()
  print_nested_table(answer);
END;
/
```

Result:

```
Result: 1 2 3 1 2 4
Result: 1 2 3 2 3 1 3
Result: 1 2 3
Result: 3 2 1
Result: 3 2 1
Result: 2 3 1
Result: 3
Result: empty set
```

Example 5–20 shows an assignment to a VARRAY of records with an assignment statement.

***Example 5–20 Assigning Values to VARRAYs with Complex Data Types***

```
DECLARE
  TYPE emp_name_rec is RECORD (
    firstname    employees.first_name%TYPE,
    lastname     employees.last_name%TYPE,
    hiredate     employees.hire_date%TYPE
  );

  -- Array type that can hold information 10 employees
  TYPE EmpList_arr IS VARRAY(10) OF emp_name_rec;
  SeniorSalespeople EmpList_arr;

  -- Declare a cursor to select a subset of columns.
  CURSOR c1 IS
    SELECT first_name, last_name, hire_date FROM employees;
  Type NameSet IS TABLE OF c1%ROWTYPE;
  SeniorTen NameSet;
  EndCounter NUMBER := 10;

BEGIN
  SeniorSalespeople := EmpList_arr();
  SELECT first_name, last_name, hire_date
    BULK COLLECT INTO SeniorTen
    FROM employees
    WHERE job_id = 'SA_REP'
    ORDER BY hire_date;
  IF SeniorTen.LAST > 0 THEN
    IF SeniorTen.LAST < 10 THEN EndCounter := SeniorTen.LAST;
    END IF;
    FOR i in 1..EndCounter LOOP
      SeniorSalespeople.EXTEND(1);
      SeniorSalespeople(i) := SeniorTen(i);
      DBMS_OUTPUT.PUT_LINE(SeniorSalespeople(i).lastname || ', '
       || SeniorSalespeople(i).firstname || ', ' ||
       SeniorSalespeople(i).hiredate);
    END LOOP;
```

```
  END IF;
END;
/
```

Result:

```
King, Janette, 30-JAN-04
Sully, Patrick, 04-MAR-04
Abel, Ellen, 11-MAY-04
McEwen, Allan, 01-AUG-04
Tucker, Peter, 30-JAN-05
Smith, Lindsey, 10-MAR-05
Ozer, Lisa, 11-MAR-05
Hutton, Alyssa, 19-MAR-05
Bernstein, David, 24-MAR-05
Hall, Peter, 20-AUG-05
```

Example 5–21 shows an assignment to a nested table of records with a `FETCH` statement.

***Example 5–21 Assigning Values to Tables with Complex Data Types***

```
DECLARE
  TYPE emp_name_rec is RECORD (
    firstname    employees.first_name%TYPE,
    lastname     employees.last_name%TYPE,
    hiredate     employees.hire_date%TYPE
  );

  -- Table type that can hold information about employees
  TYPE EmpList_tab IS TABLE OF emp_name_rec;
  SeniorSalespeople EmpList_tab;

  -- Declare a cursor to select a subset of columns.
  CURSOR c1 IS
    SELECT first_name, last_name, hire_date FROM employees;
  EndCounter NUMBER := 10;
  TYPE EmpCurTyp IS REF CURSOR;
  emp_cv EmpCurTyp;

BEGIN
  OPEN emp_cv FOR SELECT first_name, last_name, hire_date
   FROM employees
   WHERE job_id = 'SA_REP' ORDER BY hire_date;

  FETCH emp_cv BULK COLLECT INTO SeniorSalespeople;
  CLOSE emp_cv;

  -- for this example, display a maximum of ten employees
  IF SeniorSalespeople.LAST > 0 THEN
    IF SeniorSalespeople.LAST < 10 THEN
      EndCounter := SeniorSalespeople.LAST;
    END IF;
    FOR i in 1..EndCounter LOOP
      DBMS_OUTPUT.PUT_LINE (
        SeniorSalespeople(i).lastname || ', '  ||
        SeniorSalespeople(i).firstname || ', ' ||
        SeniorSalespeople(i).hiredate);
    END LOOP;
  END IF;
END;
```

```
/
```

Result:

```
King, Janette, 30-JAN-04
Sully, Patrick, 04-MAR-04
Abel, Ellen, 11-MAY-04
McEwen, Allan, 01-AUG-04
Tucker, Peter, 30-JAN-05
Smith, Lindsey, 10-MAR-05
Ozer, Lisa, 11-MAR-05
Hutton, Alyssa, 19-MAR-05
Bernstein, David, 24-MAR-05
Hall, Peter, 20-AUG-05
```

## Collection Comparisons

You can check whether a collection is null. Comparisons such as greater than, less than, and so on are not allowed. This restriction also applies to implicit comparisons. For example, collections cannot appear in a `DISTINCT`, `GROUP BY`, or `ORDER BY` list.

If you want to do such comparison operations, you must define your own notion of what it means for collections to be greater than, less than, and so on, and write one or more functions to examine the collections and their elements and return a true or false value.

For nested tables, you can check whether two nested table of the same declared type are equal or not equal, as shown in Example 5–23. You can also apply set operators to check certain conditions in a nested table or between two nested tables, as shown in Example 5–24.

Because nested tables and varrays can be atomically null, they can be tested for nullity, as shown in Example 5–22.

***Example 5–22 Checking if a Collection Is Null***

```
DECLARE
  TYPE emp_name_rec is RECORD (
    firstname    employees.first_name%TYPE,
    lastname     employees.last_name%TYPE,
    hiredate     employees.hire_date%TYPE
    );
   TYPE staff IS TABLE OF emp_name_rec;
   members staff;
BEGIN
  -- Condition yields TRUE because you have not used a constructor.
   IF members IS NULL THEN
     DBMS_OUTPUT.PUT_LINE('NULL');
   ELSE
     DBMS_OUTPUT.PUT_LINE('Not NULL');
   END IF;
END;
/
```

Result:

```
NULL
```

Example 5–23 shows that nested tables can be compared for equality or inequality. They cannot be ordered, because there is no greater than or less than comparison.

***Example 5–23 Comparing Two Nested Tables***

```
DECLARE
   TYPE dnames_tab IS TABLE OF VARCHAR2(30);
   dept_names1 dnames_tab :=
     dnames_tab('Shipping','Sales','Finance','Payroll');
   dept_names2 dnames_tab :=
     dnames_tab('Sales','Finance','Shipping','Payroll');
   dept_names3 dnames_tab :=
     dnames_tab('Sales','Finance','Payroll');
BEGIN
-- You can use = or !=, but not < or >.
-- These 2 are equal even though members are in different order.
   IF dept_names1 = dept_names2 THEN
     DBMS_OUTPUT.PUT_LINE
      ('dept_names1 and dept_names2 have the same members.');
   END IF;
   IF dept_names2 != dept_names3 THEN
      DBMS_OUTPUT.PUT_LINE
        ('dept_names2 and dept_names3 have different members.');
   END IF;
END;
/
```

Result:

```
dept_names1 and dept_names2 have the same members.
dept_names2 and dept_names3 have different members.
```

You can test certain properties of a nested table, or compare two nested tables, using ANSI-standard set operations, as shown in Example 5–24.

***Example 5–24 Comparing Nested Tables with Set Operators***

```
DECLARE
  TYPE nested_typ IS TABLE OF NUMBER;
  nt1 nested_typ := nested_typ(1,2,3);
  nt2 nested_typ := nested_typ(3,2,1);
  nt3 nested_typ := nested_typ(2,3,1,3);
  nt4 nested_typ := nested_typ(1,2,4);
  answer BOOLEAN;
  howmany NUMBER;
  PROCEDURE testify
    (truth BOOLEAN := NULL,
     quantity NUMBER := NULL) IS
  BEGIN
    IF truth IS NOT NULL THEN
      DBMS_OUTPUT.PUT_LINE
        (CASE truth WHEN TRUE THEN 'True' WHEN FALSE THEN 'False' END);
    END IF;
    IF quantity IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE(quantity);
    END IF;
  END;
BEGIN
  answer := nt1 IN (nt2,nt3,nt4); -- true, nt1 matches nt2
  testify(truth => answer);
  answer := nt1 SUBMULTISET OF nt3; -- true, all elements match
  testify(truth => answer);
  answer := nt1 NOT SUBMULTISET OF nt4; -- also true
  testify(truth => answer);
```

```
  howmany := CARDINALITY(nt3); -- number of elements in nt3
  testify(quantity => howmany);
  howmany := CARDINALITY(SET(nt3)); -- number of distinct elements
  testify(quantity => howmany);
  answer := 4 MEMBER OF nt1; -- false, no element matches
  testify(truth => answer);
  answer := nt3 IS A SET; -- false, nt3 has duplicates
  testify(truth => answer);
  answer := nt3 IS NOT A SET; -- true, nt3 has duplicates
  testify(truth => answer);
  answer := nt1 IS EMPTY; -- false, nt1 has some members
  testify(truth => answer);
END;
/
```

Result:

```
True
True
True
4
3
False
False
True
False
```

## Multidimensional Collections

Although a collection has only one dimension, you can model a multidimensional collection by creating a collection whose elements are also collections. For example, you can create a nested table of varrays, a varray of varrays, a varray of nested tables, and so on.

When creating a nested table of nested tables as a column in SQL, check the syntax of the CREATE TABLE statement to see how to define the storage table.

Example 5–25, Example 5–26, and Example 5–27 are some examples showing the syntax and possibilities for multilevel collections.

***Example 5–25 Multilevel VARRAY***

```
DECLARE
  TYPE t1 IS VARRAY(10) OF INTEGER;
  TYPE nt1 IS VARRAY(10) OF t1; -- multilevel varray type
  va t1 := t1(2,3,5);
  -- initialize multilevel varray
  nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);
  i INTEGER;
  va1 t1;
BEGIN
  -- multilevel access
  i := nva(2)(3); -- i will get value 73
  DBMS_OUTPUT.PUT_LINE('I = ' || i);
  -- add a varray element to nva
  nva.EXTEND;
  -- replace inner varray elements
  nva(5) := t1(56, 32);
  nva(4) := t1(45,43,67,43345);
  -- replace an inner integer element
```

```
  nva(4)(4) := 1; -- replaces 43345 with 1
  -- add an element to the 4th varray element
  -- and store integer 89 into it.
  nva(4).EXTEND;
  nva(4)(5) := 89;
END;
/
```

Result:

```
I = 73
```

***Example 5–26 Multilevel Nested Table***

```
DECLARE
  TYPE tb1 IS TABLE OF VARCHAR2(20);
  TYPE Ntb1 IS TABLE OF tb1; -- table of table elements
  TYPE Tv1 IS VARRAY(10) OF INTEGER;
  TYPE ntb2 IS TABLE OF tv1; -- table of varray elements
  vtb1 tb1 := tb1('one', 'three');
  vntb1 ntb1 := ntb1(vtb1);
  vntb2 ntb2 := ntb2(tv1(3,5), tv1(5,7,3));
  -- table of varray elements
BEGIN
  vntb1.EXTEND;
  vntb1(2) := vntb1(1);
  -- delete the first element in vntb1
  vntb1.DELETE(1);
  -- delete the first string
  -- from the second table in the nested table
  vntb1(2).DELETE(1);
END;
/
```

***Example 5–27 Multilevel Associative Array***

```
DECLARE
  TYPE tb1 IS TABLE OF INTEGER INDEX BY PLS_INTEGER;
  -- Index-by table of index-by tables:
  TYPE ntb1 IS TABLE OF tb1 INDEX BY PLS_INTEGER;
  TYPE va1 IS VARRAY(10) OF VARCHAR2(20);
  -- Index-by table of varray elements:
  TYPE ntb2 IS TABLE OF va1 INDEX BY PLS_INTEGER;
  v1 va1 := va1('hello', 'world');
  v2 ntb1;
  v3 ntb2;
  v4 tb1;
  v5 tb1; -- empty table
BEGIN
  v4(1) := 34;
  v4(2) := 46456;
  v4(456) := 343;
  v2(23) := v4;
  v3(34) := va1(33, 456, 656, 343);
  -- assign an empty table to v2(35) and try again
  v2(35) := v5;
  v2(35)(2) := 78; -- it works now
END;
/
```

# Collection Methods

A collection method is a built-in PL/SQL subprogram—either a function that returns information about a collection or a procedure that operates on a collection. Collection methods make collections easier to use, and make your applications easier to maintain. Table 5–2 summarizes the collection methods.

***Table 5–2 Collection Methods***

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

The basic syntax of a collection method call is:

```
collection_name.method
```

For detailed syntax, see "Collection Method Call" on page 13-27.

A collection method call can appear anywhere that a call to a PL/SQL subprogram of its type (function or procedure) can appear, except in a SQL statement.

The only collection method that you can use with an empty collection is EXISTS; all others raise the exception COLLECTION_IS_NULL.

In a subprogram, a collection parameter assumes the properties of the argument bound to it. You can apply collection methods to such parameters. For varray parameters, the value of LIMIT is always derived from the parameter type definition, regardless of the parameter mode.

Topics:

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

## EXISTS Method

The EXISTS method is a function that you can use to check if a specified element of a nested table or varray exists.

EXISTS(*n*) returns TRUE if the *n*th element of the collection exists and FALSE otherwise. If *n* is out of range, EXISTS returns FALSE instead of raising SUBSCRIPT_OUTSIDE_LIMIT.

Use EXISTS to avoid referencing a nonexistent element, which raises an exception.

***Example 5–28 Checking Whether a Collection Element EXISTS***

```
DECLARE
   TYPE NumList IS TABLE OF INTEGER;
   n NumList := NumList(1,3,5,7);
BEGIN
   n.DELETE(2); -- Delete the second element
   IF n.EXISTS(1) THEN
      DBMS_OUTPUT.PUT_LINE('OK, element #1 exists.');
   END IF;
   IF n.EXISTS(2) = FALSE THEN
      DBMS_OUTPUT.PUT_LINE('OK, element #2 was deleted.');
   END IF;
   IF n.EXISTS(99) = FALSE THEN
      DBMS_OUTPUT.PUT_LINE('OK, element #99 does not exist at all.');
   END IF;
END;
/
```

Result:

```
OK, element #1 exists.
OK, element #2 was deleted.
OK, element #99 does not exist at all.
```

## COUNT Method

The COUNT method is a function that returns the number of elements in the collection (ignoring deleted elements).

For a varray, COUNT always equals LAST. If you increase or decrease the size of a varray (using the EXTEND or TRIM method), the value of COUNT changes.

For a nested table, COUNT equals LAST unless you delete elements from its middle, in which case COUNT is smaller than LAST.

Using DELETE with no parameters sets COUNT to 0.

***Example 5–29 Counting Collection Elements with COUNT***

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n NumList := NumList(2,4,6,8);
     -- Collection starts with 4 elements.
BEGIN
   DBMS_OUTPUT.PUT_LINE
     ('There are ' || n.COUNT || ' elements in N.');
   n.EXTEND(3); -- Add 3 elements at the end.
   DBMS_OUTPUT.PUT_LINE
     ('Now there are ' || n.COUNT || ' elements in N.');
   n := NumList(86,99); -- Assign a value with 2 elements.
```

```
   DBMS_OUTPUT.PUT_LINE
     ('Now there are ' || n.COUNT || ' elements in N.');
   n.TRIM(2); -- Remove the last 2 elements, leaving none.
   DBMS_OUTPUT.PUT_LINE
     ('Now there are ' || n.COUNT || ' elements in N.');
END;
/
```

Result:

```
There are 4 elements in N.
Now there are 7 elements in N.
Now there are 2 elements in N.
Now there are 0 elements in N.
```

## LIMIT Method

The LIMIT method is a function that returns the maximum number of elements that the collection can have. If the collection has no maximum size, LIMIT returns NULL.

***Example 5–30 Checking the Maximum Size of a Collection with LIMIT***

```
DECLARE
   TYPE dnames_var IS VARRAY(7) OF VARCHAR2(30);
   dept_names dnames_var :=
     dnames_var('Shipping','Sales','Finance','Payroll');
BEGIN
   DBMS_OUTPUT.PUT_LINE
     ('dept_names has ' || dept_names.COUNT || ' elements now');
   DBMS_OUTPUT.PUT_LINE
     ('dept_names''s type can hold a maximum of '
      || dept_names.LIMIT || ' elements');
   DBMS_OUTPUT.PUT_LINE
    ('The maximum number you can use with '
     || 'dept_names.EXTEND() is '
     || (dept_names.LIMIT - dept_names.COUNT));
END;
/
```

Result:

```
dept_names has 4 elements now
dept_names's type can hold a maximum of 7 elements
The maximum number you can use with dept_names.EXTEND() is 3
```

## FIRST and LAST Methods

The FIRST and LAST methods are functions.

For a collection indexed by integers, FIRST and LAST return the first and last (smallest and largest) index numbers (ignoring deleted elements).

For an associative array indexed by strings, FIRST and LAST return the lowest and highest key values. If the NLS_COMP initialization parameter is set to ANSI, the order is based on the sort order specified by the NLS_SORT initialization parameter.

If the collection is empty, FIRST and LAST return NULL. If the collection contains only one element, FIRST and LAST return the same value. If the collection is a varray that is not empty, FIRST always returns 1.

For a varray, LAST always equals COUNT. For a nested table, LAST equals COUNT unless you delete elements from its middle, in which case LAST is larger than COUNT.

Example 5–31 uses FIRST and LAST to traverse a collection that has consecutive subscripts.

***Example 5–31 FIRST and LAST Methods***

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n NumList := NumList(1,3,5,7);
   counter INTEGER;
BEGIN
   DBMS_OUTPUT.PUT_LINE('N''s first subscript is ' || n.FIRST);
   DBMS_OUTPUT.PUT_LINE('N''s last subscript is ' || n.LAST);
-- When the subscripts are consecutive starting at 1,
-- it's simple to loop through them.
   FOR i IN n.FIRST .. n.LAST
   LOOP
      DBMS_OUTPUT.PUT_LINE('Element #' || i || ' = ' || n(i));
   END LOOP;
   n.DELETE(2); -- Delete second element.
-- When the subscripts have gaps
-- or the collection might be uninitialized,
-- the loop logic is more extensive.
-- Start at the first element
-- and look for the next element until there are no more.
   IF n IS NOT NULL THEN
      counter := n.FIRST;
      WHILE counter IS NOT NULL
      LOOP
         DBMS_OUTPUT.PUT_LINE
           ('Element #' || counter || ' = ' || n(counter));
         counter := n.NEXT(counter);
      END LOOP;
   ELSE
      DBMS_OUTPUT.PUT_LINE('N is null, nothing to do.');
   END IF;
END;
/
```

Result:

```
N's first subscript is 1
N's last subscript is 4
Element #1 = 1
Element #2 = 3
Element #3 = 5
Element #4 = 7
Element #1 = 1
Element #3 = 5
Element #4 = 7
```

## PRIOR and NEXT Methods

The PRIOR and NEXT methods are functions that let you move backward and forward in the collection.

PRIOR(*n*) returns the index number that precedes index *n* in the collection. If *n* has no predecessor, PRIOR(*n*) returns NULL.

`NEXT(n)` returns the index number that succeeds index *n* in the collection. If *n* has no successor, `NEXT(n)` returns `NULL`.

For an associative array with `VARCHAR2` keys, these methods return the appropriate key values. Ordering is based on the binary values of the characters in the string, unless the `NLS_COMP` initialization parameter is set to `ANSI`, in which case the ordering is based on the locale-specific sort order specified by the `NLS_SORT` initialization parameter.

These methods are more reliable than looping through a fixed set of subscript values, because elements might be inserted or deleted from the collection during the loop. This is especially true for associative arrays, where subscripts might not be in consecutive order.

***Example 5–32 PRIOR and NEXT Methods***

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n NumList := NumList(1966,1971,1984,1989,1999);
BEGIN
   DBMS_OUTPUT.PUT_LINE('The element after #2 is #' || n.NEXT(2));
   DBMS_OUTPUT.PUT_LINE('The element before #2 is #' || n.PRIOR(2));
   n.DELETE(3);
     -- Delete an element to show how NEXT can handle gaps.
   DBMS_OUTPUT.PUT_LINE
     ('Now the element after #2 is #' || n.NEXT(2));
   IF n.PRIOR(n.FIRST) IS NULL THEN
      DBMS_OUTPUT.PUT_LINE
        ('Can''t get PRIOR of the first element or NEXT of the last.');
   END IF;
END;
/
```

Result:

```
The element after #2 is #3
The element before #2 is #1
Now the element after #2 is #4
Can't get PRIOR of the first element or NEXT of the last.
```

You can use `PRIOR` or `NEXT` to traverse collections indexed by any series of subscripts. Example 5–33 uses `NEXT` to traverse a nested table from which some elements were deleted.

***Example 5–33 NEXT Method Accesses Elements of Nested Table***

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n NumList := NumList(1,3,5,7);
   counter INTEGER;
BEGIN
   n.DELETE(2); -- Delete second element.
-- When the subscripts have gaps,
-- loop logic is more extensive.
-- Start at first element and look for next element
-- until there are no more.
   counter := n.FIRST;
   WHILE counter IS NOT NULL
   LOOP
      DBMS_OUTPUT.PUT_LINE
        ('Counting up: Element #' || counter || ' = ' || n(counter));
```

```
      counter := n.NEXT(counter);
   END LOOP;
-- Run the same loop in reverse order.
   counter := n.LAST;
   WHILE counter IS NOT NULL
   LOOP
      DBMS_OUTPUT.PUT_LINE
        ('Counting down: Element #' || counter || ' = ' || n(counter));
      counter := n.PRIOR(counter);
   END LOOP;
END;
/
```

Result:

```
Counting up: Element #1 = 1
Counting up: Element #3 = 5
Counting up: Element #4 = 7
Counting down: Element #4 = 7
Counting down: Element #3 = 5
Counting down: Element #1 = 1
```

When traversing elements, PRIOR and NEXT skip over deleted elements.

## EXTEND Method

The EXTEND method is a procedure that increases the size of a nested table or varray. This method has these forms:

- EXTEND appends one null element to a collection.
- EXTEND(*n*) appends *n* null elements to a collection.
- EXTEND(*n*,*i*) appends *n* copies of the *i*th element to a collection.

> **Note:** This is the only form of EXTEND that you can use on a collection whose elements have the NOT NULL constraint.

You cannot use EXTEND to initialize an atomically null collection. For instructions for initializing a collection, see "Collection Initialization" on page 5-10.

EXTEND operates on the internal size of a collection, which includes elements deleted with DELETE(*n*) but not elements deleted with DELETE (see "DELETE Method" on page 5-30).

***Example 5–34 EXTEND Method***

```
DECLARE
   TYPE NumList IS TABLE OF INTEGER;
   n NumList := NumList(2,4,6,8);
   x NumList := NumList(1,3);
   PROCEDURE print_numlist(the_list NumList) IS
      output VARCHAR2(128);
   BEGIN
      FOR i IN the_list.FIRST .. the_list.LAST
      LOOP
         output :=
           output || NVL(TO_CHAR(the_list(i)),'NULL') || ' ';
      END LOOP;
      DBMS_OUTPUT.PUT_LINE(output);
```

```
   END;
BEGIN
   DBMS_OUTPUT.PUT_LINE
     ('At first, N has ' || n.COUNT || ' elements.');
   n.EXTEND(5); -- Add 5 elements at the end.
   DBMS_OUTPUT.PUT_LINE
     ('Now N has ' || n.COUNT || ' elements.');
-- Elements 5, 6, 7, 8, and 9 are all NULL.
   print_numlist(n);
   DBMS_OUTPUT.PUT_LINE
     ('At first, X has ' || x.COUNT || ' elements.');
   x.EXTEND(4,2); -- Add 4 elements at the end.
   DBMS_OUTPUT.PUT_LINE
     ('Now X has ' || x.COUNT || ' elements.');
-- Elements 3, 4, 5, and 6 are copies of element #2.
   print_numlist(x);
END;
/
```

Result:

```
At first, N has 4 elements.
Now N has 9 elements.
2 4 6 8 NULL NULL NULL NULL NULL
At first, X has 2 elements.
Now X has 6 elements.
1 3 3 3 3 3
```

When it includes deleted elements, the internal size of a nested table differs from the values returned by `COUNT` and `LAST`. This refers to deleted elements after using `DELETE(`*n*`)`, but not `DELETE` without parameters which completely removes all elements. For example, if you initialize a nested table with five elements, then delete elements 2 and 5, the internal size is 5, `COUNT` returns 3, and `LAST` returns 4. All deleted elements, regardless of position, are treated alike.

## TRIM Method

The `TRIM` method is a procedure that deletes elements from the end of a nested table or varray. This method has these forms:

- `TRIM` removes one element from the end of a collection.
- `TRIM(`*n*`)` removes *n* elements from the end of a collection.

`TRIM(`*n*`)` removes *n* elements from the end of a collection. If *n* is greater than `COUNT`, `TRIM(`*n*`)` raises `SUBSCRIPT_BEYOND_COUNT`. `TRIM` operates on the internal size of a collection. If `TRIM` encounters deleted elements, it includes them in its tally.

If *n* is too large, `TRIM(`*n*`)` raises `SUBSCRIPT_BEYOND_COUNT`.

To remove all elements, use `DELETE` without parameters. See "DELETE Method" on page 5-30.

---

**Caution:** Do not depend on interaction between `TRIM` and `DELETE`. Either treat nested tables like fixed-size arrays and use only `DELETE`, or to treat them like stacks and use only `TRIM` and `EXTEND`.

---

In Example 5–35, the `TRIM` statement removes the last three elements from nested table `courses`.

***Example 5–35 TRIM Method***

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n NumList := NumList(1,2,3,5,7,11);
   PROCEDURE print_numlist(the_list NumList) IS
      output VARCHAR2(128);
   BEGIN
      IF n.COUNT = 0 THEN
         DBMS_OUTPUT.PUT_LINE('No elements in collection.');
      ELSE
         FOR i IN the_list.FIRST .. the_list.LAST
         LOOP
            output :=
              output || NVL(TO_CHAR(the_list(i)),'NULL') || ' ';
         END LOOP;
         DBMS_OUTPUT.PUT_LINE(output);
      END IF;
   END;
BEGIN
   print_numlist(n);
   n.TRIM(2); -- Remove last 2 elements.
   print_numlist(n);
   n.TRIM; -- Remove last element.
   print_numlist(n);
   n.TRIM(n.COUNT); -- Remove all remaining elements.
   print_numlist(n);
-- If too many elements are specified,
-- TRIM raises the exception SUBSCRIPT_BEYOND_COUNT.
   BEGIN
      n := NumList(1,2,3);
      n.TRIM(100);
      EXCEPTION
        WHEN SUBSCRIPT_BEYOND_COUNT THEN
          DBMS_OUTPUT.PUT_LINE
            ('There weren''t 100 elements to be trimmed.');
   END;
-- When elements are removed by DELETE,
-- placeholders are left behind.
--  TRIM counts these placeholders
--  as it removes elements from the end.
   n := NumList(1,2,3,4);
   n.DELETE(3);  -- delete element 3
-- At this point, n contains elements (1,2,4).
-- TRIMming the last 2 elements
-- removes the 4 and the placeholder, not 4 and 2.
   n.TRIM(2);
   print_numlist(n);
END;
/
```

Result:

```
1 2 3 5 7 11
1 2 3 5
1 2 3
No elements in collection.
There weren't 100 elements to be trimmed.
1 2
```

TRIM operates on the internal size of a collection. If TRIM encounters deleted elements, it includes them in its tally. This refers to deleted elements after using DELETE(*n*), but not DELETE without parameters which completely removes all elements.

***Example 5–36 TRIM Method on Deleted Elements***

```
DECLARE
  TYPE CourseList IS TABLE OF VARCHAR2(10);
  courses CourseList;
BEGIN
  courses := CourseList('Biol 4412', 'Psyc 3112', 'Anth 3001');
  courses.DELETE(courses.LAST);  -- delete element 3
  /* At this point, COUNT equals 2, the number of valid
     elements remaining. So, you might expect the next
     statement to empty the nested table by trimming
     elements 1 and 2. Instead, it trims valid element 2
     and deleted element 3 because TRIM includes deleted
     elements in its tally. */
  courses.TRIM(courses.COUNT);
  DBMS_OUTPUT.PUT_LINE(courses(1));  -- prints 'Biol 4412'
END;
/
```

Result:

```
Biol 4412
```

In general, do not depend on the interaction between TRIM and DELETE. It is better to treat nested tables like fixed-size arrays and use only DELETE, or to treat them like stacks and use only TRIM and EXTEND.

Because PL/SQL does not keep placeholders for trimmed elements, you cannot replace a trimmed element simply by assigning it a value.

## DELETE Method

The DELETE method is a procedure that deletes elements from the collection. This method has these forms:

- DELETE deletes all elements from the collection.

> **Note:** This is the only form of DELETE that you can use with a varray (which always has consecutive subscripts).

- DELETE(*n*) deletes the *n*th element from an associative array or nested table. If the *n*th element is null, DELETE(*n*) does nothing.
- DELETE(*m*,*n*) deletes all elements in the range *m*..*n* from an associative array or nested table. If *m* is larger than *n* or if *m* or *n* is null, DELETE(*m*,*n*) does nothing.

If an element to be deleted does not exist, DELETE skips it and no exception is raised.

PL/SQL keeps placeholders for deleted elements, so that you can re-create them by assigning values.

***Example 5–37 DELETE Method***

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n NumList := NumList(10,20,30,40,50,60,70,80,90,100);
```

```
   TYPE NickList IS TABLE OF VARCHAR2(64) INDEX BY VARCHAR2(32);
   nicknames NickList;
BEGIN
   n.DELETE(2);    -- deletes element 2
   n.DELETE(3,6);  -- deletes elements 3 through 6
   n.DELETE(7,7);  -- deletes element 7
   n.DELETE(6,3);  -- does nothing since 6 > 3
   n.DELETE;      -- deletes all elements
   nicknames('Bob') := 'Robert';
   nicknames('Buffy') := 'Esmerelda';
   nicknames('Chip') := 'Charles';
   nicknames('Dan') := 'Daniel';
   nicknames('Fluffy') := 'Ernestina';
   nicknames('Rob') := 'Robert';
-- following deletes element denoted by this key
   nicknames.DELETE('Chip');
-- following deletes elements with keys in this alphabetic range
   nicknames.DELETE('Buffy','Fluffy');
END;
/
```

PL/SQL keeps placeholders for deleted elements, so you can replace a deleted element by assigning it a value. This refers to deleted elements after using `DELETE(n)`, but not `DELETE` without parameters which completely removes all elements.

`DELETE` lets you maintain sparse nested tables. You can store sparse nested tables in the database, just like any other nested tables.

The amount of memory allocated to a collection increases as the number of elements in the collection increases. If you delete the entire collection, or delete all elements individually, all of the memory used to store elements of that collection is freed.

## Collection Exceptions

Example 5–38 shows various collection exceptions that are predefined in PL/SQL. The example also includes notes on how to avoid the problems.

***Example 5–38 Collection Exceptions***

```
DECLARE
  TYPE WordList IS TABLE OF VARCHAR2(5);
  words WordList;
  err_msg VARCHAR2(100);
  PROCEDURE display_error IS
  BEGIN
    err_msg := SUBSTR(SQLERRM, 1, 100);
    DBMS_OUTPUT.PUT_LINE('Error message = ' || err_msg);
  END;
BEGIN
  BEGIN
    words(1) := 10; -- Raises COLLECTION_IS_NULL
--  A constructor has not been used yet.
--  Note: This exception applies to varrays and nested tables,
--  but not to associative arrays which do not need a constructor.
    EXCEPTION
      WHEN OTHERS THEN display_error;
  END;
--  After using a constructor, you can assign values to the elements.
    words := WordList('1st', '2nd', '3rd'); -- 3 elements created
--  Any expression that returns a VARCHAR2(5) is valid.
```

```
    words(3) := words(1) || '+2';
  BEGIN
    words(3) := 'longer than 5 characters'; -- Raises VALUE_ERROR
--  The assigned value is too long.
    EXCEPTION
      WHEN OTHERS THEN display_error;
  END;
  BEGIN
    words('B') := 'dunno'; -- Raises VALUE_ERROR
--  The subscript (B) of a nested table must be an integer.
--  Note: Also, NULL is not allowed as a subscript.
    EXCEPTION
      WHEN OTHERS THEN display_error;
  END;
  BEGIN
    words(0) := 'zero'; -- Raises SUBSCRIPT_OUTSIDE_LIMIT
--  Subscript 0 is outside the allowed subscript range.
    EXCEPTION
      WHEN OTHERS THEN display_error;
  END;
  BEGIN
    words(4) := 'maybe'; -- Raises SUBSCRIPT_BEYOND_COUNT
--  The subscript (4) exceeds the number of elements in the table.
--  To add elements, invoke the EXTEND method first.
    EXCEPTION
      WHEN OTHERS THEN display_error;
  END;
  BEGIN
    words.DELETE(1);
    IF words(1) = 'First' THEN NULL; END IF;
      -- Raises NO_DATA_FOUND
--  The element with subcript (1) was deleted.
    EXCEPTION
      WHEN OTHERS THEN display_error;
  END;
END;
/
```

Result:

```
Error message = ORA-06531: Reference to uninitialized collection
Error message = ORA-06502: PL/SQL: numeric or value error: character string
buffer too small
Error message = ORA-06502: PL/SQL: numeric or value error: character to number
conversion error
Error message = ORA-06532: Subscript outside of limit
Error message = ORA-06533: Subscript beyond count
Error message = ORA-01403: no data found
```

Execution continues in Example 5–38 because the raised exceptions are handled in sub-blocks. See "Continuing Execution After an Exception Is Raised" on page 11-19. For information about the use of SQLERRM with exception handling, see "Error Code and Error Message Retrieval" on page 11-18.

This table summarizes when a given exception is raised:

| Collection Exception | Raised when... |
|---|---|
| COLLECTION_IS_NULL | you try to operate on an atomically null collection. |

| Collection Exception | Raised when... |
| --- | --- |
| NO_DATA_FOUND | a subscript designates an element that was deleted, or a nonexistent element of an associative array. |
| SUBSCRIPT_BEYOND_COUNT | a subscript exceeds the number of elements in a collection. |
| SUBSCRIPT_OUTSIDE_LIMIT | a subscript is outside the allowed range. |
| VALUE_ERROR | a subscript is null or not convertible to the key type. This exception might occur if the key is defined as a PLS_INTEGER range, and the subscript is outside this range. |

**See Also:** "Predefined Exceptions" on page 11-7

In some cases, you can pass invalid subscripts to a method without raising an exception. For example, when you pass a null subscript to `DELETE(n)`, it does nothing. You can replace deleted elements by assigning values to them, without raising `NO_DATA_FOUND`. This refers to deleted elements after using `DELETE(n)`, but not `DELETE` without parameters which completely removes all elements.

***Example 5–39 How Invalid Subscripts are Handled with DELETE(n)***

```
DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   nums NumList := NumList(10,20,30);  -- initialize table
BEGIN
   nums.DELETE(-1);  -- does not raise SUBSCRIPT_OUTSIDE_LIMIT
   nums.DELETE(3);   -- delete 3rd element
   DBMS_OUTPUT.PUT_LINE(nums.COUNT);  -- prints 2
   nums(3) := 30;    -- allowed; does not raise NO_DATA_FOUND
   DBMS_OUTPUT.PUT_LINE(nums.COUNT);  -- prints 3
END;
/
```

Result:

```
2
3
```

Packaged collection types and local collection types are never compatible. For example, if you invoke the packaged procedure in Example 5–40, the second procedure call fails, because the packaged and local `VARRAY` types are incompatible despite their identical definitions.

***Example 5–40 Incompatibility Between Package and Local Collection Types***

```
CREATE PACKAGE pkg AS
   TYPE NumList IS TABLE OF NUMBER;
   PROCEDURE print_numlist (nums NumList);
END pkg;
/
CREATE PACKAGE BODY pkg AS
  PROCEDURE print_numlist (nums NumList) IS
  BEGIN
    FOR i IN nums.FIRST..nums.LAST LOOP
      DBMS_OUTPUT.PUT_LINE(nums(i));
    END LOOP;
  END;
END pkg;
```

```
/

DECLARE
   TYPE NumList IS TABLE OF NUMBER;
   n1 pkg.NumList := pkg.NumList(2,4); -- type from the package.
   n2 NumList := NumList(6,8);         -- local type.
BEGIN
   pkg.print_numlist(n1); -- type from pkg is legal
-- The packaged procedure cannot accept
-- a value of the local type (n2)
-- pkg.print_numlist(n2);  -- Causes a compilation error.
END;
/
```

Result:

```
2
4
```

## Record Definitions and Declarations

To create records, you define a RECORD type, then declare records of that type. You can also create or find a table, view, or PL/SQL cursor with the values you want, and use the %ROWTYPE attribute to create a matching record.

You can define RECORD types in the declarative part of any PL/SQL block, subprogram, or package. When you define your own RECORD type, you can specify a NOT NULL constraint on fields, or give them initial values. See "Record" on page 13-106.

Example 5–42 and Example 5–42 illustrate record type declarations.

***Example 5–41 Declaring and Initializing a Simple Record Type***

```
DECLARE
   TYPE DeptRecTyp IS RECORD (
      deptid NUMBER(4) NOT NULL := 99,
      dname  departments.department_name%TYPE,
      loc    departments.location_id%TYPE,
      region regions%ROWTYPE );
   dept_rec DeptRecTyp;
BEGIN
   dept_rec.dname := 'PURCHASING';
END;
/
```

***Example 5–42 Declaring and Initializing Record Types***

```
DECLARE
-- Declare a record type with 3 fields.
  TYPE rec1_t IS RECORD
    (field1 VARCHAR2(16), field2 NUMBER, field3 DATE);
-- For any fields declared NOT NULL, you must supply an initial value.
  TYPE rec2_t IS RECORD (id INTEGER NOT NULL := -1,
  name VARCHAR2(64) NOT NULL := '[anonymous]');
-- Declare record variables of the types declared
  rec1 rec1_t;
  rec2 rec2_t;
-- Declare a record variable that can hold
-- a row from the EMPLOYEES table.
-- The fields of the record automatically match the names and
-- types of the columns.
```

```
-- Don't need a TYPE declaration in this case.
  rec3 employees%ROWTYPE;
-- Or mix fields that are table columns with user-defined fields.
  TYPE rec4_t IS RECORD (first_name employees.first_name%TYPE,
                         last_name employees.last_name%TYPE,
                         rating NUMBER);
  rec4 rec4_t;
BEGIN
-- Read and write fields using dot notation
  rec1.field1 := 'Yesterday';
  rec1.field2 := 65;
  rec1.field3 := TRUNC(SYSDATE-1);
-- Didn't fill name field, so it takes initial value
  DBMS_OUTPUT.PUT_LINE(rec2.name);
END;
/
```

Result:

```
[anonymous]
```

To store a record in the database, you can specify it in an INSERT or UPDATE statement, if its fields match the columns in the table.

You can use %TYPE to specify a field type corresponding to a table column type. Your code keeps working even if the column type is changed (for example, to increase the length of a VARCHAR2 or the precision of a NUMBER). Example 5–43 defines RECORD types to hold information about a department.

***Example 5–43 %ROWTYPE in Record Declaration***

```
DECLARE
-- Best: use %ROWTYPE instead of specifying each column.
-- Use <cursor>%ROWTYPE instead of <table>%ROWTYPE because
-- you only want some columns.
-- Declaring cursor does not run query or affect performance.
   CURSOR c1 IS
     SELECT department_id, department_name, location_id
     FROM departments;
   rec1 c1%ROWTYPE;
-- Use <column>%TYPE in field declarations to avoid problems if
-- the column types change.
   TYPE DeptRec2 IS RECORD
     (dept_id   departments.department_id%TYPE,
      dept_name departments.department_name%TYPE,
      dept_loc  departments.location_id%TYPE);
   rec2 DeptRec2;
-- Write each field name, specifying type directly
-- (clumsy and unmaintainable for working with table data
-- use only for all-PL/SQL code).
   TYPE DeptRec3 IS RECORD (dept_id NUMBER,
                            dept_name VARCHAR2(14),
                            dept_loc VARCHAR2(13));
   rec3 DeptRec3;
BEGIN
   NULL;
END;
/
```

PL/SQL lets you define records that contain objects, collections, and other records (called nested records). However, records cannot be attributes of ADTs.

To declare a record that represents a row in a database table, without listing the columns, use the %ROWTYPE attribute.

Your code keeps working even after columns are added to the table. If you want to represent a subset of columns in a table, or columns from different tables, you can define a view or declare a cursor to select the right columns and do any necessary joins, and then apply %ROWTYPE to the view or cursor.

## Records as Subprogram Parameters and Function Return Values

Records are easy to process using stored subprograms because you can pass just one parameter, instead of a separate parameter for each field. For example, you can fetch a table row from the EMPLOYEES table into a record, and then pass that row as a parameter to a function that computes that employee's vacation allowance. The function can access all the information about that employee by referring to the fields in the record.

Example 5–44 shows how to return a record from a function. To make the record type visible across multiple stored subprograms, declare the record type in a package specification.

***Example 5–44 Returning a Record from a Function***

```
DECLARE
   TYPE EmpRecTyp IS RECORD (
     emp_id       NUMBER(6),
     salary       NUMBER(8,2));
   CURSOR desc_salary RETURN EmpRecTyp IS
      SELECT employee_id, salary
      FROM employees
      ORDER BY salary DESC;
   emp_rec     EmpRecTyp;
   FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp IS
   BEGIN
      OPEN desc_salary;
      FOR i IN 1..n LOOP
         FETCH desc_salary INTO emp_rec;
      END LOOP;
      CLOSE desc_salary;
      RETURN emp_rec;
   END nth_highest_salary;
BEGIN
   NULL;
END;
/
```

Like scalar variables, user-defined records can be declared as the formal parameters of subprograms, as in Example 5–45.

***Example 5–45 Record as Procedure Parameter***

```
DECLARE
   TYPE EmpRecTyp IS RECORD (
      emp_id       NUMBER(6),
      emp_sal      NUMBER(8,2) );
   PROCEDURE raise_salary (emp_info EmpRecTyp) IS
   BEGIN
      UPDATE employees SET salary = salary + salary * .10
             WHERE employee_id = emp_info.emp_id;
```

```
   END raise_salary;
BEGIN
   NULL;
END;
/
```

You can declare and reference nested records. That is, a record can be the component of another record.

***Example 5–46 Nested Record Declaration***

```
DECLARE
   TYPE TimeTyp IS RECORD ( minutes SMALLINT, hours SMALLINT );
   TYPE MeetingTyp IS RECORD (
      day     DATE,
      time_of TimeTyp,             -- nested record
      dept    departments%ROWTYPE,
        -- nested record representing a table row
      place   VARCHAR2(20),
      purpose VARCHAR2(50) );
   meeting MeetingTyp;
   seminar MeetingTyp;
BEGIN
-- Can assign one nested record to another
-- if they are of the same data type
   seminar.time_of := meeting.time_of;
END;
/
```

Such assignments are allowed even if the containing records have different data types.

# Assignments to Records

To set all the fields in a record to initial values, assign to it an uninitialized record of the same type, as in Example 5–47.

***Example 5–47 Assigning Initial Values to a Record***

```
DECLARE
  TYPE RecordTyp IS RECORD (field1 NUMBER,
                            field2 VARCHAR2(32) := 'something');
  rec1 RecordTyp;
  rec2 RecordTyp;
BEGIN
  -- At first, rec1 has the values you assign:

  rec1.field1 := 100;
  rec1.field2 := 'something else';

  -- Assign empty record to rec1, resetting fields to initial values,
  -- which are NULL for field1 and 'something' for field2.

   rec1 := rec2;

   DBMS_OUTPUT.PUT_LINE (
     'field1 = ' || NVL(TO_CHAR(rec1.field1),'<NULL>') || ',
     field2 = ' || rec1.field2
   );
END;
/
```

Result:

```
field1 = <NULL>,
     field2 = something
```

You can assign a value to a field in a record using an assignment statement with dot notation:

```
emp_info.last_name := 'Fields';
```

Values are assigned separately to each field of a record in Example 5–47. You cannot assign a list of values to a record using an assignment statement. There is no constructor-like notation for records.

You can assign values to all fields simultaneously only if you assign a record to another record with the same data type. Having fields that match is not enough, as shown in Example 5–48.

***Example 5–48 Assigning All the Fields of a Record in One Statement***

```
DECLARE
-- Two identical type declarations.
   TYPE DeptRec1 IS RECORD
     (dept_num  NUMBER(2), dept_name VARCHAR2(14));
   TYPE DeptRec2 IS RECORD
    (dept_num  NUMBER(2), dept_name VARCHAR2(14));
   dept1_info DeptRec1;
   dept2_info DeptRec2;
   dept3_info DeptRec2;
BEGIN
-- Not allowed; different data types,
-- even though fields are the same.
--      dept1_info := dept2_info;
-- This assignment is OK because the records have the same type.
   dept2_info := dept3_info;
END;
/
```

You can assign a `%ROWTYPE` record to a user-defined record if their fields match in number and order, and corresponding fields have the same data types, as in Example 5–49.

***Example 5–49 Assigning a %ROWTYPE Record to a User-Defined Record***

```
DECLARE
   TYPE RecordTyp IS RECORD (last employees.last_name%TYPE,
                             id employees.employee_id%TYPE);
   CURSOR c1 IS SELECT last_name, employee_id FROM employees;
-- Rec1 and rec2 have different types,
-- but because rec2 is based on a %ROWTYPE,
-- you can assign it to rec1 if they have
-- the right number of fields and
-- the fields have the right data types.
   rec1 RecordTyp;
   rec2 c1%ROWTYPE;
BEGIN
  SELECT last_name, employee_id INTO rec2
  FROM employees
  WHERE ROWNUM < 2;

  rec1 := rec2;
```

```
  DBMS_OUTPUT.PUT_LINE
    ('Employee #' || rec1.id || ' = ' || rec1.last);
END;
/
```

Result:

```
Employee #100 = King
```

You can also use the SELECT or FETCH statement to fetch column values into a record. The columns in the select-list must appear in the same order as the fields in your record.

***Example 5–50 SELECT INTO Assigns Values to Record***

```
DECLARE
   TYPE RecordTyp IS RECORD (last employees.last_name%TYPE,
                             id employees.employee_id%TYPE);
   rec1 RecordTyp;
BEGIN
  SELECT last_name, employee_id INTO rec1
  FROM employees
  WHERE ROWNUM < 2;

  DBMS_OUTPUT.PUT_LINE
    ('Employee #' || rec1.id || ' = ' || rec1.last);
END;
/
```

Result:

```
Employee #100 = King
```

## Record Comparisons

To test records for nullity or equality, you must write functions that accept records as parameters and compare the corresponding fields, as in Example 5–51.

***Example 5–51 Testing Records for Nullity and Equality***

```
DECLARE
  TYPE RecTyp IS RECORD (
    field1 NUMBER,
    field2 VARCHAR2(10)
  );

  r1  RecTyp;
  r2  RecTyp;

  FUNCTION record_is_null (
    rec   RecTyp
  ) RETURN BOOLEAN
  IS
  BEGIN
    RETURN ((rec.field1 IS NULL) AND (rec.field2 IS NULL));
  END;

  FUNCTION records_are_equal (
    rec1  RecTyp,
    rec2  RecTyp
```

```
  ) RETURN BOOLEAN
  IS
  BEGIN
    RETURN (
      (rec1.field1 = rec2.field1) AND
      (rec1.field2 = rec2.field2)
    );
  END;

BEGIN
  r1.field1 := NULL;
  r1.field2 := NULL;

  r2.field1 := 123;
  r2.field2 := 'abc';

  IF record_is_null(r1) THEN
    DBMS_OUTPUT.PUT_LINE('r1 is NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('r1 is not NULL');
  END IF;

  IF record_is_null(r2) THEN
    DBMS_OUTPUT.PUT_LINE('r2 is NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('r2 is not NULL');
  END IF;

  IF records_are_equal (r1, r2) THEN
    DBMS_OUTPUT.PUT_LINE('r1 equals r2');
  ELSE
    DBMS_OUTPUT.PUT_LINE('r1 does not equal r2');
  END IF;

  r1.field1 := 123;
  r1.field2 := 'abc';

  IF records_are_equal (r1, r2) THEN
    DBMS_OUTPUT.PUT_LINE('r1 equals r2');
  ELSE
    DBMS_OUTPUT.PUT_LINE('r1 does not equal r2');
  END IF;
END;
/
```

Result:

```
r1 is NULL
r2 is not NULL
r1 does not equal r2
r1 equals r2
```

## Inserting Records into Tables

The PL/SQL extension to the SQL INSERT statement lets you insert a record into a table. The record must represent a row of the table. For more information, see "INSERT Statement Extension" on page 13-89. For restrictions on inserting records into tables, see "Restrictions on Record Inserts and Updates" on page 5-43.

Example 5–52 creates the table `schedule` and initializes it by putting default values in a record and inserting the record into the table once for each week. (The `COLUMN` formatting commands are from SQL*Plus.)

***Example 5–52 Initializing a Table by Inserting a Record of Default Values***

```
DROP TABLE schedule;
CREATE TABLE schedule (
  week  NUMBER,
  Mon   VARCHAR2(10),
  Tue   VARCHAR2(10),
  Wed   VARCHAR2(10),
  Thu   VARCHAR2(10),
  Fri   VARCHAR2(10),
  Sat   VARCHAR2(10),
  Sun   VARCHAR2(10)
);

DECLARE
  default_week  schedule%ROWTYPE;
  i             NUMBER;
BEGIN
  default_week.Mon := '0800-1700';
  default_week.Tue := '0800-1700';
  default_week.Wed := '0800-1700';
  default_week.Thu := '0800-1700';
  default_week.Fri := '0800-1700';
  default_week.Sat := 'Day Off';
  default_week.Sun := 'Day Off';

  FOR i IN 1..6 LOOP
    default_week.week    := i;

    INSERT INTO schedule VALUES default_week;
  END LOOP;
END;
/

COLUMN week FORMAT 99
COLUMN Mon  FORMAT A9
COLUMN Tue  FORMAT A9
COLUMN Wed  FORMAT A9
COLUMN Thu  FORMAT A9
COLUMN Fri  FORMAT A9
COLUMN Sat  FORMAT A9
COLUMN Sun  FORMAT A9

SELECT * FROM schedule;
```

Result:

```
WEEK MON       TUE       WED       THU       FRI       SAT       SUN
---- --------- --------- --------- --------- --------- --------- ---------
   1 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   2 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   3 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   4 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   5 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   6 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
```

To efficiently insert a collection of records into a table, put the INSERT statement inside a FORALL statement. For information about the FORALL statement, see "Running One DML Statement Multiple Times (FORALL Statement)" on page 12-11.

## Updating Rows with Records

The PL/SQL extension to the SQL UPDATE statement lets you update one or more table rows with a record. The record must represent a row of the table. For more information, see "UPDATE Statement Extensions" on page 13-129. For restrictions on updating table rows with a record, see "Restrictions on Record Inserts and Updates" on page 5-43.

Example 5–53 updates the first three weeks of the table schedule (created in Example 5–52) by putting the new values in a record and updating the first three rows of the table with that record.

***Example 5–53 Updating Rows with a Record***

```
DECLARE
  default_week  schedule%ROWTYPE;
BEGIN
  default_week.Mon := 'Day Off';
  default_week.Tue := '0900-1800';
  default_week.Wed := '0900-1800';
  default_week.Thu := '0900-1800';
  default_week.Fri := '0900-1800';
  default_week.Sat := '0900-1800';
  default_week.Sun := 'Day Off';

  FOR i IN 1..3 LOOP
    default_week.week    := i;

    UPDATE schedule
    SET ROW = default_week
    WHERE week = i;
  END LOOP;
/

SELECT * FROM schedule;
```

Result:

```
WEEK MON       TUE       WED       THU       FRI       SAT       SUN
---- --------- --------- --------- --------- --------- --------- ---------
   1 Day Off   0900-1800 0900-1800 0900-1800 0900-1800 0900-1800 Day Off
   2 Day Off   0900-1800 0900-1800 0900-1800 0900-1800 0900-1800 Day Off
   3 Day Off   0900-1800 0900-1800 0900-1800 0900-1800 0900-1800 Day Off
   4 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   5 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
   6 0800-1700 0800-1700 0800-1700 0800-1700 0800-1700 Day Off   Day Off
```

To efficiently update a set of rows with a collection of records, put the UPDATE statement inside a FORALL statement. For information about the FORALL statement, see "Running One DML Statement Multiple Times (FORALL Statement)" on page 12-11.

## Returning Rows into Records

The INSERT, UPDATE, and DELETE statements can include a RETURNING clause, which returns column values from the affected row into a PL/SQL record variable. This eliminates the need to SELECT the row after an insert or update, or before a delete.

By default, you can use this clause only when operating on exactly one row. When you use bulk SQL, you can use the form RETURNING BULK COLLECT INTO to store the results in one or more collections.

If you put an UPDATE statement with a RETURNING clause in a FORALL statement, you can retrieve values from a set of updated rows into a collection of records.

Example 5–54 updates the salary of an employee and retrieves the employee's name and new salary into a record variable.

***Example 5–54 RETURNING INTO Clause with Record***

```
DECLARE
   TYPE EmpRec IS RECORD (last_name  employees.last_name%TYPE,
                          salary     employees.salary%TYPE);
   emp_info EmpRec;
   emp_id   NUMBER := 100;
BEGIN
   UPDATE employees SET salary = salary * 1.1
     WHERE employee_id = emp_id
     RETURNING last_name, salary INTO emp_info;
   DBMS_OUTPUT.PUT_LINE
     ('Just gave a raise to ' || emp_info.last_name ||
      ', who now makes ' || emp_info.salary);
   ROLLBACK;
END;
/
```

Result:

```
Just gave a raise to King, who now makes 26510
```

## Restrictions on Record Inserts and Updates

These restrictions apply to record inserts and updates:

- Record variables are allowed only in these places:
  - On the right side of the SET clause in an UPDATE statement
  - In the VALUES clause of an INSERT statement
  - In the INTO subclause of a RETURNING clause

  Record variables are not allowed in a SELECT list, WHERE clause, GROUP BY clause, or ORDER BY clause.
- The keyword ROW is allowed only on the left side of a SET clause. Also, you cannot use ROW with a subquery.
- In an UPDATE statement, only one SET clause is allowed if ROW is used.
- If the VALUES clause of an INSERT statement contains a record variable, no other variable or value is allowed in the clause.

- If the INTO subclause of a RETURNING clause contains a record variable, no other variable or value is allowed in the subclause.
- These are not supported:
  - Nested record types
  - Functions that return a record
  - Record inserts and updates using the EXECUTE IMMEDIATE statement.

## Assignments to Collections of Records

You can use the BULK COLLECT clause with a SELECT INTO or FETCH statement to retrieve a set of rows into a collection of records.

***Example 5–55 BULK COLLECT with SELECT INTO Statement***

```
DECLARE
   TYPE EmployeeSet IS TABLE OF employees%ROWTYPE;
   underpaid EmployeeSet;
     -- Holds set of rows from EMPLOYEES table.
   CURSOR c1 IS SELECT first_name, last_name FROM employees;
   TYPE NameSet IS TABLE OF c1%ROWTYPE;
   some_names NameSet;
     -- Holds set of partial rows from EMPLOYEES table.
BEGIN
-- With one query,
-- bring all relevant data into collection of records.
   SELECT * BULK COLLECT INTO underpaid FROM employees
      WHERE salary < 5000 ORDER BY salary DESC;
-- Process data by examining collection or passing it to
-- eparate procedure, instead of writing loop to FETCH each row.
   DBMS_OUTPUT.PUT_LINE
     (underpaid.COUNT || ' people make less than 5000.');
   FOR i IN underpaid.FIRST .. underpaid.LAST
   LOOP
     DBMS_OUTPUT.PUT_LINE
       (underpaid(i).last_name || ' makes ' || underpaid(i).salary);
   END LOOP;
-- You can also bring in some table columns.
-- Here you get the first and last names of 10 arbitrary employees.
   SELECT first_name, last_name
     BULK COLLECT INTO some_names
     FROM employees
     WHERE ROWNUM < 11;
   FOR i IN some_names.FIRST .. some_names.LAST
   LOOP
      DBMS_OUTPUT.PUT_LINE
        ('Employee = ' || some_names(i).first_name
         || ' ' || some_names(i).last_name);
   END LOOP;
END;
/
```

Result:

```
49 people make less than 5000.
Pataballa makes 4800
Austin makes 4800
Whalen makes 4400
```

```
Lorentz makes 4300
Sarchand makes 4200
Bull makes 4100
Bell makes 4000
Everett makes 3900
Chung makes 3800
Ladwig makes 3600
Dilly makes 3600
Rajs makes 3500
Dellinger makes 3400
Mallin makes 3300
Bissot makes 3300
Taylor makes 3200
McCain makes 3200
Nayer makes 3200
Stiles makes 3200
Khoo makes 3100
Davies makes 3100
Walsh makes 3100
Fleaur makes 3100
Feeney makes 3000
Cabrio makes 3000
Rogers makes 2900
Gates makes 2900
Baida makes 2900
Jones makes 2800
Atkinson makes 2800
Geoni makes 2800
Tobias makes 2800
Seo makes 2700
Mikkilineni makes 2700
Matos makes 2600
Grant makes 2600
OConnell makes 2600
Himuro makes 2600
Marlow makes 2500
Perkins makes 2500
Patel makes 2500
Vargas makes 2500
Sullivan makes 2500
Colmenares makes 2500
Landry makes 2400
Gee makes 2400
Philtanker makes 2200
Markle makes 2200
Olson makes 2100
Employee = Ellen Abel
Employee = Sundar Ande
Employee = Mozhe Atkinson
Employee = David Austin
Employee = Hermann Baer
Employee = Shelli Baida
Employee = Amit Banda
Employee = Elizabeth Bates
Employee = Sarah Bell
Employee = David Bernstein
```

# 6

# PL/SQL Static SQL

**Static SQL** is a PL/SQL feature that allows SQL syntax directly in a PL/SQL statement. This chapter describes static SQL and explains how to use it.

Topics:


- Description of Static SQL
- Resolution of Names in Static SQL Statements
- SQL Pseudocolumns
- Cursors
- Queries
- Subqueries
- Cursor Variables
- Cursor Expressions
- Transaction Processing and Control
- Autonomous Transactions


## Description of Static SQL

These are the PL/SQL Static SQL statements, which have the same syntax as the corresponding SQL statements, except as noted:

- `SELECT`

  For the PL/SQL syntax, see "SELECT INTO Statement" on page 13-118.

- `INSERT`

  For the PL/SQL syntax, see "INSERT Statement Extension" on page 13-89

- `UPDATE`

  For the PL/SQL syntax, see "UPDATE Statement Extensions" on page 13-129

- `DELETE`

  For the PL/SQL syntax, see "DELETE Statement Extension" on page 13-42

- `MERGE` (for syntax, see *Oracle Database SQL Language Reference*)
- `LOCK TABLE` (for syntax, see *Oracle Database SQL Language Reference*)
- `COMMIT` (for syntax, see *Oracle Database SQL Language Reference*)
- `ROLLBACK` (for syntax, see *Oracle Database SQL Language Reference*)

- `SAVEPOINT` (for syntax, see *Oracle Database SQL Language Reference*)
- `SET TRANSACTION` (for syntax, see *Oracle Database SQL Language Reference*)

A PL/SQL static SQL statement can have a PL/SQL identifier wherever its SQL counterpart can have a placeholder for a bind argument. The PL/SQL identifier must identify either a variable or a formal parameter, not a function that is visible only in the PL/SQL unit that includes the static SQL statement.

In Example 6–1, a PL/SQL anonymous block declares three PL/SQL variables and uses them in the static SQL statements `INSERT`, `UPDATE`, `DELETE`. The block also uses the static SQL statement `COMMIT`.

***Example 6–1 Static SQL Statements***

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT employee_id, first_name, last_name
  FROM employees;

DECLARE
  emp_id          employees_temp.employee_id%TYPE := 299;
  emp_first_name  employees_temp.first_name%TYPE  := 'Bob';
  emp_last_name   employees_temp.last_name%TYPE   := 'Henry';
BEGIN
  INSERT INTO employees_temp (employee_id, first_name, last_name)
  VALUES (emp_id, emp_first_name, emp_last_name);

  UPDATE employees_temp
  SET first_name = 'Robert'
  WHERE employee_id = emp_id;

  DELETE FROM employees_temp
  WHERE employee_id = emp_id
  RETURNING first_name, last_name
  INTO emp_first_name, emp_last_name;

  COMMIT;
  DBMS_OUTPUT.PUT_LINE (emp_first_name || ' ' || emp_last_name);
END;
/
```

Result:

```
Robert Henry
```

To use PL/SQL identifiers for table names, column names, and so on, use the `EXECUTE IMMEDIATE` statement, explained in "Native Dynamic SQL" on page 7-2.

---

**Note:** After PL/SQL code runs a DML statement, the values of some variables are undefined. For example:

- After a `FETCH` or `SELECT` statement raises an exception, the values of the define variables after that statement are undefined.
- After a DML statement that affects zero rows, the values of the `OUT` bind variables are undefined, unless the DML statement is a `BULK` or multiple-row operation.

---

## Resolution of Names in Static SQL Statements

When the PL/SQL compiler finds a static SQL statement:

1. If the statement is a `SELECT` statement, the PL/SQL compiler removes the `INTO` clause.
2. The PL/SQL compiler sends the statement to the SQL subsystem.
3. The SQL subsystem checks the syntax of the statement.

   If the syntax is incorrect, the compilation of the PL/SQL unit fails. If the syntax is correct, the SQL subsystem determines the names of the tables and tries to resolve the other identifiers in the scope of the SQL statement.
4. If the SQL subsystem cannot resolve an identifier in the scope of the SQL statement, the SQL subsystem sends the identifier back to the PL/SQL compiler. The identifier is called an **escaped identifier**.
5. The PL/SQL compiler tries to resolve the escaped identifier.

   First, the compiler tries to resolve the identifier in the scope of the PL/SQL unit. If that fails, the compiler tries to resolve the identifier in the scope of the schema. If that fails, the compilation of the PL/SQL unit fails. For more information about PL/SQL name resolution, see Appendix B, "PL/SQL Name Resolution".
6. If the compilation of the PL/SQL unit succeeds, the PL/SQL compiler generates the text of the regular SQL statement that is equivalent to the static SQL statement and stores that text with the generated machine code.
7. At run time, the PL/SQL run-time system calls routines that parse, bind, and run the regular SQL statement.

   The bind arguments are the escaped identifiers (see step 4).
8. If the statement is a `SELECT` statement, the PL/SQL run-time system stores the results in the PL/SQL targets specified in the `INTO` clause that the PL/SQL compiler removed in step 1.

## SQL Pseudocolumns

PL/SQL recognizes the SQL pseudocolumns `CURRVAL`, `LEVEL`, `NEXTVAL`, `ROWID`, and `ROWNUM`. However, there are limitations on the use of pseudocolumns, including the restriction on the use of some pseudocolumns in assignments or conditional tests. For more information about SQL pseudocolumns, including restrictions, see *Oracle Database SQL Language Reference*.

Topics:

- CURRVAL and NEXTVAL
- LEVEL
- ROWID
- ROWNUM

### CURRVAL and NEXTVAL

`CURRVAL` and `NEXTVAL` are sequence pseudocolumns. A **sequence** is a schema object that generates sequential numbers. When you create a sequence, you can specify its initial value and an increment. `CURRVAL` returns the current value in a specified sequence. Before you can reference `CURRVAL` in a session, you must use `NEXTVAL` to

generate a number. A reference to NEXTVAL stores the current sequence number in CURRVAL. NEXTVAL increments the sequence and returns the next value. To get the current or next value in a sequence, use dot notation:

```
sequence_name.CURRVAL
sequence_name.NEXTVAL
```

The *sequence_name* can be either local or remote.

Each time you reference *sequence_name*.NEXTVAL, the sequence is incremented immediately and permanently, whether you commit or roll back the transaction.

Example 6–2 generates a sequence number and refers to that number in multiple statements. (The sequence must exist. Example 6–2 uses the sequence EMPLOYEES_SEQ, which is in the sample schema HR. To create a sequence, use the SQL statement CREATE SEQUENCE.)

***Example 6–2 CURRVAL and NEXTVAL Pseudocolumns***

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT employee_id, first_name, last_name
  FROM employees;

DROP TABLE employees_temp2;
CREATE TABLE employees_temp2 AS
  SELECT employee_id, first_name, last_name
  FROM employees;

DECLARE
  seq_value NUMBER;
BEGIN
  -- Generate initial sequence number

  seq_value := employees_seq.NEXTVAL;

  -- Print initial sequence number:

  DBMS_OUTPUT.PUT_LINE (
    'Initial sequence value: ' || TO_CHAR(seq_value)
  );

  -- Use NEXTVAL to create unique number when inserting data:

     INSERT INTO employees_temp (employee_id, first_name, last_name)
     VALUES (employees_seq.NEXTVAL, 'Lynette', 'Smith');

  -- Use CURRVAL to store same value somewhere else:

     INSERT INTO employees_temp2 VALUES (employees_seq.CURRVAL,
                                         'Morgan', 'Smith');

  /* Because NEXTVAL values might be referenced
     by different users and applications,
     and some NEXTVAL values might not be stored in database,
     there might be gaps in sequence. */

  -- Use CURRVAL to specify record to delete:

     seq_value := employees_seq.CURRVAL;
```

```
    DELETE FROM employees_temp2
    WHERE employee_id = seq_value;

  -- Udpate employee_id with NEXTVAL for specified record:

    UPDATE employees_temp
    SET employee_id = employees_seq.NEXTVAL
    WHERE first_name = 'Lynette'
    AND last_name = 'Smith';

  -- Display final value of CURRVAL:

    seq_value := employees_seq.CURRVAL;

    DBMS_OUTPUT.PUT_LINE (
      'Ending sequence value: ' || TO_CHAR(seq_value)
    );
END;
/
```

### Usage Notes

- You can use *sequence_name*.CURRVAL and *sequence_name*.NEXTVAL wherever you can use a NUMBER expression.
- Using *sequence_name*.CURRVAL or *sequence_name*.NEXTVAL to provide an initial value for an ADT method parameter causes a compilation error.
- PL/SQL evaluates every occurrence of *sequence_name*.CURRVAL and *sequence_name*.NEXTVAL (unlike SQL, which evaluates a sequence expression once for every row in which it appears).

**See Also:**

- *Oracle Database SQL Language Reference* for general information about CURRVAL and NEXTVAL
- *Oracle Database SQL Language Reference* for general information about sequences, including the CREATE SEQUENCE statement

## LEVEL

LEVEL is a hierarchical query pseudocolumn. If a table contains hierarchical data, you can select rows in a hierarchical order using the hierarchical query clause of the SELECT statement. For each row returned by a hierarchical query, the LEVEL pseudocolumn returns 1 for a root row, 2 for a child of a root, and so on.

**See Also:**

- *Oracle Database SQL Language Reference* for general information about LEVEL
- *Oracle Database SQL Language Reference* for general information about queries, including hierarchical queries

## ROWID

The ROWID pseudocolumn returns the rowid (binary address) of a row in a database table. You can use variables of type UROWID to store rowids in a readable format.

When you select or fetch a physical rowid into a UROWID variable, you can use the ROWIDTOCHAR function, which converts the binary value to a character string. You can compare the UROWID variable to the ROWID pseudocolumn in the WHERE clause of an UPDATE or DELETE statement to identify the latest row fetched from a cursor. For an example, see "Simulating CURRENT OF Clause with ROWID Pseudocolumn" on page 6-45.

**See Also:** *Oracle Database SQL Language Reference* for more information about the ROWID pseudocolumn

### ROWNUM

For each row returned by a query, the ROWNUM pseudocolumn returns a number indicating the order in which Oracle Database selects the row from a table or set of joined rows. The first row selected has a ROWNUM of 1, the second has 2, and so on.

---

**Note:** The SQL function ROW_NUMBER provides superior support for ordering the results of a query.

---

**See Also:**

- *Oracle Database SQL Language Reference* for general information about ROWNUM
- *Oracle Database SQL Language Reference* for general information about ROW_NUMBER

## Cursors

A **cursor** is a pointer to a private SQL area that stores information about processing a specific SELECT or DML statement.

The cursors that this chapter explains are session cursors. A **session cursor** lives in session memory until the session ends, when it ceases to exist. Session cursors are different from the cursors in the private SQL area of the program global area (PGA), explained in *Oracle Database Concepts*.

A session cursor that is constructed and managed by PL/SQL is called an **implicit cursor**. A session cursor that you construct and manage is called an **explicit cursor**. With an explicit cursor, you can retrieve the rows of a result set one at a time.

You can get information about any session cursor from its attributes (which you can use in procedural statements, but not in SQL statements).

To list the session cursors that each user session currently has opened and parsed, query the dynamic performance view V$OPEN_CURSOR, described in *Oracle Database Reference*.

Topics:

- Implicit Cursors
- Explicit Cursors

### Implicit Cursors

PL/SQL opens an implicit cursor every time you run a SELECT or DML statement. You cannot control an implicit cursor, but you can get information from its attributes.

The syntax of an implicit cursor attribute value is `SQL`*`attribute`* (therefore, an implicit cursor is also called a **SQL cursor**). `SQL`*`attribute`* always refers to the most recently run `SELECT` or DML statement. If no such statement has run, the value of `SQL`*`attribute`* is `NULL`.

An implicit cursor closes after its associated statement runs; however, its attribute values remain available until another `SELECT` or DML statement runs.

The most recently run `SELECT` or DML statement might be in a different scope. To save an attribute value for later use, assign it to a local variable immediately. Otherwise, other operations, such as subprogram calls, might change the value of the attribute before you can test it.

The implicit cursor attributes are:

- SQL%ISOPEN Attribute: Is the Cursor Open?
- SQL%FOUND Attribute: Were Any Rows Affected?
- SQL%NOTFOUND Attribute: Were No Rows Affected?
- SQL%ROWCOUNT Attribute: How Many Rows Were Affected?
- `SQL%BULK_ROWCOUNT` (see "Counting Rows Affected by FORALL" on page 12-19)
- `SQL%BULK_EXCEPTIONS` (see "Handling FORALL Exceptions" on page 12-17)

> **See Also:** "Implicit Cursor Attribute" on page 13-84 for a syntax diagram

### SQL%ISOPEN Attribute: Is the Cursor Open?

`SQL%ISOPEN` always returns `FALSE`, because an implicit cursor always closes after its associated statement runs.

### SQL%FOUND Attribute: Were Any Rows Affected?

`SQL%FOUND` returns:

- `NULL` if no `SELECT` or DML statement has run
- `TRUE` if a `SELECT` statement returned one or more rows or a DML statement affected one or more rows
- `FALSE` otherwise

Example 6–3 uses `SQL%FOUND` to determine if a `DELETE` statement affected any rows.

***Example 6–3 SQL%FOUND Attribute***

```
DROP TABLE dept_temp;
CREATE TABLE dept_temp AS
  SELECT * FROM departments;

CREATE OR REPLACE PROCEDURE p (
  dept_no NUMBER
) AUTHID DEFINER AS
BEGIN
  DELETE FROM dept_temp
  WHERE department_id = dept_no;

  IF SQL%FOUND THEN
    DBMS_OUTPUT.PUT_LINE (
      'Delete succeeded for department number ' || dept_no
    );
```

```
  ELSE
    DBMS_OUTPUT.PUT_LINE ('No department number ' || dept_no);
  END IF;
END;
/
BEGIN
  p(270);
  p(400);
END;
/
```

### SQL%NOTFOUND Attribute: Were No Rows Affected?

SQL%NOTFOUND (the logical opposite of SQL%FOUND) returns:

- NULL if no SELECT or DML statement has run
- FALSE if a SELECT statement returned one or more rows or a DML statement affected one or more rows
- TRUE otherwise

The SQL%NOTFOUND attribute is not useful with the PL/SQL SELECT INTO statement, because:

- If the SELECT INTO statement returns no rows, PL/SQL raises the predefined exception NO_DATA_FOUND immediately, before you can check SQL%NOTFOUND.
- A SELECT INTO statement that invokes a SQL aggregate function always returns a value (possibly NULL). After such a statement, the SQL%NOTFOUND attribute is always FALSE, so checking it is unnecessary.

### SQL%ROWCOUNT Attribute: How Many Rows Were Affected?

SQL%ROWCOUNT returns:

- NULL if no SELECT or DML statement has run
- Otherwise, the number of rows returned by a SELECT statement or affected by a DML statement

Example 6–4 uses SQL%ROWCOUNT to determine the number of rows that were deleted.

***Example 6–4 SQL%ROWCOUNT Attribute***

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT * FROM employees;

DECLARE
  mgr_no NUMBER(6) := 122;
BEGIN
  DELETE FROM employees_temp WHERE manager_id = mgr_no;
  DBMS_OUTPUT.PUT_LINE
    ('Number of employees deleted: ' || TO_CHAR(SQL%ROWCOUNT));
END;
/
```

Result:

```
Number of employees deleted: 8
```

If a SELECT INTO statement without a BULK COLLECT clause returns multiple rows, PL/SQL raises the predefined exception TOO_MANY_ROWS and SQL%ROWCOUNT returns 1, not the actual number of rows that satisfy the query.

The value of SQL%ROWCOUNT attribute is unrelated to the state of a transaction. Therefore:

- When a transaction rolls back to a savepoint, the value of SQL%ROWCOUNT is not restored to the value it had before the savepoint.
- When an autonomous transaction ends, SQL%ROWCOUNT is not restored to the original value in the parent transaction.

## Explicit Cursors

You must declare an explicit cursor, giving it a name and associating it with a query—usually one that returns multiple rows. After declaring an explicit cursor, you must open it (with the OPEN statement), fetch rows one at a time from the result set (with the FETCH statement), and close the cursor (with the CLOSE statement). After closing the cursor, you can neither fetch records from the result set nor see the cursor attribute values.

Topics:

- Declaring Explicit Cursors
- Opening Explicit Cursors
- Fetching Data with Explicit Cursors
- Fetching Bulk Data with Explicit Cursors
- Closing Explicit Cursors
- Explicit Cursor Attributes
- When Explicit Cursors Need Column Aliases
- Explicit Cursors that Accept Parameters

**See Also:** "Cursor Variable" on page 13-39

### Declaring Explicit Cursors

You must declare an explicit cursor before referencing it in other statements. You give the cursor a name and associate it with a specific query. You can optionally declare a return type for the cursor, such as *table_name*%ROWTYPE. You can optionally specify parameters that you use in the WHERE clause instead of referring to local variables. These parameters can have initial values. Example 6–5 declares two explicit cursors.

**Note:** An explicit cursor declared in a package specification is affected by the AUTHID clause of the package. For more information, see "CREATE PACKAGE Statement" on page 14-42.

***Example 6–5 Explicit Cursor Declaration***

```
DECLARE
  my_emp_id  NUMBER(6);     -- variable for employee_id
  my_job_id  VARCHAR2(10);  -- variable for job_id
  my_sal     NUMBER(8,2);   -- variable for salary
```

```
  CURSOR c1 IS
    SELECT employee_id, job_id, salary FROM employees
    WHERE salary > 2000;

  my_dept   departments%ROWTYPE;  -- variable for departments row

  CURSOR c2 RETURN departments%ROWTYPE IS
    SELECT * FROM departments
    WHERE department_id = 110;
BEGIN
  NULL;
END;
/
```

The cursor is not a PL/SQL variable. You cannot assign a value to a cursor or use it in an expression. Cursors and variables follow the same scoping rules. Naming cursors after database tables is possible but not recommended.

A cursor can take parameters, which can appear in the associated query wherever constants can appear. The formal parameters of a cursor must be `IN` parameters; they supply values in the query, but do not return any values from the query. You cannot impose the `NOT NULL` constraint on a cursor parameter.

As this example shows, you can initialize cursor parameters to initial values. You can pass different numbers of actual parameters to a cursor, accepting or overriding the initial values. Also, you can add formal parameters without having to change existing references to the cursor.

```
DECLARE
   CURSOR c1 (low  NUMBER := 0, high NUMBER := 99) IS
             SELECT * FROM departments WHERE department_id > low
             AND department_id < high;
```

Cursor parameters can be referenced only in the query specified in the cursor declaration. The parameter values are used by the associated query when the cursor is opened.

**See Also:** "Explicit Cursor" on page 13-54

### Opening Explicit Cursors

You open an explicit cursor with the `OPEN` statement (for its syntax, see "OPEN Statement" on page 13-97). The `OPEN` statement allocates database resources to process the query, identifies the result set, and positions the cursor before the first row of the result set. (If the query has a `FOR UPDATE` clause, processing the query includes locking the rows of the result set—see "SELECT FOR UPDATE and FOR UPDATE Cursors" on page 6-44.)

Example 6–6 declares and opens an explicit cursor.

***Example 6–6   Opening an Explicit Cursor***

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name, job_id, salary
    FROM employees
    WHERE salary > 2000;
BEGIN
  OPEN c1;
END;
/
```

Generally, PL/SQL parses an explicit cursor only the first time it is opened and parses a SQL statement (creating an implicit cursor) only the first time the statement runs. All parsed SQL statements are cached. A SQL statement is reparsed only if it is aged out of the cache by a new SQL statement. Although you must close a cursor before you can reopen it, PL/SQL need not reparse the associated query. If you close and immediately reopen a cursor, PL/SQL does not reparse the associated query.

After opening an explicit cursor, use the FETCH statement to retrieve the rows of the result set.

### Fetching Data with Explicit Cursors

Unless you use the BULK COLLECT clause, explained in "Fetching Data with Explicit Cursors" on page 6-11, the FETCH statement retrieves the rows in the result set one at a time. Each fetch retrieves the current row and advances the cursor to the next row in the result set. You can store each column in a separate variable, or store the entire row in a record that has the appropriate fields, usually declared using %ROWTYPE.

For each column value returned by the query associated with the cursor, there must be a corresponding, type-compatible variable in the INTO list.

Typically, you use the FETCH statement inside a LOOP statement. A sequence of FETCH statements always runs out of data to retrieve, and no exception is raised when a FETCH statement returns no data. To detect this condition, use the cursor attribute %FOUND or %NOTFOUND, as in Example 6–7.

***Example 6–7 Fetching with a Cursor***

```
DECLARE
  v_jobid     employees.job_id%TYPE;     -- variable for job_id
  v_lastname  employees.last_name%TYPE;  -- variable for last_name

  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK');

  v_employees employees%ROWTYPE;         -- record variable for row

  CURSOR c2 IS
    SELECT * FROM employees
    WHERE REGEXP_LIKE (job_id, '[ACADFIMKSA]_M[ANGR]');

BEGIN
  OPEN c1;
  LOOP  -- Fetches 2 columns into variables
    FETCH c1 INTO v_lastname, v_jobid;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_lastname, 25, ' ') || v_jobid );
  END LOOP;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE( '-------------------------------------' );

  OPEN c2;
  LOOP  -- Fetches entire row into the v_employees record
    FETCH c2 INTO v_employees;
    EXIT WHEN c2%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_employees.last_name, 25, ' ') ||
                               v_employees.job_id );
  END LOOP;
```

```
  CLOSE c2;
END;
/
```

Result:

```
Nayer                    ST_CLERK
Mikkilineni              ST_CLERK
Landry                   ST_CLERK
...
Higgins                  AC_MGR
```

The query can reference PL/SQL variables in its scope. Any variables in the query are evaluated only when the cursor is opened. In Example 6–8, each retrieved salary is multiplied by 2, even though `factor` is incremented after every fetch.

***Example 6–8 Referencing PL/SQL Variables in Its Scope***

```
DECLARE
  my_sal employees.salary%TYPE;
  my_job employees.job_id%TYPE;
  factor INTEGER := 2;
  CURSOR c1 IS
    SELECT factor*salary FROM employees
    WHERE job_id = my_job;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_sal;
    EXIT WHEN c1%NOTFOUND;
    factor := factor + 1;  -- does not affect FETCH
  END LOOP;
  CLOSE c1;
END;
/
```

To change the result set or the values of variables in the query, you must close and reopen the cursor with the input variables set to their new values. However, you can use a different `INTO` list on separate fetches with the same cursor. Each fetch retrieves another row and assigns values to the target variables, as Example 6–9 shows.

***Example 6–9 Fetching the Same Cursor Into Different Variables***

```
DECLARE
  CURSOR c1 IS
    SELECT last_name FROM employees
    ORDER BY last_name;

  name1 employees.last_name%TYPE;
  name2 employees.last_name%TYPE;
  name3 employees.last_name%TYPE;
BEGIN
  OPEN c1;
  FETCH c1 INTO name1;  -- fetches first row
  FETCH c1 INTO name2;  -- fetches second row
  FETCH c1 INTO name3;  -- fetches third row
  CLOSE c1;
END;
/
```

If you fetch past the last row in the result set, the values of the target variables are undefined. Eventually, the FETCH statement fails to return a row. When that happens, no exception is raised. To detect the failure, use the cursor attribute %FOUND or %NOTFOUND. For more information, see "Cursor Expressions" on page 6-35.

### Fetching Bulk Data with Explicit Cursors

The BULK COLLECT clause lets you fetch all rows from the result set simultaneously. See "Retrieving Query Results into Collections" on page 12-21. Example 6–10 bulk-fetches from a cursor into two collections.

***Example 6–10 Fetching Bulk Data with a Cursor***

```
DECLARE
  TYPE IdsTab  IS TABLE OF employees.employee_id%TYPE;
  TYPE NameTab IS TABLE OF employees.last_name%TYPE;

  ids    IdsTab;
  names  NameTab;

  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM employees
    WHERE job_id = 'ST_CLERK';

BEGIN
  OPEN c1;
  FETCH c1 BULK COLLECT INTO ids, names;
  CLOSE c1;
  -- Process collection elements
  FOR i IN ids.FIRST .. ids.LAST
    LOOP
      IF ids(i) > 140 THEN
        DBMS_OUTPUT.PUT_LINE( ids(i) );
      END IF;
    END LOOP;
  FOR i IN names.FIRST .. names.LAST
    LOOP
      IF names(i) LIKE '%Ma%' THEN
        DBMS_OUTPUT.PUT_LINE( names(i) );
      END IF;
    END LOOP;
END;
/
```

Result:

```
141
142
143
144
Markle
Marlow
Mallin
Matos
```

### Closing Explicit Cursors

The CLOSE statement disables the cursor, and the result set becomes undefined. Once a cursor is closed, you can reopen it, which runs the query again with the latest values of

any cursor parameters and variables referenced in the WHERE clause. Any other operation on a closed cursor raises the predefined exception INVALID_CURSOR.

### Explicit Cursor Attributes

> **Note:** This topic applies to both explicit cursors and cursor variables. Explicit cursors and cursor variables are called **named cursors**, because you can reference them by their names. For more information about cursor variables, see "Cursor Variables" on page 6-25.

The syntax for the value of a named cursor attribute is *cursor_name* immediately followed by *attribute* (for example, c1%ISOPEN).

The named cursor attributes are:

- %ISOPEN Attribute: Is the Cursor Open?
- %FOUND Attribute: Has a Row Been Fetched?
- %NOTFOUND Attribute: Has No Row Been Fetched?
- %ROWCOUNT Attribute: How Many Rows Were Fetched?

If a named cursor is not open, referencing any attribute except %ISOPEN raises INVALID_CURSOR.

**%ISOPEN Attribute: Is the Cursor Open?** %ISOPEN returns TRUE if its named cursor is open; FALSE otherwise. Example 6–11 uses %ISOPEN to select an action.

***Example 6–11 %ISOPEN Attribute***

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11;

  the_name employees.last_name%TYPE;
  the_salary employees.salary%TYPE;
BEGIN
  IF c1%ISOPEN = FALSE THEN
    OPEN c1;
  END IF;
  FETCH c1 INTO the_name, the_salary;
  CLOSE c1;
END;
/
```

**%FOUND Attribute: Has a Row Been Fetched?** %FOUND returns:

- NULL after the named cursor is opened but before the first fetch
- TRUE if the last fetch from the named cursor returned a row
- FALSE otherwise

Example 6–12 uses %FOUND to select an action.

***Example 6–12 %FOUND Attribute***

```
DECLARE
  CURSOR c1 IS
```

```
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11;

  my_ename   employees.last_name%TYPE;
  my_salary  employees.salary%TYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_ename, my_salary;
    IF c1%FOUND THEN  -- fetch succeeded
      DBMS_OUTPUT.PUT_LINE('Name = ' || my_ename || ', salary = ' || my_salary);
    ELSE  -- fetch failed
      EXIT;
    END IF;
  END LOOP;
END;
/
```

Result:

```
Name = King, salary = 24100
Name = Kochhar, salary = 17000
Name = De Haan, salary = 17000
Name = Hunold, salary = 9100
Name = Ernst, salary = 6000
Name = Austin, salary = 4800
Name = Pataballa, salary = 4800
Name = Lorentz, salary = 4300
Name = Greenberg, salary = 12008
Name = Faviet, salary = 9000
```

**%NOTFOUND Attribute: Has No Row Been Fetched?** `%NOTFOUND` (the logical opposite of `%FOUND`) returns:

- `NULL` after the named cursor is opened but before the first fetch
- `FALSE` if the last fetch from the named cursor returned a row
- `TRUE` otherwise

Example 6–13 uses `%NOTFOUND` to exit a loop when `FETCH` fails to return a row.

***Example 6–13 %NOTFOUND Attribute***

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11;

   my_ename   employees.last_name%TYPE;
   my_salary  employees.salary%TYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_ename, my_salary;
    IF c1%NOTFOUND THEN -- fetch failed
    -- Another form of this test is
    -- "EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;"
      EXIT;
    ELSE  -- fetch succeeded
      DBMS_OUTPUT.PUT_LINE
        ('Name = ' || my_ename || ', salary = ' || my_salary);
```

```
    END IF;
  END LOOP;
END;
/
```

Result:

```
Name = King, salary = 24100
Name = Kochhar, salary = 17000
Name = De Haan, salary = 17000
Name = Hunold, salary = 9100
Name = Ernst, salary = 6000
Name = Austin, salary = 4800
Name = Pataballa, salary = 4800
Name = Lorentz, salary = 4300
Name = Greenberg, salary = 12008
Name = Faviet, salary = 9000
```

In Example 6–13, if `FETCH` never fetches a row, the loop is never exited. The `EXIT WHEN` statement runs only if its `WHEN` condition is `TRUE`. Before the first fetch, `%NOTFOUND` returns `NULL`. To prevent infinite looping, use this `EXIT` statement instead:

```
EXIT WHEN c1%NOTFOUND OR (c1%NOTFOUND IS NULL);
```

**%ROWCOUNT Attribute: How Many Rows Were Fetched?** `%ROWCOUNT` returns:

- Zero after the named cursor is opened but before the first fetch
- Otherwise, the number of rows fetched.

Example 6–14 uses `%ROWCOUNT` to determine when the fifth row is fetched.

***Example 6–14 %ROWCOUNT Attribute***

```
DECLARE
  CURSOR c1 IS
    SELECT last_name FROM employees
    WHERE ROWNUM < 11;

  name  employees.last_name%TYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO name;
    EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
    DBMS_OUTPUT.PUT_LINE(c1%ROWCOUNT || '. ' || name);
    IF c1%ROWCOUNT = 5 THEN
       DBMS_OUTPUT.PUT_LINE('--- Fetched 5th record ---');
    END IF;
  END LOOP;
  CLOSE c1;
END;
/
```

Result:

```
1. Abel
2. Ande
3. Atkinson
4. Austin
5. Baer
```

```
--- Fetched 5th record ---
6. Baida
7. Banda
8. Bates
9. Bell
10. Bernstein
```

Table 6–1 shows the value of each cursor attribute before and after OPEN, FETCH, and CLOSE statements run.

***Table 6–1 Named Cursor Attribute Values***

| Point in Time | %FOUND Value | %ISOPEN Value | %NOTFOUND Value | %ROWCOUNT Value |
|---|---|---|---|---|
| Before OPEN | exception | FALSE | exception | exception |
| After OPEN | NULL | TRUE | NULL | 0 |
| Before first FETCH | NULL | TRUE | NULL | 0 |
| After first FETCH | TRUE | TRUE | FALSE | 1 |
| Before each successive FETCH except last | TRUE | TRUE | FALSE | 1 |
| After each successive FETCH except last | TRUE | TRUE | FALSE | data dependent |
| Before last FETCH | TRUE | TRUE | FALSE | data dependent |
| After last FETCH | FALSE | TRUE | TRUE | data dependent |
| Before CLOSE | FALSE | TRUE | TRUE | data dependent |
| After CLOSE | exception | FALSE | exception | exception |

In Table 6–1:

- Referencing %FOUND, %NOTFOUND, or %ROWCOUNT before a cursor is opened or after it is closed raises INVALID_CURSOR.
- After the first FETCH, if the result set was empty, %FOUND returns FALSE, %NOTFOUND returns TRUE, and %ROWCOUNT returns 0.

## When Explicit Cursors Need Column Aliases

When the SELECT statement of an explicit cursor includes a calculated column (an expression), that column must have an alias if either of the following is true:

- You use the cursor to fetch into a record that was declared with %ROWTYPE.
- You want to reference the calculated column in your program.

In Example 6–15, the calculated column in the explicit cursor needs an alias for both of the preceding reasons.

***Example 6–15 Explicit Cursor with Calculated Column that Needs Alias***

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id,
           (salary * .05) raise
    FROM employees
    WHERE job_id LIKE '%_MAN';
  emp_rec c1%ROWTYPE;
```

```
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO emp_rec;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (
      'Raise for employee #' || emp_rec.employee_id ||
      ' is $' || emp_rec.raise
    );
  END LOOP;
  CLOSE c1;
END;
/
```

> **See Also:** "Column Aliases for Expression Values in Cursor FOR LOOP" on page 6-22

### Explicit Cursors that Accept Parameters

Instead of referring to local variables, you can declare a cursor that accepts parameters, and pass values for those parameters when you open the cursor. You can use either positional notation or named notation to pass the parameter values (for information about parameter notations, see "Positional, Named, and Mixed Notation for Actual Parameters" on page 8-12).

If the query is usually issued with certain values, you can make those values the initial values of the cursor parameters. A cursor parameter declared with an initial value does not need a corresponding actual parameter. If you omit the actual parameter, the formal parameter assumes its initial value when the `OPEN` statement runs. If the initial value of a formal parameter is an expression, and you provide a corresponding actual parameter in the `OPEN` statement, the expression is not evaluated.

> **Tip:** To avoid confusion, use different names for cursor parameters and the PL/SQL variables that you pass to those parameters.

Example 6–16 declares a cursor that accepts two parameters, and then uses a cursor `FOR LOOP` to display the wages paid to employees who earn more than a specified wage in a specified department.

***Example 6–16 Passing Parameters to a Cursor FOR LOOP***

```
DECLARE
  CURSOR c1 (job VARCHAR2, max_wage NUMBER) IS
    SELECT * FROM employees
    WHERE job_id = job
    AND salary > max_wage;
BEGIN
  FOR person IN c1('CLERK', 3000)
  LOOP
     -- process data record
    DBMS_OUTPUT.PUT_LINE (
      'Name = ' || person.last_name || ', salary = ' ||
      person.salary || ', Job Id = ' || person.job_id
    );
  END LOOP;
END;
/
```

In Example 6–17 declares a cursor that accepts two parameters, and shows several ways to open that cursor.

***Example 6–17 Passing Parameters to Explicit Cursors***

```
DECLARE
  emp_job     employees.job_id%TYPE := 'ST_CLERK';
  emp_salary  employees.salary%TYPE := 3000;
  my_record   employees%ROWTYPE;

  CURSOR c1 (job VARCHAR2, max_wage NUMBER) IS
    SELECT * FROM employees
    WHERE job_id = job
    AND salary > max_wage;
BEGIN
/* Open the cursor with one of these statements:
   OPEN c1('ST_CLERK', 3000);
   OPEN c1('ST_CLERK', emp_salary);
   OPEN c1(emp_job, 3000);
   OPEN c1(emp_job, emp_salary); */

  OPEN c1(emp_job, emp_salary);

  LOOP
     FETCH c1 INTO my_record;
     EXIT WHEN c1%NOTFOUND;
     DBMS_OUTPUT.PUT_LINE
       ('Name = ' || my_record.last_name || ', salary = ' ||
        my_record.salary || ', Job Id = ' || my_record.job_id );
  END LOOP;
END;
/
```

Result:

```
Name = Nayer, salary = 3200, Job Id = ST_CLERK
Name = Bissot, salary = 3300, Job Id = ST_CLERK
Name = Mallin, salary = 3300, Job Id = ST_CLERK
Name = Ladwig, salary = 3600, Job Id = ST_CLERK
Name = Stiles, salary = 3200, Job Id = ST_CLERK
Name = Rajs, salary = 3500, Job Id = ST_CLERK
Name = Davies, salary = 3100, Job Id = ST_CLERK
```

# Queries

PL/SQL lets you perform queries and access individual fields or entire rows from the result set. In traditional database programming, you process query results using an internal data structure called a cursor. In most situations, PL/SQL can manage the cursor for you, so that code to process query results is straightforward and compact. This section explains how to process both simple queries where PL/SQL manages everything, and complex queries where you interact with the cursor.

To process a multiple-row query, you must use either a cursor `FOR` loop (an implicit cursor in a `FOR LOOP`) or an explicit cursor and `FETCH` statement. The cursor `FOR` loop needs less code, but the explicit cursor is more flexible. For example, you can:

- Process several queries in parallel by declaring and opening multiple cursors.
- Process multiple rows in a single loop iteration, skip rows, or split the processing into multiple loops.

Topics:

- Selecting At Most One Row (SELECT INTO Statement)
- Selecting Multiple Rows (BULK COLLECT Clause)
- Looping Through Multiple Rows (Cursor FOR LOOP Statement)
- Processing Complicated Queries with Explicit Cursors

## Selecting At Most One Row (SELECT INTO Statement)

If you expect a query to only return one row, you can write a regular SQL `SELECT` statement with an additional `INTO` clause specifying the PL/SQL variable to hold the result.

If the query might return multiple rows, but you do not care about values after the first, you can restrict any result set to a single row by comparing the `ROWNUM` value. If the query might return no rows at all, use an exception handler to specify any actions to take when no data is found.

If you just want to check whether a condition exists in your data, you might be able to code the query with the `COUNT(*)` operator, which always returns a number and never raises the `NO_DATA_FOUND` exception.

For more information about the `SELECT INTO` statement, see "SELECT INTO Statement" on page 13-118.

## Selecting Multiple Rows (BULK COLLECT Clause)

If you must bring a large quantity of data into local PL/SQL variables, rather than looping through a result set one row at a time, you can use the `BULK COLLECT` clause. When you query only certain columns, you can store all the results for each column in a separate collection variable. When you query all the columns of a table, you can store the entire result set in a collection of records, which makes it convenient to loop through the results and refer to different columns. See Example 6–10, "Fetching Bulk Data with a Cursor" on page 6-13.

This technique can be very fast, but also very memory-intensive. If you use it often, you might be able to improve your code by doing more of the work in SQL:

- If you must loop once through the result set, use a `FOR LOOP` as described in the following sections. This technique avoids the memory overhead of storing a copy of the result set.
- If you are looping through the result set to scan for certain values or filter the results into a smaller set, do this scanning or filtering in the original query instead. You can add more `WHERE` clauses in simple cases, or use set operators such as `INTERSECT` and `MINUS` if you are comparing two or more sets of results.
- If you are looping through the result set and running another query or a DML statement for each result row, you can probably find a more efficient technique. For queries, look at including subqueries or `EXISTS` or `NOT EXISTS` clauses in the original query. For DML statements, look at the `FORALL` statement, which is much faster than coding these statements inside a regular loop.

**See Also:**

- "FETCH Statement" on page 13-67 for more information about its BULK COLLECT clause
- "SELECT INTO Statement" on page 13-118 for more information about its BULK COLLECT clause

## Looping Through Multiple Rows (Cursor FOR LOOP Statement)

Perhaps the most common case of a query is one where you issue the SELECT statement, then immediately loop through the rows of the result set. PL/SQL lets you use a cursor FOR LOOP for this kind of query.

The iterator variable for the cursor FOR LOOP does need not be declared in advance. It is a %ROWTYPE record whose field names match the column names from the query, and that exists only during the loop. When you use expressions rather than explicit column names, use column aliases so that you can refer to the corresponding values inside the loop.

Topics:

- Implicit Cursor FOR LOOP
- Explicit Cursor FOR LOOP
- Column Aliases for Expression Values in Cursor FOR LOOP

**See Also:** "Cursor FOR LOOP Statement" on page 13-37 for the syntax of the cursor FOR LOOP

### Implicit Cursor FOR LOOP

With PL/SQL, it is very simple to issue a query, retrieve each row of the result into a %ROWTYPE record, and process each row in a loop:

- You include the text of the query directly in the FOR LOOP.
- PL/SQL creates a record variable with fields corresponding to the columns of the result set.
- You refer to the fields of this record variable inside the loop. You can perform tests and calculations, display output, or store the results somewhere else.

In Example 6–18, a query returns the name and job ID of employees with manager IDs greater than 120.

***Example 6–18 Implicit Cursor FOR Loop***

```
BEGIN
  FOR item IN (
    SELECT last_name, job_id
    FROM employees
    WHERE job_id LIKE '%CLERK%'
    AND manager_id > 120
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/
```

Result:

```
Name = Bissot, Job = ST_CLERK
Name = Atkinson, Job = ST_CLERK
Name = Marlow, Job = ST_CLERK
...
Name = Grant, Job = SH_CLERK
```

Before each iteration of the FOR LOOP, PL/SQL fetches into the implicitly declared record. The sequence of statements inside the loop runs once for each row that satisfies the query. When you leave the loop, the cursor is closed automatically. The cursor is closed even if you use an EXIT or GOTO statement to leave the loop before all rows are fetched, or an exception is raised inside the loop. See "Cursor FOR LOOP Statement" on page 13-37.

### Explicit Cursor FOR LOOP

If you must reference the same query from different parts of the same subprogram, you can declare a cursor that specifies the query, and process the results using a FOR LOOP statement.

***Example 6–19 Explicit Cursor FOR LOOP***

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE job_id LIKE '%CLERK%' AND manager_id > 120;
BEGIN
  FOR item IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/
```

Result:

```
Name = Bissot, Job = ST_CLERK
Name = Atkinson, Job = ST_CLERK
Name = Marlow, Job = ST_CLERK
...
Name = Grant, Job = SH_CLERK
```

### Column Aliases for Expression Values in Cursor FOR LOOP

In a cursor FOR LOOP, PL/SQL creates a %ROWTYPE record with fields corresponding to columns in the result set. The fields have the same names as corresponding columns in the SELECT list.

The select list might contain an expression, such as a column plus a constant, or two columns concatenated. If so, use a column alias to give unique names to the appropriate columns.

In Example 6–20, full_name and dream_salary are aliases for expressions in the query.

***Example 6–20 Alias for Expressions in Query***

```
BEGIN
  FOR item IN (
```

```
    SELECT first_name || ' ' || last_name AS full_name,
           salary * 10                    AS dream_salary
    FROM employees
    WHERE ROWNUM <= 5
  ) LOOP
    DBMS_OUTPUT.PUT_LINE
      (item.full_name || ' dreams of making ' || item.dream_salary);
  END LOOP;
END;
/
```

Result:

```
Steven King dreams of making 241000
Neena Kochhar dreams of making 170000
Lex De Haan dreams of making 170000
Alexander Hunold dreams of making 91000
Bruce Ernst dreams of making 60000
```

**See Also:** "When Explicit Cursors Need Column Aliases" on page 6-17

## Processing Complicated Queries with Explicit Cursors

For full control over query processing, you can use explicit cursors in combination with the OPEN, FETCH, and CLOSE statements.

You might want to specify a query in one place but retrieve the rows somewhere else, even in another subprogram. Or you might want to choose very different query parameters, such as ORDER BY or GROUP BY clauses, depending on the situation. Or you might want to process some rows differently than others, and so need more than a simple loop.

Because explicit cursors are so flexible, you can choose from different notations depending on your needs. The following sections describe all the query-processing features that explicit cursors provide.

For more information about explicit cursors, see "Explicit Cursors" on page 6-9.

# Subqueries

A **subquery** is a query (usually enclosed in parentheses) that appears in another DML statement. The DML statement acts upon the result set of the subquery. For example:

- You can use a subquery to find the MAX, MIN, or AVG value for a column, and use that single value in a comparison in a WHERE clause.
- You can use a subquery to find a set of values, and use this values in an IN or NOT IN comparison in a WHERE clause. This technique can avoid joins.
- You can filter a set of values with a subquery, and apply other operations like ORDER BY and GROUP BY in the outer query.
- You can use a subquery for a table name, in the FROM clause of a query. This technique lets you join a table with a small set of rows from another table, instead of joining the entire tables.
- You can create a table or insert into a table, using a set of rows defined by a subquery.

Example 6–21 uses subqueries in its cursor declarations.

***Example 6–21 Subqueries in Cursor Declarations***

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name
    FROM employees
    WHERE salary > ( SELECT AVG(salary) FROM employees );

  CURSOR c2 IS
    SELECT *
    FROM ( SELECT last_name, salary
           FROM employees
           ORDER BY salary DESC, last_name
         )
    ORDER BY salary DESC, last_name;
BEGIN
  FOR person IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Above-average salary: ' || person.last_name);
  END LOOP;

  FOR person IN c2
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Highest paid: ' || person.last_name || ' $' || person.salary);
  END LOOP;
END;
/
```

Result:

```
Above-average salary: King
Above-average salary: Kochhar
Above-average salary: De Haan
...
Above-average salary: Gietz
Highest paid: King $24100
Highest paid: De Haan $17000
Highest paid: Kochhar $17000
...
Highest paid: Olson $2100
```

Using a subquery in the `FROM` clause, the query in Example 6–22 returns the number and name of each department with five or more employees.

***Example 6–22 Subquery in FROM Clause***

```
DECLARE
  CURSOR c1 IS
    SELECT t1.department_id, department_name, staff
    FROM departments t1,
         ( SELECT department_id, COUNT(*) AS staff
           FROM employees
           GROUP BY department_id
         ) t2
    WHERE (t1.department_id = t2.department_id) AND staff >= 5;

BEGIN
   FOR dept IN c1
   LOOP
     DBMS_OUTPUT.PUT_LINE ('Department = '
```

```
        || dept.department_name || ', staff = ' || dept.staff);
   END LOOP;
END;
/
```

Result:

```
Department = Shipping, staff = 45
Department = IT, staff = 5
Department = Finance, staff = 6
Department = Sales, staff = 34
Department = Purchasing, staff = 6
```

While a subquery is evaluated once for each table, a **correlated subquery** is evaluated once for each row. Example 6–23 returns the name and salary of each employee whose salary exceeds the departmental average. For each row in the table, the correlated subquery computes the average salary for the corresponding department.

***Example 6–23 Correlated Subquery***

```
DECLARE
  CURSOR c1 IS
    SELECT department_id, last_name, salary
    FROM employees t
    WHERE salary > ( SELECT AVG(salary)
                     FROM employees
                     WHERE t.department_id = department_id
                   )
    ORDER BY department_id;
BEGIN
  FOR person IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE('Making above-average salary = ' || person.last_name);
  END LOOP;
END;
/
```

Result:

```
Making above-average salary = Hartstein
Making above-average salary = Raphaely
Making above-average salary = Weiss
...
Making above-average salary = Higgins
```

# Cursor Variables

A **cursor variable** is like an explicit cursor, except that it is not limited to one query. You can open a cursor variable for a query, process the result set, and then use the cursor variable for another query. A cursor variable has the same attributes as an explicit cursor (for details, see "Explicit Cursor Attributes" on page 6-14).

The data type of a cursor variable is REF CURSOR. Informally, a cursor variable is sometimes called a REF CURSOR.

Topics:

- Purpose of Cursor Variables
- Cursor Variable Creation
- Cursor Variables as Subprogram Parameters

- Cursor Variable Control (OPEN FOR, FETCH, and CLOSE Statements)
- Cursor Variables as Host Variables
- Cursor Variable Restrictions

## Purpose of Cursor Variables

You use cursor variables to pass query result sets between PL/SQL stored subprograms and their clients. This is possible because PL/SQL and its clients share a pointer to the work area where the result set is stored.

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, then pass it as an input host variable (bind variable) to PL/SQL. Application development tools such as Oracle Forms, which have a PL/SQL engine, can use cursor variables entirely on the client side. Or, you can pass cursor variables back and forth between a client and the database server through remote subprogram calls.

A work area remains accessible while any cursor variable points to it, even if you pass the value of a cursor variable from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block embedded in a Pro*C program, the work area to which the cursor variable points remains accessible after the block completes.

If you have a PL/SQL engine on the client side, calls from client to server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, and continue to fetch from it back on the client side. You can also reduce network traffic by having a PL/SQL block open or close several host cursor variables in a single round trip.

Cursor variables are also useful for passing query results between subprograms. Opening the cursor variable in one subprogram and processing it in a different subprogram helps to centralize data retrieval. This technique is also useful for multilanguage applications, where a PL/SQL subprogram might return a result set to a subprogram written in a different language, such as Java or Visual Basic.

## Cursor Variable Creation

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

You can define `REF CURSOR` types in any PL/SQL block, subprogram, or package. In this example, you declare a `REF CURSOR` type that represents a result set from the `DEPARTMENTS` table:

```
DECLARE
  TYPE DeptCurTyp IS REF CURSOR RETURN departments%ROWTYPE;
BEGIN
  NULL;
END;
/
```

`REF CURSOR` types can be strong (with a return type) or weak (with no return type). Strong `REF CURSOR` types are less error-prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with queries that return the right set of columns. Weak `REF CURSOR` types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query. Because there is no type checking with a weak `REF CURSOR`, all such types are interchangeable. Instead of creating a type, you can use the predefined type `SYS_REFCURSOR`.

If both cursor variables involved in an assignment are strongly typed, they must have the same data type (not just the same return type). If one or both cursor variables are weakly typed, they can have different data types.

After defining a REF CURSOR type, you can declare cursor variables of that type in any PL/SQL block or subprogram.

***Example 6–24 Cursor Variable Declarations***

```
DECLARE
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;  -- strong
  TYPE genericcurtyp IS REF CURSOR;  -- weak

  cursor1  empcurtyp;
  cursor2  genericcurtyp;
  my_cursor SYS_REFCURSOR;  -- no new type needed
  TYPE deptcurtyp IS REF CURSOR RETURN departments%ROWTYPE;
  dept_cv deptcurtyp;  -- declare cursor variable
BEGIN
  NULL;
END;
/
```

To avoid declaring the same REF CURSOR type in each subprogram that uses it, you can put the REF CURSOR declaration in a package specification. You can declare cursor variables of that type in the corresponding package body, or in your own subprogram.

In the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to refer to a strongly typed cursor variable, as shown in Example 6–25.

***Example 6–25 Cursor Variables Returning %ROWTYPE Variables***

```
DECLARE
  TYPE TmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
  tmp_cv TmpCurTyp;

  TYPE EmpCurTyp IS REF CURSOR RETURN tmp_cv%ROWTYPE;
  emp_cv EmpCurTyp;
BEGIN
  NULL;
END;
/
```

You can also use %ROWTYPE to provide the data type of a record variable, as shown in Example 6–26.

***Example 6–26 %ROWTYPE Attribute Provides Data Type***

```
DECLARE
  dept_rec departments%ROWTYPE;  -- record variable
  TYPE DeptCurTyp IS REF CURSOR RETURN dept_rec%TYPE;
  dept_cv DeptCurTyp;  -- cursor variable
BEGIN
  NULL;
END;
/
```

Example 6–27 specifies a user-defined RECORD type in the RETURN clause.

***Example 6–27 Cursor Variable Returning a Record Type***

```
DECLARE
  TYPE EmpRecTyp IS RECORD (
    employee_id NUMBER,
    last_name VARCHAR2(25),
    salary   NUMBER(8,2));

  TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
  emp_cv EmpCurTyp;
BEGIN
  NULL;
END;
/
```

## Cursor Variables as Subprogram Parameters

You can declare cursor variables as the formal parameters of subprograms.

When you declare a cursor variable as the formal parameter of a subprogram that fetches from the cursor variable, you must specify the IN or IN OUT mode. However, if the subprogram also opens the cursor variable, you must specify the IN OUT mode.

Example 6–28 defines a REF CURSOR type and then declares a cursor variable of that type as a formal parameter.

***Example 6–28 Cursor Variable as Parameter***

```
DECLARE
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  emp  empcurtyp;

  /* After building result set,
     process all rows inside a single procedure,
     rather than invoking a procedure for each row. */

  PROCEDURE process_emp_cv (emp_cv IN empcurtyp) IS
    person employees%ROWTYPE;
  BEGIN
    DBMS_OUTPUT.PUT_LINE ('-----');
    DBMS_OUTPUT.PUT_LINE ('Here are the names from the result set:');

    LOOP
      FETCH emp_cv INTO person;
      EXIT WHEN emp_cv%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE ('Name = ' || person.first_name ||
                            ' ' || person.last_name);
    END LOOP;
  END;
BEGIN
  -- Find 10 arbitrary employees.
  OPEN emp FOR SELECT * FROM employees WHERE ROWNUM < 11;
  process_emp_cv(emp);
  CLOSE emp;

  -- Find employees matching a condition.
  OPEN emp FOR SELECT * FROM employees WHERE last_name LIKE 'R%';
  process_emp_cv(emp);
  CLOSE emp;
END;
/
```

Result:

```
-----
Here are the names from the result set:
Name = Steven King
Name = Neena Kochhar
Name = Lex De Haan
Name = Alexander Hunold
Name = Bruce Ernst
Name = David Austin
Name = Valli Pataballa
Name = Diana Lorentz
Name = Nancy Greenberg
Name = Daniel Faviet
-----
Here are the names from the result set:
Name = Trenna Rajs
Name = Den Raphaely
Name = Michael Rogers
Name = John Russell
```

Like all pointers, cursor variables increase the possibility of parameter aliasing. See "Overloaded Subprograms" on page 8-13.

Be careful when passing cursor variables as parameters. At run time, PL/SQL raises ROWTYPE_MISMATCH if the return types of the actual and formal parameters are incompatible.

## Cursor Variable Control (OPEN FOR, FETCH, and CLOSE Statements)

You use three statements to control a cursor variable: OPEN FOR, FETCH, and CLOSE. First, you OPEN a cursor variable FOR a multiple-row query. Then, you FETCH rows from the result set. When all the rows are processed, you CLOSE the cursor variable.

Topics:

- Opening Cursor Variables
- Fetching from Cursor Variables
- Closing Cursor Variables
- INVALID_CURSOR Exception

### Opening Cursor Variables

The OPEN FOR statement associates a cursor variable with a multiple-row query, runs the query, and identifies the result set. The cursor variable can be declared in either PL/SQL or a PL/SQL host environment, such as an OCI program.

Unlike cursors, cursor variables do not take parameters. Instead, you can pass whole queries (not just parameters) to a cursor variable. Although a PL/SQL stored subprogram can open a cursor variable and pass it back to a calling subprogram, the calling and called subprograms must be in the same instance. You cannot pass or return cursor variables to procedures and functions called through database links. When you declare a cursor variable as the formal parameter of a subprogram that opens the cursor variable, you must specify the IN OUT mode. That way, the subprogram can pass an open cursor back to the caller.

For the syntax of the OPEN FOR statement, see "OPEN FOR Statement" on page 13-98. This section explains the static SQL case, which uses *select_statement*. For the

dynamic SQL case, which uses *dynamic_string*, see "OPEN FOR Statement" on page 13-98.

The SELECT statement for the query can be coded directly in the statement, or can be a string variable or string literal. When you use a string as the query, it can include placeholders for bind variables, and you specify the corresponding values with a USING clause.

Example 6–29 opens a cursor variable.

***Example 6–29 Opening Cursor Variable***

```
DECLARE
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  emp_cv empcurtyp;
BEGIN
  IF NOT emp_cv%ISOPEN THEN
    OPEN emp_cv FOR SELECT * FROM employees;
  END IF;
  CLOSE emp_cv;
END;
/
```

Other OPEN FOR statements can open the same cursor variable for different queries. You need not close a cursor variable before reopening it. Consecutive OPENs of a static cursor raise the predefined exception CURSOR_ALREADY_OPEN. When you reopen a cursor variable for a different query, the previous query is lost.

Typically, you open a cursor variable by passing it to a stored subprogram that declares an IN OUT parameter that is a cursor variable, as in Example 6–30.

***Example 6–30 Stored Procedure to Open a Cursor Variable***

```
CREATE OR REPLACE PACKAGE emp_data AS
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp);
END emp_data;
/
CREATE OR REPLACE PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS
  BEGIN
    OPEN emp_cv FOR SELECT * FROM employees;
  END open_emp_cv;
END emp_data;
/
```

You can also use a standalone stored subprogram to open the cursor variable. Define the REF CURSOR type in a package, then reference that type in the parameter declaration for the stored subprogram.

To centralize data retrieval, you can group type-compatible queries in a stored subprogram. In Example 6–31, the packaged subprogram declares a selector as one of its formal parameters. When invoked, the subprogram opens the cursor variable emp_cv for the chosen query.

***Example 6–31 Stored Procedure to Open Cursor Variables with Different Queries***

```
CREATE OR REPLACE PACKAGE emp_data AS
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp, choice INT);
END emp_data;
```

```
/
CREATE OR REPLACE PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp, choice INT) IS
  BEGIN
    IF choice = 1 THEN
      OPEN emp_cv FOR SELECT *
      FROM employees
      WHERE commission_pct IS NOT NULL;
    ELSIF choice = 2 THEN
      OPEN emp_cv FOR SELECT *
      FROM employees
      WHERE salary > 2500;
    ELSIF choice = 3 THEN
      OPEN emp_cv FOR SELECT *
      FROM employees
      WHERE department_id = 100;
    END IF;
  END;
END emp_data;
/
```

For more flexibility, a stored subprogram can run queries with different return types, shown in Example 6–32.

***Example 6–32 Cursor Variable with Different Return Types***

```
CREATE OR REPLACE PACKAGE admin_data AS
  TYPE gencurtyp IS REF CURSOR;
  PROCEDURE open_cv (generic_cv IN OUT gencurtyp, choice INT);
END admin_data;
/
CREATE OR REPLACE PACKAGE BODY admin_data AS
  PROCEDURE open_cv (generic_cv IN OUT gencurtyp, choice INT) IS
  BEGIN
    IF choice = 1 THEN
      OPEN generic_cv FOR SELECT * FROM employees;
    ELSIF choice = 2 THEN
      OPEN generic_cv FOR SELECT * FROM departments;
    ELSIF choice = 3 THEN
      OPEN generic_cv FOR SELECT * FROM jobs;
    END IF;
  END;
END admin_data;
/
```

## Fetching from Cursor Variables

The `FETCH` statement retrieves rows from the result set of a multiple-row query. It works the same with cursor variables as with explicit cursors. Example 6–33 fetches rows one at a time from a cursor variable into a record.

***Example 6–33 Fetching from Cursor Variable into Record***

```
DECLARE
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  emp_cv empcurtyp;
  emp_rec employees%ROWTYPE;
BEGIN
  OPEN emp_cv FOR SELECT * FROM employees WHERE employee_id < 120;
  LOOP
```

```
    FETCH emp_cv INTO emp_rec; -- fetch from cursor variable
    EXIT WHEN emp_cv%NOTFOUND; -- exit when last row is fetched
    -- process data record
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || emp_rec.first_name || ' ' || emp_rec.last_name);
  END LOOP;
  CLOSE emp_cv;
END;
/
```

Result:

```
Name = Steven King
Name = Neena Kochhar
Name = Lex De Haan
...
Name = Karen Colmenares
```

Using the BULK COLLECT clause, you can bulk fetch rows from a cursor variable into one or more collections as shown in Example 6–34.

***Example 6–34 Fetching from Cursor Variable into Collections***

```
DECLARE
  TYPE empcurtyp IS REF CURSOR;
  TYPE namelist IS TABLE OF employees.last_name%TYPE;
  TYPE sallist IS TABLE OF employees.salary%TYPE;
  emp_cv  empcurtyp;
  names   namelist;
  sals    sallist;
BEGIN
  OPEN emp_cv FOR
    SELECT last_name, salary FROM employees
    WHERE job_id = 'SA_REP';

  FETCH emp_cv BULK COLLECT INTO names, sals;
  CLOSE emp_cv;
  -- loop through the names and sals collections
  FOR i IN names.FIRST .. names.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || names(i) || ', salary = ' || sals(i));
  END LOOP;
END;
/
```

Result:

```
Name = Tucker, salary = 10000
Name = Bernstein, salary = 9500
Name = Hall, salary = 9000
...
Name = Johnson, salary = 6200
```

Any variables in the associated query are evaluated only when the cursor variable is opened. To change the result set or the values of variables in the query, reopen the cursor variable with the variables set to new values. You can use a different INTO clause on separate fetches with the same cursor variable. Each fetch retrieves another row from the same result set.

PL/SQL makes sure the return type of the cursor variable is compatible with the INTO clause of the FETCH statement. If there is a mismatch, an error occurs at compile time if the cursor variable is strongly typed, or at run time if it is weakly typed. At run time, PL/SQL raises the predefined exception ROWTYPE_MISMATCH before the first fetch. If you trap the exception and run the FETCH statement using a different (compatible) INTO clause, no rows are lost.

When declaring a cursor variable as the formal parameter of a subprogram that fetches from or closes the cursor variable, you must specify the IN or IN OUT mode. If the subprogram opens the cursor variable, you must specify the IN OUT mode.

If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

### Closing Cursor Variables

The CLOSE statement disables a cursor variable and makes the associated result set undefined. Close the cursor variable after the last row is processed.

When declaring a cursor variable as the formal parameter of a subprogram that closes the cursor variable, you must specify the IN or IN OUT mode. If you try to close an already-closed or never-opened cursor variable, PL/SQL raises the predefined exception INVALID_CURSOR.

### INVALID_CURSOR Exception

If you try to fetch from, close, or refer to attributes of a cursor variable that does not point to a query work area, PL/SQL raises the INVALID_CURSOR exception. You can make a cursor variable or cursor subprogram parameter point to a query work area in two ways:

- OPEN the cursor variable for the query.
- Assign to the cursor variable the value of an open host cursor variable or open PL/SQL cursor variable.

If you assign an unopened cursor variable to another cursor variable, the second one remains invalid even after you open the first one.

## Cursor Variables as Host Variables

You can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program. To use the cursor variable, you must pass it as a host variable to PL/SQL.

Example 6–35, a Pro*C program passes a host cursor variable and selector to a PL/SQL block, which opens the cursor variable for the chosen query.

***Example 6–35 Cursor Variable as Host Variable***

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL_CURSOR  generic_cv;  -- host cursor variable
  int         choice;
EXEC SQL END DECLARE SECTION;
EXEC SQL ALLOCATE :generic_cv;  -- Initialize host cursor variable.
-- Pass host cursor variable and selector to PL/SQL block.
/
EXEC SQL EXECUTE
BEGIN
  IF :choice = 1 THEN
    OPEN :generic_cv FOR SELECT * FROM employees;
```

```
  ELSIF :choice = 2 THEN
    OPEN :generic_cv FOR SELECT * FROM departments;
  ELSIF :choice = 3 THEN
    OPEN :generic_cv FOR SELECT * FROM jobs;
  END IF;
END;
END-EXEC;
```

Host cursor variables are compatible with any query return type. They act like weakly typed PL/SQL cursor variables.

When passing host cursor variables to PL/SQL, you can reduce network traffic by grouping `OPEN FOR` statements, as in Example 6–36, where the PL/SQL block opens multiple cursor variables in a single round trip.

***Example 6–36 Reducing Network Traffic When Passing Host Variables***

```
/* anonymous PL/SQL block in host environment */
BEGIN
  OPEN :emp_cv FOR SELECT * FROM employees;
  OPEN :dept_cv FOR SELECT * FROM departments;
  OPEN :loc_cv FOR SELECT * FROM locations;
END;
/
```

This technique can be useful when you want to populate a multiblock form, as in in Oracle Forms. When you pass host cursor variables to a PL/SQL block for opening, the query work areas to which they point remain accessible after the block completes, so your OCI or Pro*C program can use these work areas for ordinary cursor operations. For example, you open several such work areas in a single round trip:

```
BEGIN
  OPEN :c1 FOR SELECT 1 FROM DUAL;
  OPEN :c2 FOR SELECT 1 FROM DUAL;
  OPEN :c3 FOR SELECT 1 FROM DUAL;
END;
/
```

The cursors assigned to `c1`, `c2`, and `c3` act normally, and you can use them for any purpose. When finished, release the cursors as follows:

```
BEGIN
  CLOSE :c1; CLOSE :c2; CLOSE :c3;
END;
/
```

## Cursor Variable Restrictions

Cursor variables are subject to these restrictions:

- You cannot declare cursor variables in a package specification, as Example 6–37 shows.
- If you bind a host cursor variable into PL/SQL from an Oracle Call Interface (OCI) client, you cannot fetch from it on the server side unless you also open it there on the same server call.
- You cannot use comparison operators to test cursor variables for equality, inequality, or nullity.

- Database columns cannot store the values of cursor variables. There is no equivalent type to use in a CREATE TABLE statement.
- You cannot store cursor variables in a collection.
- Cursors and cursor variables are not interchangeable (you cannot use one where the other is expected). For example, you cannot reference a cursor variable in a cursor FOR LOOP.
- A cursor variable is permitted in a server-to-server RPC only if the remote database is not an Oracle Database accessed through a Procedural Gateway.

> **Note:** LOB parameters are not permitted in a server-to-server RPC.

***Example 6–37 Declaration of Cursor Variables in a Package***

```
CREATE OR REPLACE PACKAGE emp_data AS
  TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
  -- emp_cv EmpCurTyp; -- not allowed
  PROCEDURE open_emp_cv;
END emp_data;
/
CREATE OR REPLACE PACKAGE BODY emp_data AS
  -- emp_cv EmpCurTyp; -- not allowed
  PROCEDURE open_emp_cv IS
    emp_cv EmpCurTyp; -- this is legal
  BEGIN
    OPEN emp_cv FOR SELECT * FROM employees;
  END open_emp_cv;
END emp_data;
/
```

# Cursor Expressions

A cursor expression returns a nested cursor. Each row in the result set can contain values, as usual, and cursors produced by subqueries involving the other values in the row. A single query can return a large set of related values retrieved from multiple tables. You can process the result set with nested loops that fetch first from the rows of the result set, and then from any nested cursors in those rows.

A cursor expression has this syntax:

CURSOR(*subquery*)

A nested cursor opens when the containing row is fetched from the parent cursor.

A nested cursor closes only when one of the following happens:

- A user explicitly closes the nested cursor.
- The parent cursor is rerun, closed, or canceled.
- A fetch on one of its parent cursors causes an exception.

In Example 6–38, the cursor c1 is associated with a query that includes a cursor expression. For each department in the departments table, the nested cursor returns the last name of each employee in that department (which it retrieves from the employees table).

***Example 6–38 Cursor Expression***

```
DECLARE
```

```
  TYPE emp_cur_typ IS REF CURSOR;

  emp_cur    emp_cur_typ;
  dept_name  departments.department_name%TYPE;
  emp_name   employees.last_name%TYPE;

  CURSOR c1 IS
    SELECT department_id,
      CURSOR ( SELECT e.last_name
                FROM employees e
                WHERE e.department_id = d.department_id
              ) employees
    FROM departments d
    WHERE department_name LIKE 'A%';
BEGIN
  OPEN c1;
  LOOP  -- Process each row of query result set
    FETCH c1 INTO dept_name, emp_cur;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Department: ' || dept_name);

    LOOP -- Process each row of subquery result set
      FETCH emp_cur INTO emp_name;
      EXIT WHEN emp_cur%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE('-- Employee: ' || emp_name);
    END LOOP;
  END LOOP;
  CLOSE c1;
END;
/
```

Result:

```
Department: 10
-- Employee: Whalen
Department: 110
-- Employee: Higgins
-- Employee: Gietz
```

You can pass a cursor expression to a function that has a formal parameter of the type REF CURSOR. Using a cursor expression as an actual parameter lets you pass the function a set of rows. Cursor expressions are often passed as parameters to pipelined table functions, as explained in "Passing Data with Cursor Variables" on page 12-46.

### Cursor Expression Restrictions

- You cannot use a cursor expression with an implicit cursor.
- Cursor expressions can appear only:
  - In a SELECT statement that is not nested in any other query expression, unless it is a subquery of the cursor expression itself.
  - As arguments to table functions, in the FROM clause of a SELECT statement.
- Cursor expressions can appear only in the outermost SELECT list of the query specification.
- Cursor expressions cannot appear in view declarations.
- You cannot perform BIND and EXECUTE operations on cursor expressions.

## Transaction Processing and Control

**Transaction processing** is an Oracle Database feature that enables multiple users to work on the database concurrently, and ensures that each user sees a consistent version of data and that all changes are applied in the right order. Different users can write to the same data structures without harming each other's data or coordinating with each other, because Oracle Database locks data structures automatically. To maximize data availability, Oracle Database locks the minimum amount of data for the minimum amount of time.

You rarely must write extra code to prevent problems with multiple users accessing data concurrently. However, if you do need this level of control, you can request locks on tables or rows.

You can control transactions with the SQL transaction control statements `COMMIT`, `SAVEPOINT`, `ROLLBACK`, and `SET TRANSACTION`.

Explicitly end every transaction, using either the `COMMIT` or `ROLLBACK` statement. Otherwise, the client environment determines its final state. For example, in the SQL*Plus environment, if your PL/SQL block does not include a `COMMIT` or `ROLLBACK` statement, the final state of your transaction depends on what you do after running the block. If you run a data definition, data control, or `COMMIT` statement or if you issue the `EXIT`, `DISCONNECT`, or `QUIT` statement, the database commits the transaction. If you run a `ROLLBACK` statement or stop the SQL*Plus session, the database rolls back the transaction.

Whether you end the transaction in your PL/SQL program or from a client program depends on the application logic.

Oracle Database uses transactions to ensure data integrity. A **transaction** is a series of DML statements that does a logical unit of work. For example, two `UPDATE` statements might credit one bank account and debit another. It is important not to allow one operation to succeed while the other fails. A transaction can span multiple blocks, and a block can contain multiple transactions.

At the end of a transaction, the database either commits (makes permanent) or rolls back (undoes) all changes in the transaction. If your program fails in the middle of a transaction, the database detects the failure and rolls back the transaction, restoring the database to its former state.

The SQL transaction control statements are `COMMIT`, `ROLLBACK`, `SAVEPOINT`, and `SET TRANSACTION` statements. `COMMIT` makes permanent any database changes made during the current transaction. `ROLLBACK` ends the current transaction and undoes any changes made since the transaction began. `SAVEPOINT` marks the current point in the processing of a transaction. Used with `ROLLBACK`, `SAVEPOINT` undoes part of a transaction. `SET TRANSACTION` sets transaction properties such as read/write access and isolation level. See "Transaction Processing and Control" on page 6-37.

Topics:

- COMMIT Statement
- ROLLBACK Statement
- SAVEPOINT Statement
- Implicit Rollbacks
- SET TRANSACTION Statement
- Overriding Default Locking

**See Also:**

- *Oracle Database Concepts* for information about transactions
- *Oracle Database SQL Language Reference* for information about the `COMMIT` statement
- *Oracle Database SQL Language Reference* for information about the `SAVEPOINT` statement
- *Oracle Database SQL Language Reference* for information about the `ROLLBACK` statement

## COMMIT Statement

The `COMMIT` statement ends the current transaction, making any changes made during that transaction permanent and visible to other users. Transactions are not tied to PL/SQL `BEGIN` `END` blocks. A block can contain multiple transactions, and a transaction can span multiple blocks.

Asynchronous commit provides more control for the user with the `WRITE` clause. This option specifies the priority with which the redo information generated by the commit operation is written to the redo log.

In Example 6–39, a transaction transfers money from one bank account to another. It is important that the money both leaves one account and enters the other, hence the `COMMIT` `WRITE` `IMMEDIATE` `NOWAIT` statement.

***Example 6–39 COMMIT Statement with COMMENT and WRITE Clauses***

```
DROP TABLE accounts;
CREATE TABLE accounts (
  account_id  NUMBER(6),
  balance     NUMBER (10,2)
);

INSERT INTO accounts (account_id, balance)
VALUES (7715, 6350.00);

INSERT INTO accounts (account_id, balance)
VALUES (7720, 5100.50);

CREATE OR REPLACE PROCEDURE transfer (
  from_acct  NUMBER,
  to_acct    NUMBER,
  amount     NUMBER
) AUTHID DEFINER AS
BEGIN
  UPDATE accounts
  SET balance = balance - amount
  WHERE account_id = from_acct;

  UPDATE accounts
  SET balance = balance + amount
  WHERE account_id = to_acct;

  COMMIT WRITE IMMEDIATE NOWAIT;
END;
/
```

Query before transfer:

```
SELECT * FROM accounts;
```

Result:

```
ACCOUNT_ID    BALANCE
---------- ----------
      7715       6350
      7720     5100.5

BEGIN
  transfer(7715, 7720, 250);
END;
/
```

Query after transfer:

```
SELECT * FROM accounts;
```

Result:

```
ACCOUNT_ID    BALANCE
---------- ----------
      7715       6100
      7720     5350.5
```

**Note:** The default PL/SQL commit behavior for nondistributed transactions is BATCH NOWAIT if the COMMIT_LOGGING and COMMIT_WAIT database initialization parameters have not been set.

**See Also:**

- *Oracle Database Advanced Application Developer's Guide* for more information about committing transactions
- *Oracle Database Concepts* for information about distributed transactions
- *Oracle Database SQL Language Reference*for information about the COMMIT statement
- *Oracle Data Guard Concepts and Administration* for information about ensuring no loss of data during a failover to a standby database

## ROLLBACK Statement

The ROLLBACK statement ends the current transaction and undoes any changes made during that transaction. If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because an exception is raised or a SQL statement fails, a rollback lets you take corrective action and perhaps start over.

Example 6–40 inserts information about an employee into three different tables. If an INSERT statement tries to store a duplicate employee number, the predefined exception DUP_VAL_ON_INDEX is raised. To ensure that changes to all three tables are undone, the exception handler runs a ROLLBACK.

***Example 6–40 ROLLBACK Statement***

```
DROP TABLE emp_name;
```

```
CREATE TABLE emp_name AS
  SELECT employee_id, last_name
  FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);


DROP TABLE emp_sal;
CREATE TABLE emp_sal AS
  SELECT employee_id, salary
  FROM employees;

CREATE UNIQUE INDEX empsal_ix
ON emp_sal (employee_id);


DROP TABLE emp_job;
CREATE TABLE emp_job AS
  SELECT employee_id, job_id
  FROM employees;

CREATE UNIQUE INDEX empjobid_ix
ON emp_job (employee_id);


DECLARE
  emp_id        NUMBER(6);
  emp_lastname  VARCHAR2(25);
  emp_salary    NUMBER(8,2);
  emp_jobid     VARCHAR2(10);
BEGIN
  SELECT employee_id, last_name, salary, job_id
  INTO emp_id, emp_lastname, emp_salary, emp_jobid
  FROM employees
  WHERE employee_id = 120;

  INSERT INTO emp_name (employee_id, last_name)
  VALUES (emp_id, emp_lastname);

  INSERT INTO emp_sal (employee_id, salary)
  VALUES (emp_id, emp_salary);

  INSERT INTO emp_job (employee_id, job_id)
  VALUES (emp_id, emp_jobid);

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK;
    DBMS_OUTPUT.PUT_LINE('Inserts were rolled back');
END;
/
```

**See Also:** *Oracle Database SQL Language Reference* for more information about the ROLLBACK statement

## SAVEPOINT Statement

The SAVEPOINT statement names and marks the current point in the processing of a transaction. Savepoints let you roll back part of a transaction instead of the whole transaction. The number of active savepoints for each session is unlimited.

Example 6–41 marks a savepoint before doing an insert. If the INSERT statement tries to store a duplicate value in the employee_id column, the predefined exception DUP_VAL_ON_INDEX is raised and the transaction rolls back to the savepoint, undoing only the INSERT statement.

***Example 6–41 SAVEPOINT and ROLLBACK Statements***

```
DROP TABLE emp_name;
CREATE TABLE emp_name AS
  SELECT employee_id, last_name, salary
  FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);

DECLARE
  emp_id        employees.employee_id%TYPE;
  emp_lastname  employees.last_name%TYPE;
  emp_salary    employees.salary%TYPE;

BEGIN
  SELECT employee_id, last_name, salary
  INTO emp_id, emp_lastname, emp_salary
  FROM employees
  WHERE employee_id = 120;

  UPDATE emp_name
  SET salary = salary * 1.1
  WHERE employee_id = emp_id;

  DELETE FROM emp_name
  WHERE employee_id = 130;

  SAVEPOINT do_insert;

  INSERT INTO emp_name (employee_id, last_name, salary)
  VALUES (emp_id, emp_lastname, emp_salary);

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK TO do_insert;
  DBMS_OUTPUT.PUT_LINE('Insert was rolled back');
END;
/
```

When you roll back to a savepoint, any savepoints marked after that savepoint are erased. The savepoint to which you roll back is not erased. A simple rollback or commit erases all savepoints.

If you mark a savepoint in a recursive subprogram, new instances of the SAVEPOINT statement run at each level in the recursive descent, but you can only roll back to the most recently marked savepoint.

Savepoint names are undeclared identifiers. Reusing a savepoint name in a transaction moves the savepoint from its old position to the current point in the transaction, which means that a rollback to the savepoint affects only the current part of the transaction.

***Example 6–42 Reusing a SAVEPOINT with ROLLBACK***

```
DROP TABLE emp_name;
CREATE TABLE emp_name AS
  SELECT employee_id, last_name, salary
  FROM employees;

CREATE UNIQUE INDEX empname_ix
ON emp_name (employee_id);

DECLARE
  emp_id        employees.employee_id%TYPE;
  emp_lastname  employees.last_name%TYPE;
  emp_salary    employees.salary%TYPE;

BEGIN
  SELECT employee_id, last_name, salary
  INTO emp_id, emp_lastname, emp_salary
  FROM employees
  WHERE employee_id = 120;

  SAVEPOINT my_savepoint;

  UPDATE emp_name
  SET salary = salary * 1.1
  WHERE employee_id = emp_id;

  DELETE FROM emp_name
  WHERE employee_id = 130;

  SAVEPOINT my_savepoint;

  INSERT INTO emp_name (employee_id, last_name, salary)
  VALUES (emp_id, emp_lastname, emp_salary);

EXCEPTION
  WHEN DUP_VAL_ON_INDEX THEN
    ROLLBACK TO my_savepoint;
    DBMS_OUTPUT.PUT_LINE('Transaction rolled back.');
END;
/
```

**See Also:** *Oracle Database SQL Language Reference* for more information about the SET TRANSACTION SQL statement

## Implicit Rollbacks

Before running an INSERT, UPDATE, or DELETE statement, the database marks an implicit savepoint (unavailable to you). If the statement fails, the database rolls back to the savepoint. Usually, just the failed SQL statement is rolled back, not the whole transaction. If the statement raises an unhandled exception, the host environment determines what is rolled back.

The database can also roll back single SQL statements to break deadlocks. The database signals an error to a participating transaction and rolls back the current statement in that transaction.

Before running a SQL statement, the database must parse it, that is, examine it to ensure it follows syntax rules and refers to valid schema objects. Errors detected while running a SQL statement cause a rollback, but errors detected while parsing the statement do not.

If you exit a stored subprogram with an unhandled exception, PL/SQL does not assign values to `OUT` parameters, and does not do any rollback.

## SET TRANSACTION Statement

You use the `SET TRANSACTION` statement to begin a read-only or read-write transaction, establish an isolation level, or assign your current transaction to a specified rollback segment. Read-only transactions are useful for running multiple queries while other users update the same tables.

During a read-only transaction, all queries refer to the same snapshot of the database, providing a multi-table, multi-query, read-consistent view. Other users can continue to query or update data as usual. A commit or rollback ends the transaction.

In Example 6–43 a read-only transaction gather order totals for the day, the past week, and the past month. The totals are unaffected by other users updating the database during the transaction. The `orders` table is in the sample schema OE.

***Example 6–43 SET TRANSACTION Statement in Read-Only Transaction***

```
DECLARE
  daily_order_total    NUMBER(12,2);
  weekly_order_total   NUMBER(12,2);
  monthly_order_total  NUMBER(12,2);
BEGIN
   COMMIT; -- end previous transaction
   SET TRANSACTION READ ONLY NAME 'Calculate Order Totals';

   SELECT SUM (order_total)
   INTO daily_order_total
   FROM orders
   WHERE order_date = SYSDATE;

   SELECT SUM (order_total)
   INTO weekly_order_total
   FROM orders
   WHERE order_date = SYSDATE - 7;

   SELECT SUM (order_total)
   INTO monthly_order_total
   FROM orders
   WHERE order_date = SYSDATE - 30;

   COMMIT; -- ends read-only transaction
END;
/
```

The `SET TRANSACTION` statement must be the first SQL statement in a read-only transaction and can only appear once in a transaction. If you set a transaction to `READ ONLY`, subsequent queries see only changes committed before the transaction began. The use of `READ ONLY` does not affect other users or transactions.

Only the `SELECT`, `OPEN`, `FETCH`, `CLOSE`, `LOCK TABLE`, `COMMIT`, and `ROLLBACK` statements are allowed in a read-only transaction. Queries cannot be `FOR UPDATE`.

**See Also:** *Oracle Database SQL Language Reference* for more information about the SQL statement `SET TRANSACTION`

## Overriding Default Locking

By default, Oracle Database locks data structures automatically, which enables different applications to write to the same data structures without harming each other's data or coordinating with each other.

If you must have exclusive access to data during a transaction, you can override default locking with these SQL statements:

- `LOCK TABLE`, which explicitly locks entire tables.
- `SELECT` with the `FOR UPDATE` clause (`SELECT FOR UPDATE`) , which explicitly locks specific rows of a table.

Topics:

- LOCK TABLE Statement
- SELECT FOR UPDATE and FOR UPDATE Cursors
- Simulating CURRENT OF Clause with ROWID Pseudocolumn

### LOCK TABLE Statement

The `LOCK TABLE` statement explicitly locks one or more tables in a specified lock mode so that you can share or deny access to them.

The lock mode determines what other locks can be placed on the table. For example, many users can acquire row share locks on a table at the same time, but only one user at a time can acquire an exclusive lock. While one user has an exclusive lock on a table, no other users can insert, delete, or update rows in that table.

A table lock never prevents other users from querying a table, and a query never acquires a table lock. Only if two different transactions try to modify the same row does one transaction wait for the other to complete. The `LOCK TABLE` statement lets you specify how long to wait for another transaction to complete.

Table locks are released when the transaction that acquired them is either committed or rolled back.

**See Also:**

- *Oracle Database Advanced Application Developer's Guide* for more information about locking tables explicitly
- *Oracle Database SQL Language Reference* for more information about the `LOCK TABLE` statement

### SELECT FOR UPDATE and FOR UPDATE Cursors

The `SELECT` statement with the `FOR UPDATE` clause (`SELECT FOR UPDATE`) selects the rows of the result set and locks them. `SELECT FOR UPDATE` enables you to base an update on the existing values in the rows, because it ensures that no other user can change those values before you update them. You can also use `SELECT FOR UPDATE` to lock rows that you do not want to update, as in Example 9–20.

By default, the `SELECT FOR UPDATE` statement waits until the requested row lock is acquired. To change this behavior, use the `NOWAIT`, `WAIT`, or `SKIP LOCKED` clause of the `SELECT FOR UPDATE` statement. For information about these clauses, see *Oracle Database SQL Language Reference*.

When SELECT FOR UPDATE is associated with an explicit cursor, the cursor is called a **FOR UPDATE cursor**. Only a FOR UPDATE cursor can appear in the CURRENT OF clause of an UPDATE or DELETE statement. (The CURRENT OF clause, a PL/SQL extension to the WHERE clause of the SQL statements UPDATE and DELETE, restricts the statement to the current row of the cursor.)

In Example 6–44, a FOR UPDATE cursor appears in the CURRENT OF clause of an UPDATE statement.

***Example 6–44 FOR UPDATE Cursor in CURRENT OF Clause of UPDATE Statement***

```
DECLARE
  my_emp_id NUMBER(6);
  my_job_id VARCHAR2(10);
  my_sal    NUMBER(8,2);
  CURSOR c1 IS
    SELECT employee_id, job_id, salary
    FROM employees FOR UPDATE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_emp_id, my_job_id, my_sal;
    IF my_job_id = 'SA_REP' THEN
      UPDATE employees
      SET salary = salary * 1.02
      WHERE CURRENT OF c1;
    END IF;
    EXIT WHEN c1%NOTFOUND;
  END LOOP;
END;
/
```

When SELECT FOR UPDATE queries multiple tables, it locks only rows whose columns appear in the FOR UPDATE clause.

In Example 6–45, SELECT FOR UPDATE queries the tables EMPLOYEES and DEPARTMENTS, but only SALARY appears in the FOR UPDATE clause. SALARY is a column of EMPLOYEES, but not of DEPARTMENTS; therefore, SELECT FOR UPDATE locks only rows of EMPLOYEES. If the FOR UPDATE clause included DEPARTMENT_ID or MANAGER_ID, which are columns of both EMPLOYEES and DEPARTMENTS, SELECT FOR UPDATE would lock rows of both tables.

***Example 6–45 SELECT FOR UPDATE with Multiple Tables***

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, department_name
    FROM employees, departments
    WHERE employees.department_id = departments.department_id
    AND job_id = 'SA_MAN'
    FOR UPDATE OF salary;
BEGIN
  NULL;
END;
/
```

### Simulating CURRENT OF Clause with ROWID Pseudocolumn

The rows of the result set are locked when you open a FOR UPDATE cursor, not as they are fetched. The rows are unlocked when you commit or roll back the transaction.

After the rows are unlocked, you cannot fetch from the FOR UPDATE cursor, as Example 6–46 shows (the result is the same if you substitute ROLLBACK for COMMIT).

***Example 6–46 Trying to Fetch with FOR UPDATE Cursor After COMMIT Statement***

```
DECLARE
  CURSOR c1 IS
    SELECT * FROM employees
    FOR UPDATE OF salary;
  emp_rec  employees%ROWTYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO emp_rec;  -- fails on second iteration
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE (
      'emp_rec.employee_id = ' ||
      TO_CHAR(emp_rec.employee_id)
    );
    IF emp_rec.employee_id = 105 THEN
      UPDATE employees
      SET salary = salary * 1.05
      WHERE employee_id = 105;
    END IF;
    COMMIT;  -- releases locks
  END LOOP;
END;
/
```

Result:

```
emp_rec.employee_id = 100
DECLARE
*
ERROR at line 1:
ORA-01002: fetch out of sequence
ORA-06512: at line 9
```

The workaround is to simulate the CURRENT OF clause with the ROWID pseudocolumn. Select the rowid of each row into a UROWID variable and use the rowid to identify the current row during subsequent updates and deletes, as in Example 6–47.

---

**Caution:** Because no FOR UPDATE clause locks the fetched rows, other users might unintentionally overwrite your changes.

---

**Note:** The extra space needed for read consistency is not released until the cursor is closed, which can slow down processing for large updates.

---

***Example 6–47 Simulating CURRENT OF Clause with ROWID Pseudocolumn***

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id, rowid
    FROM employees;  -- no FOR UPDATE clause
  my_lastname   employees.last_name%TYPE;
  my_jobid      employees.job_id%TYPE;
```

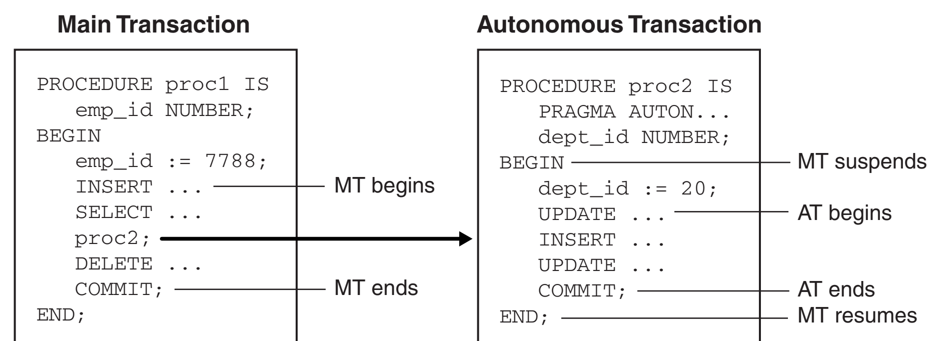
```
  my_rowid      UROWID;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_lastname, my_jobid, my_rowid;
    EXIT WHEN c1%NOTFOUND;

    UPDATE employees
    SET salary = salary * 1.02
    WHERE rowid = my_rowid;  -- simulates WHERE CURRENT OF c1

    COMMIT;
  END LOOP;
  CLOSE c1;
END;
/
```

# Autonomous Transactions

An **autonomous transaction** is an independent transaction started by another transaction, the main transaction. Autonomous transactions do SQL operations and commit or roll back, without committing or rolling back the main transaction. For example, if you write auditing data to a log table, you want to commit the audit data even if the operation you are auditing later fails; if something goes wrong recording the audit data, you do not want the main operation to be rolled back.

Figure 6–1 shows how control flows from the main transaction (MT) to an autonomous transaction (AT) and back again.

***Figure 6–1 Transaction Control Flow***



---

**Note:** Although an autonomous transaction is started by another transaction, it is not a nested transaction, because:

- It does not share transactional resources (such as locks) with the main transaction.
- It does not depend on the main transaction.

  For example, if the main transaction rolls back, nested transactions roll back, but autonomous transactions do not.
- Its committed changes are visible to other transactions immediately.

  A nested transaction's committed changes are not visible to other transactions until the main transaction commits.
- Exceptions raised in an autonomous transaction cause a transaction-level rollback, not a statement-level rollback.

---

Topics:

- Advantages of Autonomous Transactions
- Transaction Context
- Transaction Visibility
- Declaring Autonomous Transactions
- Controlling Autonomous Transactions
- Autonomous Triggers
- Invoking Autonomous Functions from SQL

## Advantages of Autonomous Transactions

Once started, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. You can log events, increment retry counters, and so on, even if the main transaction rolls back.

Autonomous transactions help you build modular, reusable software components. You can encapsulate autonomous transactions in stored subprograms. A calling application needs not know whether operations done by that stored subprogram succeeded or failed.

## Transaction Context

The main transaction shares its context with nested routines, but not with autonomous transactions. When one autonomous routine invokes another (or itself, recursively), the routines share no transaction context. When an autonomous routine invokes a nonautonomous routine, the routines share the same transaction context.

## Transaction Visibility

Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. These changes become visible to the main transaction when it resumes, if its isolation level is set to `READ COMMITTED` (the default).

If you set the isolation level of the main transaction to SERIALIZABLE, changes made by its autonomous transactions are *not* visible to the main transaction when it resumes:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

---

**Note:**

- Transaction properties apply only to the transaction in which they are set.
- Cursor attributes are not affected by autonomous transactions.

---

## Declaring Autonomous Transactions

To declare an autonomous transaction, use the AUTONOMOUS_TRANSACTION pragma. For information about this pragma, see "AUTONOMOUS_TRANSACTION Pragma" on page 13-6.

> **Tip:** For readability, put the AUTONOMOUS_TRANSACTION pragma at the top of the declarative section. (The pragma is allowed anywhere in the declarative section.)

You cannot apply the AUTONOMOUS_TRANSACTION pragma to an entire package, but you can apply it to each subprogram in a package.

Example 6–48 marks a packaged function as autonomous.

***Example 6–48 Declaring an Autonomous Function in a Package***

```
CREATE OR REPLACE PACKAGE emp_actions AS  -- package specification
  FUNCTION raise_salary (emp_id NUMBER, sal_raise NUMBER)
  RETURN NUMBER;
END emp_actions;
/
CREATE OR REPLACE PACKAGE BODY emp_actions AS  -- package body
  -- code for function raise_salary
  FUNCTION raise_salary (emp_id NUMBER, sal_raise NUMBER)
  RETURN NUMBER IS
    PRAGMA AUTONOMOUS_TRANSACTION;
    new_sal NUMBER(8,2);
  BEGIN
    UPDATE employees SET salary =
      salary + sal_raise WHERE employee_id = emp_id;
    COMMIT;
    SELECT salary INTO new_sal FROM employees
      WHERE employee_id = emp_id;
    RETURN new_sal;
  END raise_salary;
END emp_actions;
/
```

Example 6–49 marks a standalone stored subprogram as autonomous.

***Example 6–49 Declaring an Autonomous Standalone Procedure***

```
CREATE OR REPLACE PROCEDURE lower_salary
   (emp_id NUMBER, amount NUMBER)
AS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
```

```
  UPDATE employees
  SET salary =  salary - amount
  WHERE employee_id = emp_id;

  COMMIT;
END lower_salary;
/
```

Example 6–50 marks a schema-level PL/SQL block as autonomous. (A nested PL/SQL block cannot be autonomous.)

***Example 6–50 Declaring an Autonomous PL/SQL Block***

```
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
  emp_id NUMBER(6)   := 200;
  amount NUMBER(6,2) := 200;
BEGIN
  UPDATE employees
  SET salary =  salary - amount
  WHERE employee_id = emp_id;

  COMMIT;
END;
/
```

## Controlling Autonomous Transactions

The first SQL statement in an autonomous routine begins a transaction. When one transaction ends, the next SQL statement begins another transaction. All SQL statements run since the last commit or rollback comprise the current transaction. To control autonomous transactions, use these statements, which apply only to the current (active) transaction:

- `COMMIT`
- `ROLLBACK` [`TO` *`savepoint_name`*]
- `SAVEPOINT` *`savepoint_name`*
- `SET TRANSACTION`

Topics:

- Entering and Exiting
- Committing and Rolling Back
- Savepoints
- Avoiding Errors with Autonomous Transactions

### Entering and Exiting

When you enter the executable section of an autonomous routine, the main transaction suspends. When you exit the routine, the main transaction resumes.

To exit normally, you must explicitly commit or roll back all autonomous transactions. If the routine (or any routine invoked by it) has pending transactions, an exception is raised, and the pending transactions are rolled back.

### Committing and Rolling Back

COMMIT and ROLLBACK end the active autonomous transaction but do not exit the autonomous routine. When one transaction ends, the next SQL statement begins another transaction. A single autonomous routine can contain several autonomous transactions, if it issues several COMMIT statements.

### Savepoints

The scope of a savepoint is the transaction in which it is defined. Savepoints defined in the main transaction are unrelated to savepoints defined in its autonomous transactions. In fact, the main transaction and an autonomous transaction can use the same savepoint names.

You can roll back only to savepoints marked in the current transaction. In an autonomous transaction, you cannot roll back to a savepoint marked in the main transaction. To do so, you must resume the main transaction by exiting the autonomous routine.

When in the main transaction, rolling back to a savepoint marked before you started an autonomous transaction does *not* roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

### Avoiding Errors with Autonomous Transactions

To avoid some common errors, remember:

- If an autonomous transaction attempts to access a resource held by the main transaction, a deadlock can occur. The database raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.
- The database initialization parameter TRANSACTIONS specifies the maximum number of concurrent transactions. That number might be exceeded because an autonomous transaction runs concurrently with the main transaction.
- If you try to exit an active autonomous transaction without committing or rolling back, the database raises an exception. If the exception goes unhandled, the transaction is rolled back.

## Autonomous Triggers

A database trigger can do the following only if it is autonomous:

- Run Transaction Control Language (TCL) statements
- Use native dynamic SQL to run Data Definition Language (DDL) statements

  For information about native dynamic SQL, see "Native Dynamic SQL" on page 7-2.

One use of database triggers is to log events transparently—for example, to log all inserts into a table, even those that roll back. In Example 6–51, whenever a row is inserted into the EMPLOYEES table, a trigger inserts the same row into an audit table. Because the trigger is autonomous, it can commit changes to the audit table regardless of whether they are committed to the main table.

***Example 6–51 Autonomous Trigger the Logs INSERT Statements***

```
-- Audit table:

DROP TABLE emp_audit;
CREATE TABLE emp_audit (
```

```
  emp_audit_id NUMBER(6),
  up_date DATE,
  new_sal NUMBER(8,2),
  old_sal NUMBER(8,2)
);

-- Autonomous trigger on employees table:

CREATE OR REPLACE TRIGGER audit_sal
  BEFORE UPDATE OF salary ON employees FOR EACH ROW
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO emp_audit (
    emp_audit_id,
    up_date,
    new_sal,
    old_sal
  )
  VALUES (
    :old.employee_id,
    SYSDATE,
    :new.salary,
    :old.salary
  );
  COMMIT;
END;
/
UPDATE employees
SET salary = salary * 1.05
WHERE employee_id = 115;

COMMIT;

UPDATE employees
SET salary = salary * 1.05
WHERE employee_id = 116;

ROLLBACK;

-- Show that both committed and rolled-back updates
-- add rows to audit table

SELECT * FROM emp_audit
WHERE emp_audit_id = 115 OR emp_audit_id = 116;
```

Result:

```
EMP_AUDIT_ID UP_DATE      NEW_SAL    OLD_SAL
------------ --------- ---------- ----------
         115 06-MAR-09     3320.1       3162
         116 06-MAR-09     3105.9       2958

2 rows selected.
```

In Example 6–52, an autonomous trigger uses native dynamic SQL (an `EXECUTE IMMEDIATE` statement) to drop a temporary table after a row is inserted into the table `emp_audit`.

***Example 6–52 Autonomous Trigger Using Native Dynamic SQL for DDL***

```
DROP TABLE emp_audit;
CREATE TABLE emp_audit (
  emp_audit_id  NUMBER(6),
  up_date       DATE,
  new_sal       NUMBER(8,2),
  old_sal       NUMBER(8,2)
);
DROP TABLE temp_audit;
CREATE TABLE temp_audit (
  emp_audit_id NUMBER(6),
  up_date DATE
);

CREATE OR REPLACE TRIGGER drop_temp_table
  AFTER INSERT ON emp_audit
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE temp_audit';
  COMMIT;
END;
/
-- Show how trigger works
SELECT * FROM temp_audit;
```

Result:

```
no rows selected

INSERT INTO emp_audit (emp_audit_id, up_date, new_sal, old_sal)
VALUES (999, SYSDATE, 5000, 4500);

SELECT * FROM temp_audit;
```

Result:

```
SELECT * FROM temp_audit
              *
ERROR at line 1:
ORA-00942: table or view does not exist
```

For general information about triggers, see Chapter 9, "PL/SQL Triggers."

## Invoking Autonomous Functions from SQL

A function invoked from SQL statements must obey rules meant to control side effects (for details, see "Subprogram Side Effects" on page 8-27). To check for violations of the rules, use the `RESTRICT_REFERENCES` pragma. This pragma asserts that a function does not read or write database tables or package variables. For more information about this pragma, see *Oracle Database Advanced Application Developer's Guide*.

By definition, autonomous routines never violate the rules `read no database state (RNDS)` and `write no database state (WNDS)`.

The packaged function `log_msg` in Example 6–53 is autonomous. Therefore, when the query invokes the function, the function inserts a message into database table `debug_output` without violating the rule `write no database state`.

***Example 6–53 Invoking an Autonomous Function***

```
DROP TABLE debug_output;
CREATE TABLE debug_output (message VARCHAR2(200));

CREATE OR REPLACE PACKAGE debugging AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2;
  PRAGMA RESTRICT_REFERENCES(log_msg, WNDS, RNDS);
END debugging;
/
CREATE OR REPLACE PACKAGE BODY debugging AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2 IS
    PRAGMA AUTONOMOUS_TRANSACTION;
  BEGIN
    /* This insert does not violate the constraint WNDS
       because this is an autonomous routine */

  INSERT INTO debug_output (message)
  VALUES (msg);

  COMMIT;
  RETURN msg;
  END;
END debugging;
/
-- Invoke packaged function from query
DECLARE
  my_emp_id    NUMBER(6);
  my_last_name VARCHAR2(25);
  my_count     NUMBER;
BEGIN
  my_emp_id := 120;

  SELECT debugging.log_msg(last_name)
  INTO my_last_name
  FROM employees
  WHERE employee_id = my_emp_id;

  /* Even if you roll back in this scope,
     the insert into 'debug_output' remains committed,
     because it is part of an autonomous transaction. */

  ROLLBACK;
END;
/
```

# 7

# PL/SQL Dynamic SQL

**Dynamic SQL** is a programming methodology for generating and running SQL statements at run time. It is useful when writing general-purpose and flexible programs like ad hoc query systems, when writing programs that must run DDL statements, or when you do not know at compilation time the full text of a SQL statement or the number or data types of its input and output variables.

PL/SQL provides two ways to write dynamic SQL:

- Native dynamic SQL, a PL/SQL language (that is, native) feature for building and running dynamic SQL statements
- `DBMS_SQL` package, an API for building, running, and describing dynamic SQL statements

Native dynamic SQL code is easier to read and write than equivalent code that uses the `DBMS_SQL` package, and runs noticeably faster (especially when it can be optimized by the compiler). However, to write native dynamic SQL code, you must know at compile time the number and data types of the input and output variables of the dynamic SQL statement. If you do not know this information at compile time, you must use the `DBMS_SQL` package.

When you need both the `DBMS_SQL` package and native dynamic SQL, you can switch between them, using the "DBMS_SQL.TO_REFCURSOR Function" on page 7-7 and "DBMS_SQL.TO_CURSOR_NUMBER Function" on page 7-8.

Topics:

- When You Need Dynamic SQL
- Native Dynamic SQL
- DBMS_SQL Package
- SQL Injection

## When You Need Dynamic SQL

In PL/SQL, you need dynamic SQL to run:

- SQL whose text is unknown at compile time

  For example, a `SELECT` statement that includes an identifier that is unknown at compile time (such as a table name) or a `WHERE` clause in which the number of subclauses is unknown at compile time.

- SQL that is not supported as static SQL

  That is, any SQL construct not included in "Description of Static SQL" on page 6-1.

If you do not need dynamic SQL, use static SQL, which has these advantages:

- Successful compilation verifies that static SQL statements reference valid database objects and that the necessary privileges are in place to access those objects.
- Successful compilation creates schema object dependencies.

  For information about schema object dependencies, see *Oracle Database Advanced Application Developer's Guide*.

For information about using static SQL statements with PL/SQL, see Chapter 6, "PL/SQL Static SQL."

# Native Dynamic SQL

Native dynamic SQL processes most dynamic SQL statements with the `EXECUTE IMMEDIATE` statement.

If the dynamic SQL statement is a `SELECT` statement that returns multiple rows, native dynamic SQL gives you these choices:

- Use the `EXECUTE IMMEDIATE` statement with the `BULK COLLECT INTO` clause.
- Use the `OPEN FOR`, `FETCH`, and `CLOSE` statements.

The SQL cursor attributes work the same way after native dynamic SQL `INSERT`, `UPDATE`, `DELETE`, and single-row `SELECT` statements as they do for their static SQL counterparts. For more information about SQL cursor attributes, see "Cursors" on page 6-6.

Topics:

- EXECUTE IMMEDIATE Statement
- OPEN FOR, FETCH, and CLOSE Statements
- Repeated Placeholder Names in Dynamic SQL Statements

## EXECUTE IMMEDIATE Statement

The `EXECUTE IMMEDIATE` statement is the means by which native dynamic SQL processes most dynamic SQL statements.

If the dynamic SQL statement is **self-contained** (that is, if it has no placeholders for bind arguments and the only result that it can possibly return is an error), then the `EXECUTE IMMEDIATE` statement needs no clauses.

If the dynamic SQL statement includes placeholders for bind arguments, each placeholder must have a corresponding bind argument in the appropriate clause of the `EXECUTE IMMEDIATE` statement, as follows:

- If the dynamic SQL statement is a `SELECT` statement that can return at most one row, put out-bind arguments (defines) in the `INTO` clause and in-bind arguments in the `USING` clause.
- If the dynamic SQL statement is a `SELECT` statement that can return multiple rows, put out-bind arguments (defines) in the `BULK COLLECT INTO` clause and in-bind arguments in the `USING` clause.
- If the dynamic SQL statement is a DML statement without a `RETURNING INTO` clause, other than `SELECT`, put all bind arguments in the `USING` clause.

- If the dynamic SQL statement is a DML statement with a RETURNING INTO clause, put in-bind arguments in the USING clause and out-bind arguments in the RETURNING INTO clause.
- If the dynamic SQL statement is an anonymous PL/SQL block or a CALL statement, put all bind arguments in the USING clause.

  If the dynamic SQL statement invokes a subprogram, ensure that:

  - Every bind argument that corresponds to a placeholder for a subprogram parameter has the same parameter mode as that subprogram parameter (as in Example 7–1) and a data type that is compatible with that of the subprogram parameter. (For information about compatible data types, see "Formal and Actual Subprogram Parameters" on page 8-6.)
  - No bind argument has a data type that SQL does not support (such as BOOLEAN in Example 7–2).

The USING clause cannot contain the literal NULL. To work around this restriction, use an uninitialized variable where you want to use NULL, as in Example 7–3.

For syntax details of the EXECUTE IMMEDIATE statement, see "EXECUTE IMMEDIATE Statement" on page 13-49.

***Example 7–1 Invoking a Subprogram from a Dynamic PL/SQL Block***

```
-- Subprogram that dynamic PL/SQL block invokes:
CREATE OR REPLACE PROCEDURE create_dept (
  deptid IN OUT NUMBER,
  dname  IN     VARCHAR2,
  mgrid  IN     NUMBER,
  locid  IN     NUMBER
) AS
BEGIN
  deptid := departments_seq.NEXTVAL;

  INSERT INTO departments (
    department_id,
    department_name,
    manager_id,
    location_id
  )
  VALUES (deptid, dname, mgrid, locid);
END;
/
DECLARE
  plsql_block VARCHAR2(500);
  new_deptid  NUMBER(4);
  new_dname   VARCHAR2(30) := 'Advertising';
  new_mgrid   NUMBER(6)    := 200;
  new_locid   NUMBER(4)    := 1700;
BEGIN
 -- Dynamic PL/SQL block invokes subprogram:
  plsql_block := 'BEGIN create_dept(:a, :b, :c, :d); END;';

 /* Specify bind arguments in USING clause.
    Specify mode for first parameter.
    Modes of other parameters are correct by default. */

  EXECUTE IMMEDIATE plsql_block
    USING IN OUT new_deptid, new_dname, new_mgrid, new_locid;
END;
```

```
/
```

***Example 7–2 Unsupported Data Type in Native Dynamic SQL***

```
DECLARE
  dyn_stmt VARCHAR2(200);
  b1       BOOLEAN;

  FUNCTION f (x INTEGER)
    RETURN BOOLEAN
  AS
  BEGIN
    NULL;
  END f;

BEGIN
  dyn_stmt := 'BEGIN :b := f(5); END;';
  EXECUTE IMMEDIATE dyn_stmt USING OUT b1;
END;
/
```

Result:

```
  EXECUTE IMMEDIATE dyn_stmt USING OUT b1;
                                       *
ERROR at line 15:
ORA-06550: line 15, column 40:
PLS-00457: expressions have to be of SQL types
ORA-06550: line 15, column 3:
PL/SQL: Statement ignored
```

***Example 7–3 Uninitialized Variable for NULL in USING Clause***

```
CREATE TABLE employees_temp AS
  SELECT * FROM EMPLOYEES
/
DECLARE
  a_null  CHAR(1);  -- Set to NULL automatically at run time
BEGIN
  EXECUTE IMMEDIATE 'UPDATE employees_temp SET commission_pct = :x'
    USING a_null;
END;
/
```

## OPEN FOR, FETCH, and CLOSE Statements

If the dynamic SQL statement represents a SELECT statement that returns multiple rows, you can process it with native dynamic SQL as follows:

1. Use an OPEN FOR statement to associate a cursor variable with the dynamic SQL statement. In the USING clause of the OPEN FOR statement, specify a bind argument for each placeholder in the dynamic SQL statement.

   The USING clause cannot contain the literal NULL. To work around this restriction, use an uninitialized variable where you want to use NULL, as in Example 7–3.

   For syntax details, see "OPEN FOR Statement" on page 13-98.

2. Use the FETCH statement to retrieve result set rows one at a time, several at a time, or all at once.

   For syntax details, see "FETCH Statement" on page 13-67.

3. Use the CLOSE statement to close the cursor variable.

   For syntax details, see "CLOSE Statement" on page 13-21.

Example 7–4 lists all employees who are managers, retrieving result set rows one at a time.

***Example 7–4 Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements***

```
DECLARE
  TYPE EmpCurTyp  IS REF CURSOR;
  v_emp_cursor    EmpCurTyp;
  emp_record      employees%ROWTYPE;
  v_stmt_str      VARCHAR2(200);
  v_e_job         employees.job%TYPE;
BEGIN
  -- Dynamic SQL statement with placeholder:
  v_stmt_str := 'SELECT * FROM employees WHERE job_id = :j';

  -- Open cursor & specify bind argument in USING clause:
  OPEN v_emp_cursor FOR v_stmt_str USING 'MANAGER';

  -- Fetch rows from result set one at a time:
  LOOP
    FETCH v_emp_cursor INTO emp_record;
    EXIT WHEN v_emp_cursor%NOTFOUND;
  END LOOP;

  -- Close cursor:
  CLOSE v_emp_cursor;
END;
/
```

## Repeated Placeholder Names in Dynamic SQL Statements

If you repeat placeholder names in dynamic SQL statements, be aware that the way placeholders are associated with bind arguments depends on the kind of dynamic SQL statement.

Topics:

- Dynamic SQL Statement is Not Anonymous Block or CALL Statement
- Dynamic SQL Statement is Anonymous Block or CALL Statement

### Dynamic SQL Statement is Not Anonymous Block or CALL Statement

If the dynamic SQL statement does not represent an anonymous PL/SQL block or a CALL statement, repetition of placeholder names is insignificant. Placeholders are associated with bind arguments in the USING clause by position, not by name.

For example, in this dynamic SQL statement, the repetition of the name :x is insignificant:

```
sql_stmt := 'INSERT INTO payroll VALUES (:x, :x, :y, :x)';
```

In the corresponding USING clause, you must supply four bind arguments. They can be different; for example:

```
EXECUTE IMMEDIATE sql_stmt USING a, b, c, d;
```

The preceding EXECUTE IMMEDIATE statement runs this SQL statement:

```
INSERT INTO payroll VALUES (a, b, c, d)
```

To associate the same bind argument with each occurrence of :x, you must repeat that bind argument; for example:

```
EXECUTE IMMEDIATE sql_stmt USING a, a, b, a;
```

The preceding EXECUTE IMMEDIATE statement runs this SQL statement:

```
INSERT INTO payroll VALUES (a, a, b, a)
```

### Dynamic SQL Statement is Anonymous Block or CALL Statement

If the dynamic SQL statement represents an anonymous PL/SQL block or a CALL statement, repetition of placeholder names is significant. Each unique placeholder name must have a corresponding bind argument in the USING clause. If you repeat a placeholder name, you need not repeat its corresponding bind argument. All references to that placeholder name correspond to one bind argument in the USING clause.

In Example 7–5, all references to the first unique placeholder name, :x, are associated with the first bind argument in the USING clause, a, and the second unique placeholder name, :y, is associated with the second bind argument in the USING clause, b.

***Example 7–5 Repeated Placeholder Names in Dynamic PL/SQL Block***

```
CREATE PROCEDURE calc_stats (
  w NUMBER,
  x NUMBER,
  y NUMBER,
  z NUMBER )
IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(w + x + y + z);
END;
/
DECLARE
  a NUMBER := 4;
  b NUMBER := 7;
  plsql_block VARCHAR2(100);
BEGIN
  plsql_block := 'BEGIN calc_stats(:x, :x, :y, :x); END;';
  EXECUTE IMMEDIATE plsql_block USING a, b;  -- calc_stats(a, a, b, a)
END;
/
```

## DBMS_SQL Package

The DBMS_SQL package defines an entity called a SQL cursor number. Because the SQL cursor number is a PL/SQL integer, you can pass it across call boundaries and store it.

You must use the DBMS_SQL package to run a dynamic SQL statement when you do not know either of these until run-time:

- SELECT list
- What placeholders in a SELECT or DML statement must be bound

In these situations, you must use native dynamic SQL instead of the DBMS_SQL package:

- The dynamic SQL statement retrieves rows into records.
- You want to use the SQL cursor attribute %FOUND, %ISOPEN, %NOTFOUND, or %ROWCOUNT after issuing a dynamic SQL statement that is an INSERT, UPDATE, DELETE, or single-row SELECT statement.

For information about native dynamic SQL, see "Native Dynamic SQL" on page 7-2.

When you need both the DBMS_SQL package and native dynamic SQL, you can switch between them, using:

- DBMS_SQL.TO_REFCURSOR Function
- DBMS_SQL.TO_CURSOR_NUMBER Function

---

**Note:** You can invoke DBMS_SQL subprograms remotely.

---

**See Also:** *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_SQL package, including instructions for running a dynamic SQL statement that has an unknown number of input or output variables ("Method 4")

## DBMS_SQL.TO_REFCURSOR Function

The DBMS_SQL.TO_REFCURSOR function converts a SQL cursor number to a weakly-typed variable of the PL/SQL data type REF CURSOR, which you can use in native dynamic SQL statements.

Before passing a SQL cursor number to the DBMS_SQL.TO_REFCURSOR function, you must OPEN, PARSE, and EXECUTE it (otherwise an error occurs).

After you convert a SQL cursor number to a REF CURSOR variable, DBMS_SQL operations can access it only as the REF CURSOR variable, not as the SQL cursor number. For example, using the DBMS_SQL.IS_OPEN function to see if a converted SQL cursor number is still open causes an error.

Example 7–6 uses the DBMS_SQL.TO_REFCURSOR function to switch from the DBMS_SQL package to native dynamic SQL.

***Example 7–6 Switching from DBMS_SQL Package to Native Dynamic SQL***

```
CREATE OR REPLACE TYPE vc_array IS TABLE OF VARCHAR2(200);
/
CREATE OR REPLACE TYPE numlist IS TABLE OF NUMBER;
/
CREATE OR REPLACE PROCEDURE do_query_1 (
  placeholder vc_array,
  bindvars vc_array,
  sql_stmt VARCHAR2
)
IS
  TYPE curtype IS REF CURSOR;
  src_cur     curtype;
  curid       NUMBER;
  bindnames   vc_array;
  empnos      numlist;
  depts       numlist;
```

```
  ret         NUMBER;
  isopen      BOOLEAN;
BEGIN
  -- Open SQL cursor number:
  curid := DBMS_SQL.OPEN_CURSOR;

  -- Parse SQL cursor number:
  DBMS_SQL.PARSE(curid, sql_stmt, DBMS_SQL.NATIVE);

  bindnames := placeholder;

  -- Bind arguments:
  FOR i IN 1 .. bindnames.COUNT LOOP
    DBMS_SQL.BIND_VARIABLE(curid, bindnames(i), bindvars(i));
  END LOOP;

  -- Run SQL cursor number:
  ret := DBMS_SQL.EXECUTE(curid);

  -- Switch from DBMS_SQL to native dynamic SQL:
  src_cur := DBMS_SQL.TO_REFCURSOR(curid);
  FETCH src_cur BULK COLLECT INTO empnos, depts;

  -- This would cause an error because curid was converted to a REF CURSOR:
  -- isopen := DBMS_SQL.IS_OPEN(curid);

  CLOSE src_cur;
END;
/
```

## DBMS_SQL.TO_CURSOR_NUMBER Function

The DBMS_SQL.TO_CURSOR function converts a REF CURSOR variable (either strongly or weakly typed) to a SQL cursor number, which you can pass to DBMS_SQL subprograms.

Before passing a REF CURSOR variable to the DBMS_SQL.TO_CURSOR function, you must OPEN it.

After you convert a REF CURSOR variable to a SQL cursor number, native dynamic SQL operations cannot access it.

After a FETCH operation begins, passing the DBMS_SQL cursor number to the DBMS_SQL.TO_REFCURSOR or DBMS_SQL.TO_CURSOR function causes an error.

Example 7–7 uses the DBMS_SQL.TO_CURSOR function to switch from native dynamic SQL to the DBMS_SQL package.

***Example 7–7 Switching from Native Dynamic SQL to DBMS_SQL Package***

```
CREATE OR REPLACE PROCEDURE do_query_2 (
  sql_stmt VARCHAR2
)
IS
  TYPE curtype IS REF CURSOR;
  src_cur   curtype;
  curid     NUMBER;
  desctab   DBMS_SQL.DESC_TAB;
  colcnt    NUMBER;
  namevar   VARCHAR2(50);
  numvar    NUMBER;
```

```
  datevar   DATE;
  empno     NUMBER := 100;
BEGIN
  -- sql_stmt := SELECT ... FROM employees WHERE employee_id = :b1';

  -- Open REF CURSOR variable:
  OPEN src_cur FOR sql_stmt USING empno;

  -- Switch from native dynamic SQL to DBMS_SQL package:
  curid := DBMS_SQL.TO_CURSOR_NUMBER(src_cur);
  DBMS_SQL.DESCRIBE_COLUMNS(curid, colcnt, desctab);

  -- Define columns:
  FOR i IN 1 .. colcnt LOOP
    IF desctab(i).col_type = 2 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, numvar);
    ELSIF desctab(i).col_type = 12 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, datevar);
      -- statements
    ELSE
      DBMS_SQL.DEFINE_COLUMN(curid, i, namevar, 50);
    END IF;
  END LOOP;

  -- Fetch rows with DBMS_SQL package:
  WHILE DBMS_SQL.FETCH_ROWS(curid) > 0 LOOP
    FOR i IN 1 .. colcnt LOOP
      IF (desctab(i).col_type = 1) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, namevar);
      ELSIF (desctab(i).col_type = 2) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, numvar);
      ELSIF (desctab(i).col_type = 12) THEN
        DBMS_SQL.COLUMN_VALUE(curid, i, datevar);
        -- statements
      END IF;
    END LOOP;
  END LOOP;

  DBMS_SQL.CLOSE_CURSOR(curid);
END;
/
```

# SQL Injection

SQL injection maliciously exploits applications that use client-supplied data in SQL statements, thereby gaining unauthorized access to a database to view or manipulate restricted data. This section describes SQL injection vulnerabilities in PL/SQL and explains how to guard against them.

To try the examples in this topic, connect to the HR schema and run the statements in Example 7–8.

***Example 7–8 Setup for SQL Injection Examples***

```
DROP TABLE secret_records;
CREATE TABLE secret_records (
  user_name    VARCHAR2(9),
  service_type VARCHAR2(12),
  value        VARCHAR2(30),
  date_created DATE
```

```
);

INSERT INTO secret_records (
  user_name, service_type, value, date_created
)
VALUES ('Andy', 'Waiter', 'Serve dinner at Cafe Pete', SYSDATE);

INSERT INTO secret_records (
  user_name, service_type, value, date_created
)
VALUES ('Chuck', 'Merger', 'Buy company XYZ', SYSDATE);
```

Topics:

- SQL Injection Techniques
- Guarding Against SQL Injection

## SQL Injection Techniques

All SQL injection techniques exploit a single vulnerability: String input is not correctly validated and is concatenated into a dynamic SQL statement.

Topics:

- Statement Modification
- Statement Injection
- Data Type Conversion

### Statement Modification

**Statement modification** means deliberately altering a dynamic SQL statement so that it runs in a way unintended by the application developer. Typically, the user retrieves unauthorized data by changing the `WHERE` clause of a `SELECT` statement or by inserting a `UNION ALL` clause. The classic example of this technique is bypassing password authentication by making a `WHERE` clause always `TRUE`.

Example 7–9 creates a procedure that is vulnerable to statement modification and then invokes that procedure with and without statement modification. With statement modification, the procedure returns a supposedly secret record.

***Example 7–9 Procedure Vulnerable to Statement Modification***

Create vulnerable procedure:

```
CREATE OR REPLACE PROCEDURE get_record (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2,
  rec          OUT VARCHAR2
)
IS
  query VARCHAR2(4000);
BEGIN
  -- Following SELECT statement is vulnerable to modification
  -- because it uses concatenation to build WHERE clause.
  query := 'SELECT value FROM secret_records WHERE user_name='''
           || user_name
           || ''' AND service_type='''
           || service_type
           || '''';
```

```
  DBMS_OUTPUT.PUT_LINE('Query: ' || query);
  EXECUTE IMMEDIATE query INTO rec ;
  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec );
END;
/
```

Demonstrate procedure without SQL injection:

```
SET SERVEROUTPUT ON;

DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_record('Andy', 'Waiter', record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records WHERE user_name='Andy' AND
service_type='Waiter'
Rec: Serve dinner at Cafe Pete
```

Example of statement modification:

```
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_record(
  'Anybody '' OR service_type=''Merger''--',
  'Anything',
  record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records WHERE user_name='Anybody ' OR
service_type='Merger'--' AND service_type='Anything'
Rec: Buy company XYZ

PL/SQL procedure successfully completed.
```

### Statement Injection

**Statement injection** means that a user appends one or more SQL statements to a dynamic SQL statement. Anonymous PL/SQL blocks are vulnerable to this technique.

Example 7–10 creates a procedure that is vulnerable to statement injection and then invokes that procedure with and without statement injection. With statement injection, the procedure deletes the supposedly secret record exposed in Example 7–9.

***Example 7–10 Procedure Vulnerable to Statement Injection***

Create vulnerable procedure:

```
CREATE OR REPLACE PROCEDURE p (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2
)
IS
  block1 VARCHAR2(4000);
```

```
BEGIN
  -- Following block is vulnerable to statement injection
  -- because it is built by concatenation.
  block1 :=
    'BEGIN
    DBMS_OUTPUT.PUT_LINE(''user_name: ' || user_name || ''');'
    || 'DBMS_OUTPUT.PUT_LINE(''service_type: ' || service_type || ''');
    END;';

  DBMS_OUTPUT.PUT_LINE('Block1: ' || block1);

  EXECUTE IMMEDIATE block1;
END;
/
```

Demonstrate procedure without SQL injection:

```
SET SERVEROUTPUT ON;

BEGIN
  p('Andy', 'Waiter');
END;
/
```

Result:

```
Block: BEGIN
         DBMS_OUTPUT.PUT_LINE('user_name: Andy');
         DBMS_OUTPUT.PUT_LINE('service_type: Waiter');
       END;
user_name: Andy
service_type: Waiter
```

SQL*Plus formatting command:

```
COLUMN date_created FORMAT A12;
```

Query:

```
SELECT * FROM secret_records;
```

Result:

```
USER_NAME SERVICE_TYPE VALUE                          DATE_CREATED
--------- ------------ ------------------------------ ------------
Andy      Waiter       Serve dinner at Cafe Pete      18-MAR-09
Chuck     Merger       Buy company XYZ                18-MAR-09
```

Example of statement modification:

```
BEGIN
  p('Anybody', 'Anything'');
  DELETE FROM secret_records WHERE service_type=INITCAP(''Merger');
END;
/
```

Result:

```
Block: BEGIN
       DBMS_OUTPUT.PUT_LINE('user_name: Anybody');
       DBMS_OUTPUT.PUT_LINE('service_type: Anything');
       DELETE FROM secret_records WHERE service_type=INITCAP('Merger');
     END;
```

```
user_name: Anybody
service_type: Anything

PL/SQL procedure successfully completed.
```

Query:

```
SELECT * FROM secret_records;
```

Result:

```
USER_NAME SERVICE_TYPE VALUE                          DATE_CREATED
--------- ------------ ------------------------------ ------------
Andy      Waiter       Serve dinner at Cafe Pete      18-MAR-09

1 row selected.
```

### Data Type Conversion

A less known SQL injection technique uses NLS session parameters to modify or inject SQL statements.

A datetime or numeric value that is concatenated into the text of a dynamic SQL statement must be converted to the VARCHAR2 data type. The conversion can be either implicit (when the value is an operand of the concatentation operator) or explicit (when the value is the argument of the TO_CHAR function). This data type conversion depends on the NLS settings of the database session that runs the dynamic SQL statement. The conversion of datetime values uses format models specified in the parameters NLS_DATE_FORMAT, NLS_TIMESTAMP_FORMAT, or NLS_TIMESTAMP_TZ_FORMAT, depending on the particular datetime data type. The conversion of numeric values applies decimal and group separators specified in the parameter NLS_NUMERIC_CHARACTERS.

One datetime format model is "*text*". The *text* is copied into the conversion result. For example, if the value of NLS_DATE_FORMAT is '"Month:" Month', then in June, TO_CHAR(SYSDATE) returns 'Month: June'. The datetime format model can be abused as shown in Example 7–11.

***Example 7–11 Procedure Vulnerable to SQL Injection Through Data Type Conversion***

```
SELECT * FROM secret_records;
```

Result:

```
USER_NAME SERVICE_TYPE VALUE                          DATE_CREATE
--------- ------------ ------------------------------ -----------
Andy      Waiter       Serve dinner at Cafe Pete      06-MAR-2009
Chuck     Merger       Buy company XYZ                06-MAR-2009
```

Create vulnerable procedure:

```
-- Return records not older than a month

CREATE OR REPLACE PROCEDURE get_recent_record (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2,
  rec          OUT VARCHAR2
)
IS
  query VARCHAR2(4000);
BEGIN
  /* Following SELECT statement is vulnerable to modification
```

```
     because it uses concatenation to build WHERE clause
     and because SYSDATE depends on the value of NLS_DATE_FORMAT. */

  query := 'SELECT value FROM secret_records WHERE user_name='''
           || user_name
           || ''' AND service_type='''
           || service_type
           || ''' AND date_created>'''
           || (SYSDATE - 30)
           || '''';

  DBMS_OUTPUT.PUT_LINE('Query: ' || query);
  EXECUTE IMMEDIATE query INTO rec;
  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);
END;
/
```

Demonstrate procedure without SQL injection:

```
SET SERVEROUTPUT ON;
ALTER SESSION SET NLS_DATE_FORMAT='DD-MON-YYYY';

DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_recent_record('Andy', 'Waiter', record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records WHERE user_name='Andy' AND
service_type='Waiter' AND date_created>'04-FEB-2009'
Rec: Serve dinner at Cafe Pete
```

Example of statement modification:

```
ALTER SESSION SET NLS_DATE_FORMAT='"'' OR service_type=''Merger"';

DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_recent_record('Anybody', 'Anything', record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records WHERE user_name='Anybody' AND
service_type='Anything' AND date_created>'' OR service_type='Merger'
Rec: Buy company XYZ

PL/SQL procedure successfully completed.
```

## Guarding Against SQL Injection

If you use dynamic SQL in your PL/SQL applications, you must check the input text to ensure that it is exactly what you expected. You can use the following techniques:

- Bind Arguments
- Validation Checks

- Explicit Format Models

### Bind Arguments

The most effective way to make your PL/SQL code invulnerable to SQL injection attacks is to use bind arguments. The database uses the values of bind arguments exclusively and does not interpret their contents in any way. (Bind arguments also improve performance.)

The procedure in Example 7–12 is invulnerable to SQL injection because it builds the dynamic SQL statement with bind arguments (not by concatenation as in the vulnerable procedure in Example 7–9). The same binding technique fixes the vulnerable procedure shown in Example 7–10.

***Example 7–12 Bind Arguments Guarding Against SQL Injection***

Create invulnerable procedure:

```
CREATE OR REPLACE PROCEDURE get_record_2 (
  user_name    IN  VARCHAR2,
  service_type IN  VARCHAR2,
  rec          OUT VARCHAR2
)
IS
  query VARCHAR2(4000);
BEGIN
  query := 'SELECT value FROM secret_records
            WHERE user_name=:a
            AND service_type=:b';

  DBMS_OUTPUT.PUT_LINE('Query: ' || query);

  EXECUTE IMMEDIATE query INTO rec USING user_name, service_type;

  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);
END;
/
```

Demonstrate procedure without SQL injection:

```
SET SERVEROUTPUT ON;
DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_record_2('Andy', 'Waiter', record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records
            WHERE user_name=:a
            AND service_type=:b
Rec: Serve dinner at Cafe Pete

PL/SQL procedure successfully completed.
```

Attempt statement modification:

```
DECLARE
  record_value VARCHAR2(4000);
BEGIN
```

```
  get_record_2('Anybody '' OR service_type=''Merger''--',
               'Anything',
               record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records
            WHERE user_name=:a
            AND service_type=:b
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "HR.GET_RECORD_2", line 14
ORA-06512: at line 4
```

### Validation Checks

Always have your program validate user input to ensure that it is what is intended. For example, if the user is passing a department number for a DELETE statement, check the validity of this department number by selecting from the departments table. Similarly, if a user enters the name of a table to be deleted, check that this table exists by selecting from the static data dictionary view ALL_TABLES.

---

**Caution:** When checking the validity of a user name and its password, always return the same error regardless of which item is invalid. Otherwise, a malicious user who receives the error message "invalid password" but not "invalid user name" (or the reverse) can realize that he or she has guessed one of these correctly.

---

In validation-checking code, the subprograms in the DBMS_ASSERT package are often useful. For example, you can use the DBMS_ASSERT.ENQUOTE_LITERAL function to enclose a string literal in quotation marks, as Example 7–13 does. This prevents a malicious user from injecting text between an opening quotation mark and its corresponding closing quotation mark.

---

**Caution:** Although the DBMS_ASSERT subprograms are useful in validation code, they do not replace it. For example, an input string can be a qualified SQL name (verified by DBMS_ASSERT.QUALIFIED_SQL_NAME) and still be a fraudulent password.

---

**See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about DBMS_ASSERT subprograms

In Example 7–13, the procedure raise_emp_salary checks the validity of the column name that was passed to it before it updates the employees table, and then the anonymous block invokes the procedure from both a dynamic PL/SQL block and a dynamic SQL statement.

***Example 7–13 Validation Checks Guarding Against SQL Injection***

```
CREATE OR REPLACE PROCEDURE raise_emp_salary (
  column_value  NUMBER,
```

```
  emp_column    VARCHAR2,
  amount NUMBER )
IS
  v_column  VARCHAR2(30);
  sql_stmt  VARCHAR2(200);
BEGIN
  -- Check validity of column name that was given as input:
  SELECT column_name INTO v_column
  FROM USER_TAB_COLS
  WHERE TABLE_NAME = 'EMPLOYEES'
  AND COLUMN_NAME = emp_column;

  sql_stmt := 'UPDATE employees SET salary = salary + :1 WHERE '
    || DBMS_ASSERT.ENQUOTE_NAME(v_column,FALSE) || ' = :2';

  EXECUTE IMMEDIATE sql_stmt USING amount, column_value;

  -- If column name is valid:
  IF SQL%ROWCOUNT > 0 THEN
    DBMS_OUTPUT.PUT_LINE('Salaries were updated for: '
      || emp_column || ' = ' || column_value);
  END IF;

  -- If column name is not valid:
  EXCEPTION
    WHEN NO_DATA_FOUND THEN
      DBMS_OUTPUT.PUT_LINE ('Invalid Column: ' || emp_column);
END raise_emp_salary;
/

DECLARE
  plsql_block  VARCHAR2(500);
BEGIN
  -- Invoke raise_emp_salary from a dynamic PL/SQL block:
  plsql_block :=
    'BEGIN raise_emp_salary(:cvalue, :cname, :amt); END;';

  EXECUTE IMMEDIATE plsql_block
    USING 110, 'DEPARTMENT_ID', 10;

  -- Invoke raise_emp_salary from a dynamic SQL statement:
  EXECUTE IMMEDIATE 'BEGIN raise_emp_salary(:cvalue, :cname, :amt); END;'
    USING 112, 'EMPLOYEE_ID', 10;
END;
/
```

Result:

```
Salaries were updated for: DEPARTMENT_ID = 110
Salaries were updated for: EMPLOYEE_ID = 112
```

### Explicit Format Models

If you use datetime and numeric values that are concatenated into the text of a SQL or PL/SQL statement, and you cannot pass them as bind variables, convert them to text using explicit format models that are independent from the values of the NLS parameters of the running session. Ensure that the converted values have the format of SQL datetime or numeric literals. Using explicit locale-independent format models to construct SQL is recommended not only from a security perspective, but also to ensure that the dynamic SQL statement runs correctly in any globalization environment.

The procedure in Example 7–14 is invulnerable to SQL injection because it converts the datetime parameter value, `SYSDATE - 30`, to a `VARCHAR2` value explicitly, using the `TO_CHAR` function and a locale-independent format model (not implicitly, as in the vulnerable procedure in Example 7–11).

***Example 7–14 Explicit Format Models Guarding Against SQL Injection***

Create invulnerable procedure:

```
-- Return records not older than a month

CREATE OR REPLACE PROCEDURE get_recent_record (
  user_name     IN  VARCHAR2,
  service_type  IN  VARCHAR2,
  rec           OUT VARCHAR2
)
IS
  query VARCHAR2(4000);
BEGIN
  /* Following SELECT statement is vulnerable to modification
     because it uses concatenation to build WHERE clause. */

  query := 'SELECT value FROM secret_records WHERE user_name='''
           || user_name
           || ''' AND service_type='''
           || service_type
           || ''' AND date_created> DATE '''
           || TO_CHAR(SYSDATE - 30,'YYYY-MM-DD')
           || '''';

  DBMS_OUTPUT.PUT_LINE('Query: ' || query);
  EXECUTE IMMEDIATE query INTO rec;
  DBMS_OUTPUT.PUT_LINE('Rec: ' || rec);
END;
/
```

Attempt statement modification:

```
ALTER SESSION SET NLS_DATE_FORMAT='"'' OR service_type=''Merger"';

DECLARE
  record_value VARCHAR2(4000);
BEGIN
  get_recent_record('Anybody', 'Anything', record_value);
END;
/
```

Result:

```
Query: SELECT value FROM secret_records WHERE user_name='Anybody' AND
service_type='Anything' AND date_created> DATE '2009-02-04'
DECLARE
*
ERROR at line 1:
ORA-01403: no data found
ORA-06512: at "SYS.GET_RECENT_RECORD", line 21
ORA-06512: at line 4
```

# 8 PL/SQL Subprograms

This chapter explains how to turn sets of statements into reusable subprograms. Subprograms are the building blocks of modular, maintainable applications.

Topics:


- Overview of PL/SQL Subprograms
- Subprogram Parts
- Nested Subprograms that Invoke Each Other
- Subprogram Parameters
- Overloaded Subprograms
- Subprogram Call Resolution
- Invoker's Rights and Definer's Rights (AUTHID Property)
- Recursive Subprograms
- External Subprograms
- Subprogram Side Effects
- Subprogram Parameter Aliasing
- PL/SQL Function Result Cache


## Overview of PL/SQL Subprograms

A PL/SQL subprogram is a named PL/SQL block that can be invoked with a set of parameters (if it has them). A subprogram can be either a procedure or a function. Typically, you use a procedure to perform an action and a function to compute and return a value.

You can create a subprogram either at schema level, inside a package, or inside a PL/SQL block (which can be another subprogram).

A subprogram created at schema level is a **standalone stored subprogram**. You create it with the `CREATE PROCEDURE` or `CREATE FUNCTION` statement. It is stored in the database until you drop it with the `DROP PROCEDURE` or `DROP FUNCTION` statement.

A subprogram created inside a package is a **packaged subprogram**. It is stored in the database until you drop the package with the `DROP PACKAGE` statement.

A subprogram created inside a PL/SQL block is a **nested subprogram**. You can either declare and define it at the same time, or you can declare it first (**forward declaration**) and then define it later in the same block. A nested subprogram is stored in the database only if it is nested in a standalone or packaged subprogram.

**See Also:**

- "CREATE PROCEDURE Statement" on page 14-49 for more information about creating standalone stored procedures
- "CREATE FUNCTION Statement" on page 14-32 for more information about creating standalone stored functions
- "CREATE PACKAGE Statement" on page 14-42 for more information about creating standalone stored functions
- "Procedure" on page 13-103 for more information about creating procedures inside PL/SQL blocks
- "Function" on page 13-76 for more information about creating functions inside PL/SQL blocks

### Subprogram Calls

A subprogram call has this form:

```
subprogram_name [ ( [ parameter [, parameter]... ] ) ]
```

If the subprogram has no parameters, or specifies an initial value for every parameter, you can either omit the parameter list or specify an empty parameter list.

A procedure call is a PL/SQL statement. For example:

```
raise_salary(employee_id, amount);
```

A function call is part of an expression. For example:

```
IF salary_ok(new_salary, new_title) THEN ...
```

**See Also:** "Subprogram Parameters" on page 8-6 for more information about subprogram calls

### Reasons to Use Subprograms

- Subprograms let you extend the PL/SQL language.

  Procedure calls are like new statements. Function calls are like new expressions and operators.

- Subprograms let you break a program into manageable, well-defined modules.

  You can use top-down design and the stepwise refinement approach to problem solving.

- Subprograms promote re-usability.

  Once tested, a subprogram can be reused in any number of applications. You can invoke PL/SQL subprograms from many different environments, so that you need not rewrite them each time you use another language or API to access the database.

- Subprograms promote maintainability.

  You can change the internal details of a subprogram without changing the other subprograms that invoke it. Subprograms are an important component of other maintainability features, such as packages and ADTs.

- Dummy subprograms ("stubs") let you defer the definition of procedures and functions until after you have tested the main program.

You can design applications from the top down, thinking abstractly, without worrying about implementation details.

- Subprograms can be grouped into PL/SQL packages.

  Packages make code even more reusable and maintainable, and can be used to define an API.

- You can hide the implementation details of subprograms by placing them in PL/SQL packages.

  You can define subprograms in a package body without declaring their specifications in the package specification. However, such subprograms can be invoked only from inside the package. At least one statement must appear in the executable part of a subprogram. The NULL statement meets this requirement.

## Subprogram Parts

A subprogram always has a name, and can have a parameter list.

Like every PL/SQL block, a subprogram has an optional declarative part, a required executable part, and an optional exception-handling part, and can specify PRAGMA AUTONOMOUS_TRANSACTION, which makes it autonomous (independent).

The **declarative part** of a subprogram does not begin with the keyword DECLARE, as the declarative part of an anonymous block does. The declarative part contains declarations of types, cursors, constants, variables, exceptions, and nested subprograms. These items are local to the subprogram and cease to exist when the subprogram completes execution.

The **executable part** of a subprogram contains statements that assign values, control execution, and manipulate data.

The **exception-handling part** of a subprogram contains code that handles run-time errors.

Example 8–1 declares and defines a procedure (at the same time) inside an anonymous block. The procedure has the required executable part and the optional exception-handling part, but not the optional declarative part. The executable part of the block invokes the procedure.

***Example 8–1 Declaring, Defining, and Invoking a Simple PL/SQL Procedure***

```
-- Declarative part of block begins
DECLARE
  in_string  VARCHAR2(100) := 'This is my test string.';
  out_string VARCHAR2(200);

  -- Procedure declaration and definition begins
  PROCEDURE double (original IN VARCHAR2,
                    new_string OUT VARCHAR2)
  IS
    -- Declarative part of procedure (optional) goes here
    -- Executable part of procedure begins
    BEGIN
      new_string := original || ' + ' || original;
    -- Executable part of procedure ends
    -- Exception-handling part of procedure begins
    EXCEPTION
      WHEN VALUE_ERROR THEN
        DBMS_OUTPUT.PUT_LINE('Output buffer not long enough.');
```

```
    END;
    -- Exception-handling part of procedure ends
    -- Procedure declaration and definition ends
-- Declarative part of block ends
-- Executable part of block begins
BEGIN
  double(in_string, out_string);  -- Procedure invocation
  DBMS_OUTPUT.PUT_LINE(in_string || ' - ' || out_string);
END;
-- Executable part of block ends
/
```

A procedure and a function have the same structure, except that:

- A function heading must include a RETURN clause that specifies the data type of the return value. A procedure heading cannot have a RETURN clause.
- A function must have at least one RETURN statement in its executable part. In a procedure, the RETURN statement is optional. For details, see "RETURN Statement" on page 8-4.
- Only a function declaration or definition can include these options:

| Option | Description |
|---|---|
| DETERMINISTIC option | Helps the optimizer avoid redundant function calls. |
| PARALLEL_ENABLE option | Enables the function to be used safely in slave sessions of parallel DML evaluations. |
| PIPELINED option | Returns the results of a table function iteratively. |
| RESULT_CACHE option | Stores function results in the PL/SQL function result cache (appears only in declaration). |
| RESULT_CACHE clause | Stores function results in the PL/SQL function result cache (appears only in definition). |

**See Also:**


- "Procedure" on page 13-103 for the syntax of procedure declarations and definitions
- "Function" on page 13-76 for the syntax of function declarations and definitions, including descriptions of the items in the preceding table
- "Subprogram Parameters" on page 8-6 for more information about subprogram parameters
- "PL/SQL Function Result Cache" on page 8-29 for more information about the RESULT_CACHE option and the RESULT_CACHE clause


### RETURN Statement

The RETURN statement (not to be confused with the RETURN clause, which specifies the data type of the return value of a function) immediately ends the execution of the subprogram that contains it and returns control to the caller. Execution continues with the statement following the subprogram call.

A subprogram can contain several RETURN statements. The subprogram need not end with a RETURN statement. Running any RETURN statement completes the subprogram immediately.

In a procedure or anonymous block, a RETURN statement cannot contain an expression and does not return a value.

In a function, a RETURN statement must contain an expression. When the RETURN statement runs, the expression is evaluated, and its value is assigned to the function identifier. The function identifier acts like a variable of the type specified in the RETURN clause.

The expression in a function RETURN statement can be arbitrarily complex. For example:

```
CREATE OR REPLACE FUNCTION half_of_square(original NUMBER)
  RETURN NUMBER IS
BEGIN
  RETURN (original * original)/2 + (original * 4);
END half_of_square;
/
```

A function must have at least one execution path that leads to a RETURN statement.

> **See Also:** "RETURN Statement" on page 13-111 for the syntax of the RETURN statement

Example 8–2 declares and defines a function (at the same time) inside an anonymous block. The function has the optional declarative part and the required executable part, but not the optional exception-handling part. The executable part of the block invokes the function.

***Example 8–2 Declaring, Defining, and Invoking a Simple PL/SQL Function***

```
-- Declarative part of block begins
DECLARE
  -- Function declaration and definition begins
  FUNCTION square (original NUMBER)
    RETURN NUMBER  -- RETURN clause
  AS
  -- Declarative part of function begins
    original_squared NUMBER;
  -- Declarative part of function ends
  -- Executable part of function begins
  BEGIN
    original_squared := original * original;
    RETURN original_squared;  -- RETURN statement
  -- Exception-handling part of function (optional) goes here
  END;
  -- Executable part of function ends
  -- Function declaration and definition ends
-- Declarative part of block ends
-- Executable part of block begins
BEGIN
  DBMS_OUTPUT.PUT_LINE(square(100));  -- Function invocation
END;
-- Executable part of block ends
/
```

# Nested Subprograms that Invoke Each Other

In a block, you can create multiple nested subprograms. If they invoke each other, you need forward declaration, because a subprogram must be declared before it can be invoked. With forward declaration, you declare a subprogram, but do not define it until after you have defined the other subprograms that invoke it. A forward declaration and its corresponding definition must appear in the same block.

The block in Example 8–3 creates two procedures that invoke each other.

***Example 8–3 Creating Nested Subprograms that Invoke Each Other***

```
DECLARE
  -- Declare proc1 (forward declaration):
  PROCEDURE proc1(number1 NUMBER);

  -- Declare and define proc 2:
  PROCEDURE proc2(number2 NUMBER) IS
  BEGIN
    proc1(number2);
  END;

  -- Define proc 1:
  PROCEDURE proc1(number1 NUMBER) IS
  BEGIN
    proc2 (number1);
  END;

BEGIN
  NULL;
END;
/
```

# Subprogram Parameters

A subprogram heading can declare formal parameters. Each formal parameter declaration can specify a mode and a initial value. When you invoke the subprogram, you can pass actual parameters to it.

Topics:

- Formal and Actual Subprogram Parameters
- Subprogram Parameter Passing Methods
- Subprogram Parameter Modes
- Initial Values for Subprogram Parameters
- Positional, Named, and Mixed Notation for Actual Parameters

## Formal and Actual Subprogram Parameters

**Formal parameters** are the variables declared in the subprogram header and referenced in its execution part. **Actual parameters** are the variables or expressions that you pass to the subprogram when you invoke it. Corresponding formal and actual parameters must have compatible data types.

A good programming practice is to use different names for formal and actual parameters, as in Example 8–4.

***Example 8–4 Formal Parameters and Actual Parameters***

```
DECLARE
  emp_num NUMBER(6) := 120;
  bonus   NUMBER(6) := 100;
  merit   NUMBER(4) := 50;

  PROCEDURE raise_salary (
    emp_id NUMBER,  -- formal parameter
    amount NUMBER   -- formal parameter
  ) IS
  BEGIN
    UPDATE employees
    SET salary = salary + amount
    WHERE employee_id = emp_id;
  END raise_salary;

BEGIN
  raise_salary(emp_num, bonus);  -- actual parameters
  raise_salary(emp_num, merit + bonus);  -- actual parameters
END;
/
```

## Subprogram Parameter Passing Methods

The PL/SQL compiler has two methods of passing an actual parameter to a subprogram:

- **By value**, where the compiler passes the value of the actual parameter
- **By reference**, where the compiler passes a pointer to the value of the actual parameter (the actual and formal parameters reference the same item)

The method by which the compiler passes an actual parameter depends on its mode and on the `NOCOPY` hint. If the mode is `IN`, the compiler always passes the parameter by reference. With `IN` mode, you cannot specify `NOCOPY`. If the mode is `OUT` or `IN OUT`, the compiler passes the parameter by value by default; however, if you specify `NOCOPY`, the compiler might pass the parameter by reference. The optimizer decides, silently, whether to obey or disregard the `NOCOPY` hint.

To pass an actual parameter by value, the compiler evaluates the actual parameter and assigns its value to the corresponding formal parameter. If necessary, the compiler converts the data type of the actual parameter to the data type of the corresponding formal parameter (this is why corresponding formal and actual parameters must have compatible data types). This form of conversion is explained in "Implicit Conversion" on page 3-27.

A good programming practice is to avoid implicit conversion, either by using explicit conversion (explained in "Explicit Conversion" on page 3-27) or by declaring the variables that you intend to use as actual parameters with the same data types as their corresponding formal parameters (see Example 8–5).

***Example 8–5 Avoiding and Causing Implicit Conversion of Actual Parameters***

```
CREATE OR REPLACE PROCEDURE s
  (n IN PLS_INTEGER)
IS
BEGIN
  NULL;
END;
/
```

```
DECLARE
  y PLS_INTEGER :=1;   -- Avoids implicit conversion
  z INTEGER :=1;       -- Causes implicit conversion
BEGIN
  s(y);
  s(z);
END;
/
```

---

**Note:** The specifications of many packages and types that Oracle Database supplies declare formal parameters with this notation:

```
i1 IN VARCHAR2 CHARACTER SET ANY_CS
i2 IN VARCHAR2 CHARACTER SET i1%CHARSET
```

Do not use this notation when declaring your own formal or actual parameters. It is reserved for Oracle implementation of the supplied packages types.

---

## Subprogram Parameter Modes

Parameter modes define the action of formal parameters. The three parameter modes are `IN` (the default), `OUT`, and `IN OUT`.

Any parameter mode can be used with any subprogram. However, it is poor programming practice to use the `OUT` or `IN OUT` mode for function parameters, because a function with `OUT` or `IN OUT` parameters returns multiple values and has side effects.

Topics:

- IN Mode
- OUT Mode
- IN OUT Mode
- Summary of Subprogram Parameter Modes

### IN Mode

The compiler always passes an `IN` parameter by reference. Inside the subprogram, an `IN` parameter acts like a constant: The subprogram can assign it an initial value, which is used if that parameter is omitted from the subprogram call, but cannot assign it any other value.

You can pass a constant, literal, initialized variable, or expression as an `IN` parameter.

### OUT Mode

By default, the compiler passes an `OUT` parameter by value (if you specify the `NOCOPY` hint, the compiler might pass it by reference). Inside the subprogram, an `OUT` parameter acts like a variable. You can change its value and reference the value after assigning it, as in Example 8–6.

***Example 8–6 Subprogram Parameter Mode OUT***

```
DECLARE
  emp_num        NUMBER(6) := 120;
  bonus          NUMBER(6) := 50;
  emp_last_name  VARCHAR2(25);
```

```
  PROCEDURE raise_salary ( emp_id IN NUMBER
                         , amount IN NUMBER
                         , emp_name OUT VARCHAR2 )
  IS
  BEGIN
    UPDATE employees
    SET salary = salary + amount
    WHERE employee_id = emp_id;

    SELECT last_name INTO emp_name
    FROM employees
    WHERE employee_id = emp_id;
  END raise_salary;

BEGIN
  raise_salary(emp_num, bonus, emp_last_name);
  DBMS_OUTPUT.PUT_LINE
    ('Salary was updated for: ' || emp_last_name);
END;
/
```

If you specify the NOCOPY hint for an OUT parameter, the compiler might pass the parameter by reference instead of by value.

You must pass a variable, not a constant or an expression, to an OUT parameter. Its previous value is lost unless you specify the NOCOPY hint or the subprogram exits with an unhandled exception. OUT parameters declared with numeric subtypes are constrained by those subtypes, but OUT parameters declared with VARCHAR2 subtypes are contrained by the sizes of their actual parameters. See "Initial Values for Subprogram Parameters" on page 8-10.

The initial value of an OUT parameter is NULL; therefore, the data type of an OUT parameter cannot be a subtype defined as NOT NULL, such as NATURALN or POSITIVEN. Otherwise, when you invoke the subprogram, PL/SQL raises VALUE_ERROR.

Before exiting a subprogram, assign values to all OUT formal parameters. Otherwise, the corresponding actual parameters will be NULL. If the subprogram is exited successfully, PL/SQL assigns values to the actual parameters. If the subprogram is exited with an unhandled exception, PL/SQL does not assign values to the actual parameters.

### IN OUT Mode

By default, the compiler passes an IN OUT parameter by value (if you specify the NOCOPY hint, the compiler might pass them by reference). An IN OUT parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and its value can be read.

The actual parameter that corresponds to an IN OUT formal parameter must be a variable, not a constant or an expression. Typically, an IN OUT parameter is a string buffer or numeric accumulator, which the subprogram reads and then updates. IN OUT parameters declared with numeric subtypes are constrained by those subtypes, but IN OUT parameters declared with VARCHAR2 subtypes are contrained by the sizes of their actual parameters.

If you exit a subprogram successfully, PL/SQL assigns values to the actual parameters. If you exit with an unhandled exception, PL/SQL does not assign values to the actual parameters.

### Summary of Subprogram Parameter Modes

Table 8–1 summarizes the characteristics of parameter modes.

***Table 8–1 Parameter Modes***

| IN | OUT | IN OUT |
|---|---|---|
| The default | Must be specified | Must be specified |
| Passes a value to the subprogram. | Returns a value to the caller. | Passes an initial value to the subprogram and returns an updated value to the caller. |
| Formal parameter acts like a constant. | Formal parameter acts like an uninitialized variable. | Formal parameter acts like an initialized variable. |
| Formal parameter cannot be assigned a value. | Formal parameter must be assigned a value. | Formal parameter should be assigned a value. |
| Actual parameter can be a constant, initialized variable, literal, or expression. | Actual parameter must be a variable. | Actual parameter must be a variable. |
| Actual parameter is passed by reference. | By default, actual parameter is passed by value; if `NOCOPY` is specified, it might be passed by reference. | By default, actual parameter is passed by value (in both directions) if `NOCOPY` is specified, it might be passed by reference. |
| | Parameters declared with numeric subtypes are constrained by those subtypes, but parameters declared with `VARCHAR2` subtypes are contrained by the sizes of their actual parameters. | Parameters declared with numeric subtypes are constrained by those subtypes, but parameters declared with `VARCHAR2` subtypes are contrained by the sizes of their actual parameters. |

## Initial Values for Subprogram Parameters

By initializing formal `IN` parameters to initial values, you can pass different numbers of actual parameters to a subprogram, accepting the initial values for omitted actual parameters. You can also add formal parameters without having to change every call to the subprogram.

If an actual parameter is omitted, the initial value of its corresponding formal parameter is used.

You cannot skip a formal parameter by omitting its actual parameter. To omit the first parameter and specify the second, use named notation (see "Positional, Named, and Mixed Notation for Actual Parameters" on page 8-12).

You cannot assign `NULL` to an uninitialized formal parameter by omitting its actual parameter. You must either assign `NULL` as an initial value or pass `NULL` explicitly.

Example 8–7 shows the effect of initial values for subprogram parameters.

***Example 8–7 Procedure with Initial Parameter Values***

```
DECLARE
  emp_num NUMBER(6) := 120;
  bonus   NUMBER(6);
  merit   NUMBER(4);

  PROCEDURE raise_salary (emp_id IN NUMBER,
    amount IN NUMBER := 100,
```

```
    extra  IN NUMBER := 50
  ) IS
  BEGIN
    UPDATE employees
    SET salary = salary + amount + extra
    WHERE employee_id = emp_id;
  END raise_salary;

BEGIN
  -- Same as raise_salary(120, 100, 50):
  raise_salary(120);

  -- Same as raise_salary(120, 100, 25):
  raise_salary(emp_num, extra => 25);
END;
/
```

If the initial value of a formal parameter is an expression, and you provide a corresponding actual parameter when you invoke the subprogram, the expression is not evaluated, as Example 8–8 shows.

***Example 8–8 Formal Parameter with Expression as Initial Value***

```
DECLARE
  cnt  pls_integer := 0;

  FUNCTION dflt RETURN pls_integer IS
  BEGIN
    cnt := cnt + 1;
    RETURN 42;
  END dflt;

  PROCEDURE p (
    i IN pls_integer := dflt() -- initial value is expression
  ) IS
  BEGIN
    DBMS_Output.Put_Line(i);
  END p;

BEGIN
  FOR j IN 1..5 LOOP
    p(j);  -- Actual parameter is provided
  END loop;
  DBMS_Output.Put_Line('cnt: '||cnt);
  p();  -- Actual parameter is not provided
  DBMS_Output.Put_Line('cnt: '||cnt);
END;
/
```

The output of Example 8–8 is:

```
1
2
3
4
5
Cnt: 0
42
Cnt: 1
```

## Positional, Named, and Mixed Notation for Actual Parameters

When invoking a subprogram, you can specify the actual parameters using either positional, named, or mixed notation. Table 8–2 compares these notations.

*Table 8–2 PL/SQL Subprogram Parameter Notations*

| Notation | Description | Usage Notes |
|---|---|---|
| Positional | Specify the same parameters in the same order as the procedure declares them. | Compact and readable, but has these disadvantages:<br>▪ If you specify the parameters (especially literals) in the wrong order, the bug can be hard to detect.<br>▪ If the procedure's parameter list changes, you must change your code. |
| Named | Specify the name and value of each parameter, using the association operator, =>. Order of parameters is insignificant. | More verbose than positional notation, but easier to read and maintain. You can sometimes avoid changing your code if the procedure's parameter list changes (for example, if parameters are reordered or an optional parameter is added). Safer than positional notation when you invoke an API that you did not define, or define an API for others to use. |
| Mixed | Start with positional notation, then use named notation for the remaining parameters. | Recommended when you invoke procedures that have required parameters followed by optional parameters, and you must specify only a few of the optional parameters. |

Example 8–9 shows equivalent subprogram calls using positional, named, and mixed notation.

*Example 8–9 Subprogram Calls Using Positional, Named, and Mixed Notation*

```
DECLARE
  emp_num NUMBER(6) := 120;
  bonus   NUMBER(6) := 50;

  PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS
  BEGIN
    UPDATE employees SET salary =
      salary + amount WHERE employee_id = emp_id;
  END raise_salary;
BEGIN
  -- Positional notation:
  raise_salary(emp_num, bonus);
  -- Named notation (parameter order is insignificant):
  raise_salary(amount => bonus, emp_id => emp_num);
  raise_salary(emp_id => emp_num, amount => bonus);
  -- Mixed notation:
  raise_salary(emp_num, amount => bonus);
END;
/
CREATE OR REPLACE FUNCTION compute_bonus (
  emp_id NUMBER,
  bonus NUMBER
)  RETURN NUMBER
IS
  emp_sal NUMBER;
BEGIN
  SELECT salary INTO emp_sal
```

```
  FROM employees
  WHERE employee_id = emp_id;

  RETURN emp_sal + bonus;
END compute_bonus;
/
SELECT compute_bonus(120, 50) FROM DUAL;                   -- positional
SELECT compute_bonus(bonus => 50, emp_id => 120) FROM DUAL; -- named
SELECT compute_bonus(120, bonus => 50) FROM DUAL;           -- mixed
```

## Overloaded Subprograms

PL/SQL lets you overload local subprograms, packaged subprograms, and type methods. You can use the same name for several different subprograms if their formal parameters differ in number, order, or data type family.

Example 8–10 defines two subprograms with the same name, `initialize`. The procedures initialize different types of collections. Because the processing in these two procedures is the same, it is logical to give them the same name.

You can put the two `initialize` procedures in the same block, subprogram, package, or ADT. PL/SQL determines which procedure to invoke by checking their formal parameters. The version of `initialize` that PL/SQL uses depends on whether you invoke the procedure with a `date_tab_typ` or `num_tab_typ` parameter.

***Example 8–10 Overloaded Subprogram***

```
DECLARE
  TYPE date_tab_typ IS TABLE OF DATE   INDEX BY PLS_INTEGER;
  TYPE num_tab_typ  IS TABLE OF NUMBER INDEX BY PLS_INTEGER;

  hiredate_tab  date_tab_typ;
  sal_tab       num_tab_typ;

  PROCEDURE initialize (tab OUT date_tab_typ, n INTEGER) IS
  BEGIN
    FOR i IN 1..n LOOP
      tab(i) := SYSDATE;
    END LOOP;
  END initialize;

  PROCEDURE initialize (tab OUT num_tab_typ, n INTEGER) IS
  BEGIN
    FOR i IN 1..n LOOP
      tab(i) := 0.0;
    END LOOP;
  END initialize;

BEGIN
  initialize(hiredate_tab, 50);  -- Invokes first (date_tab_typ) version
  initialize(sal_tab, 100);      -- Invokes second (num_tab_typ) version
END;
/
```

For an example of an overloaded procedure in a package, see Example 10–6 on page 10-11.

Topics:

- Subprograms that You Can Overload
- Subprograms that You Cannot Overload
- Subprogram Overload Errors

## Subprograms that You Can Overload

You can overload subprograms if their formal parameters differ only in numeric data type. This technique is useful in writing mathematical application programming interfaces (APIs), because several versions of a function can use the same name, and each can accept a different numeric type. For example, a function that accepts BINARY_FLOAT might be faster, while a function that accepts BINARY_DOUBLE might provide more precision.

To avoid problems or unexpected results passing parameters to such overloaded subprograms:

- Ensure that the expected version of a subprogram is invoked for each set of expected parameters. For example, if you have overloaded functions that accept BINARY_FLOAT and BINARY_DOUBLE, which is invoked if you pass a VARCHAR2 literal such as '5.0'?
- Qualify numeric literals and use conversion functions to make clear what the intended parameter types are. For example, use literals such as 5.0f (for BINARY_FLOAT), 5.0d (for BINARY_DOUBLE), or conversion functions such as TO_BINARY_FLOAT, TO_BINARY_DOUBLE, and TO_NUMBER.

PL/SQL looks for matching numeric parameters in this order:

1. PLS_INTEGER (or BINARY_INTEGER, an identical data type)
2. NUMBER
3. BINARY_FLOAT
4. BINARY_DOUBLE

A VARCHAR2 value can match a NUMBER, BINARY_FLOAT, or BINARY_DOUBLE parameter.

PL/SQL uses the first overloaded subprogram that matches the supplied parameters. For example, the SQRT function takes a single parameter. There are overloaded versions that accept a NUMBER, a BINARY_FLOAT, or a BINARY_DOUBLE parameter. If you pass a PLS_INTEGER parameter, the first matching overload is the one with a NUMBER parameter.

The SQRT function that takes a NUMBER parameter is likely to be slowest. To use a faster version, use the TO_BINARY_FLOAT or TO_BINARY_DOUBLE function to convert the parameter to another data type before passing it to the SQRT function.

If PL/SQL must convert a parameter to another data type, it first tries to convert it to a higher data type. For example:

- The ATAN2 function takes two parameters of the same type. If you pass parameters of different types—for example, one PLS_INTEGER and one BINARY_FLOAT—PL/SQL tries to find a match where both parameters use the higher type. In this case, that is the version of ATAN2 that takes two BINARY_FLOAT parameters; the PLS_INTEGER parameter is converted upwards.
- A function takes two parameters of different types. One overloaded version takes a PLS_INTEGER and a BINARY_FLOAT parameter. Another overloaded version takes a NUMBER and a BINARY_DOUBLE parameter. If you invoke this function and

pass two NUMBER parameters, PL/SQL first finds the overloaded version where the second parameter is BINARY_FLOAT. Because this parameter is a closer match than the BINARY_DOUBLE parameter in the other overload, PL/SQL then looks downward and converts the first NUMBER parameter to PLS_INTEGER.

## Subprograms that You Cannot Overload

You cannot overload these subprograms:

- Standalone subprograms
- Subprograms whose formal parameters differ only in mode; for example:

```
PACKAGE pkg IS
  PROCEDURE s (p IN  VARCHAR2);
  PROCEDURE s (p OUT VARCHAR2);
END pkg;
```

- Subprograms whose formal parameters differ only in subtype; for example:

```
PACKAGE pkg IS
  PROCEDURE s (p INTEGER);
  PROCEDURE s (p REAL);
END pkg;
```

  INTEGER and REAL are subtypes of NUMBER, so they belong to the same data type family.

- Functions that differ only in return value data type, even if the data types are in different families; for example:

```
PACKAGE pkg IS
  FUNCTION f (p INTEGER) RETURN BOOLEAN;
  FUNCTION f (p INTEGER) RETURN INTEGER;
END pkg;
```

## Subprogram Overload Errors

The PL/SQL compiler catches overload errors as soon as it can determine that it will be unable to tell which subprogram was invoked. When subprograms have identical headings, the compiler catches the overload error when you try to compile the subprograms themselves (if they are local) or when you try to compile the package specification that declares them (if they are packaged); otherwise, it catches the error when you try to compile an ambiguous invocation of a subprogram.

When you try to compile the package specification in Example 8–11, which declares subprograms with identical headings, you get compile-time error PLS-00305.

***Example 8–11 Overload Error That Causes Compile-Time Error***

```
CREATE OR REPLACE PACKAGE pkg1 IS
  PROCEDURE s (p VARCHAR2);
  PROCEDURE s (p VARCHAR2);
END pkg1;
/
```

Although the package specification in Example 8–12 violates the rule that you cannot overload subprograms whose formal parameters differ only in subtype, you can compile it without error.

***Example 8–12 Overload Error That Compiles Successfully***

```
CREATE OR REPLACE PACKAGE pkg2 IS
  SUBTYPE t1 IS VARCHAR2(10);
  SUBTYPE t2 IS VARCHAR2(10);
  PROCEDURE s (p t1);
  PROCEDURE s (p t2);
END pkg2;
/
```

However, when you try to compile an invocation of `pkg2.s`, as in Example 8–13, you get compile-time error PLS-00307.

***Example 8–13 Invocation of Improperly Overloaded Subprogram***

```
CREATE OR REPLACE PROCEDURE p IS
  a pkg2.t1 := 'a';
BEGIN
  pkg2.s(a);  -- Causes compile-time error PLS-00307
END p;
/
```

Suppose that you correct the overload error in Example 8–12 by giving the formal parameters of the overloaded subprograms different names, as in Example 8–14.

***Example 8–14 Properly Overloaded Subprogram***

```
CREATE OR REPLACE PACKAGE pkg2 IS
  SUBTYPE t1 IS VARCHAR2(10);
  SUBTYPE t2 IS VARCHAR2(10);
  PROCEDURE s (p1 t1);
  PROCEDURE s (p2 t2);
END pkg2;
/
```

Now you can compile an invocation of `pkg2.s` without error if you specify the actual parameter with named notation, as in Example 8–15. (If you specify the actual parameter with positional notation, as in Example 8–13, you still get compile-time error PLS-00307.)

***Example 8–15 Invocation of Properly Overloaded Subprogram***

```
CREATE OR REPLACE PROCEDURE p IS
  a pkg2.t1 := 'a';
BEGIN
  pkg2.s(p1=>a);  -- Compiles without error
END p;
/
```

The package specification in Example 8–16 violates no overload rules and compiles without error. However, you can still get compile-time error PLS-00307 when invoking its overloaded procedure, as in the second invocation in Example 8–17.

***Example 8–16 Package Specification Without Overload Errors***

```
CREATE OR REPLACE PACKAGE pkg3 IS
  PROCEDURE s (p1 VARCHAR2);
  PROCEDURE s (p1 VARCHAR2, p2 VARCHAR2 := 'p2');
END pkg3;
/
```

***Example 8–17 Improper Invocation of Properly Overloaded Subprogram***

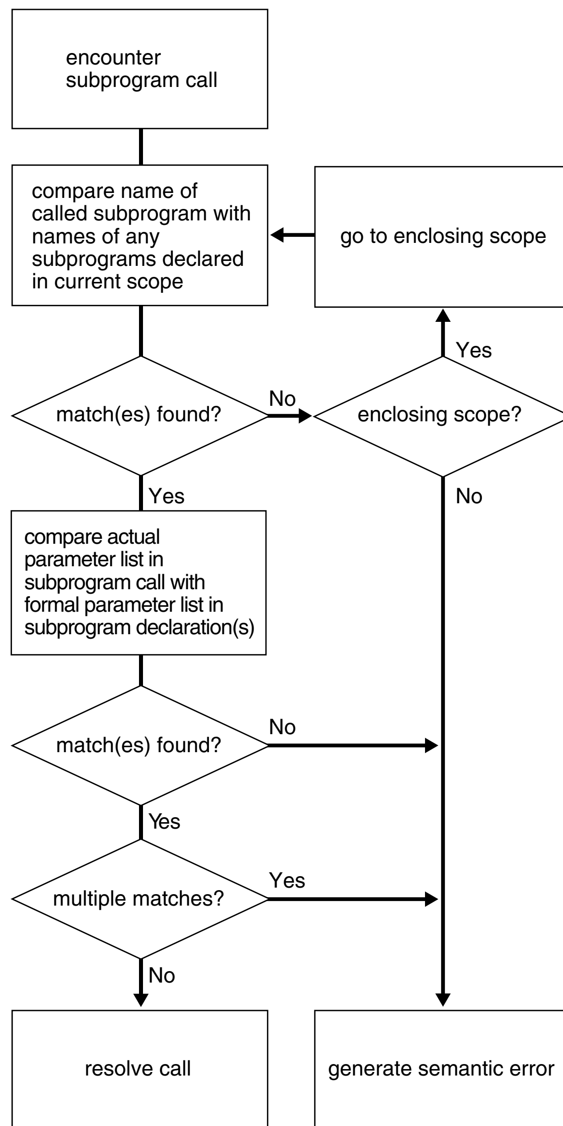
```
CREATE OR REPLACE PROCEDURE p IS
  a1 VARCHAR2(10) := 'a1';
  a2 VARCHAR2(10) := 'a2';
BEGIN
  pkg3.s(p1=>a1, p2=>a2);  -- Compiles without error
  pkg3.s(p1=>a1);          -- Causes compile-time error PLS-00307
END p;
/
```

# Subprogram Call Resolution

Figure 8–1 shows how the PL/SQL compiler resolves subprogram calls. When the compiler encounters a subprogram call, it tries to find a declaration that matches the call. The compiler searches first in the current scope and then, if necessary, in successive enclosing scopes. The compiler looks more closely when it finds one or more subprogram declarations in which the subprogram name matches the name of the called subprogram.

To resolve a call among possibly like-named subprograms at the same level of scope, the compiler must find an exact match between the actual and formal parameters. They must match in number, order, and data type (unless some formal parameters were assigned initial values). If no match is found or if multiple matches are found, the compiler generates a semantic error.

***Figure 8–1 How the PL/SQL Compiler Resolves Calls***




In Example 8–18, the function `balance` tries to invoke the enclosing procedure `swap`, using appropriate actual parameters. However, `balance` contains two local procedures named `swap`, and neither has parameters of the same type as the enclosing procedure `swap`. Therefore, the invocation causes compilation error PLS-00306.

***Example 8–18 Resolving PL/SQL Procedure Names***

```
DECLARE
  PROCEDURE swap (
    n1 NUMBER,
    n2 NUMBER
  )
  IS
    num1 NUMBER;
    num2 NUMBER;

    FUNCTION balance
      (bal NUMBER)
```

```
      RETURN NUMBER
    IS
      x NUMBER := 10;

      PROCEDURE swap (
        d1 DATE,
        d2 DATE
      ) IS
      BEGIN
        NULL;
      END;

      PROCEDURE swap (
        b1 BOOLEAN,
        b2 BOOLEAN
      ) IS
      BEGIN
        NULL;
      END;

    BEGIN  -- balance
      swap(num1, num2);  -- Causes compile-time error PLS-00306
      RETURN x;
    END balance;

  BEGIN  -- enclosing procedure swap
    NULL;
  END swap;

BEGIN  -- anonymous block
  NULL;
END;   -- anonymous block
/
```

## Invoker's Rights and Definer's Rights (AUTHID Property)

The AUTHID property of a stored PL/SQL unit affects the name resolution and privilege checking of SQL statements that the unit issues at run time. The AUTHID property does not affect compilation, and has no meaning for units that have no code, such as collection types.

AUTHID property values are exposed in the static data dictionary view *_PROCEDURES. For units for which AUTHID has meaning, the view shows the value CURRENT_USER or DEFINER; for other units, the view shows NULL.

For stored PL/SQL units that you create or alter with the following statements, you can use the optional AUTHID clause to specify either DEFINER (the default) or CURRENT_USER:


- "CREATE FUNCTION Statement" on page 14-32
- "CREATE PACKAGE Statement" on page 14-42
- "CREATE PROCEDURE Statement" on page 14-49
- "CREATE TYPE Statement" on page 14-69
- "ALTER TYPE Statement" on page 14-17


A unit whose AUTHID value is CURRENT_USER is called an **invoker's rights unit**, or **IR unit**. A unit whose AUTHID value is DEFINER is called a **definer's rights unit**, or **DR**

**unit**. An anonymous block always behaves like an IR unit. A trigger or view always behaves like a DR unit.

The AUTHID property of a unit determines whether the unit is IR or DR, and it affects both name resolution and privilege checking at run time:

- The context for name resolution is CURRENT_SCHEMA.
- The privileges checked are those of the CURRENT_USER and the enabled roles.

When a session starts, CURRENT_SCHEMA has the value of the schema owned by SESSION_USER, and CURRENT_USER has the same value as SESSION_USER. (To get the current value of CURRENT_SCHEMA, CURRENT_USER, or SESSION_USER, use the SYS_CONTEXT function, documented in *Oracle Database SQL Language Reference*.)

CURRENT_SCHEMA can be changed during the session with the SQL statement ALTER SESSION SET CURRENT_SCHEMA. CURRENT_USER cannot be changed programmatically, but it might change when a PL/SQL unit or a view is pushed onto, or popped from, the call stack.

> **Note:** Oracle recommends against issuing ALTER SESSION SET CURRENT_SCHEMA from in a stored PL/SQL unit.

During a server call, when a DR unit is pushed onto the call stack, the database stores the currently enabled roles and the current values of CURRENT_USER and CURRENT_SCHEMA. It then changes both CURRENT_USER and CURRENT_SCHEMA to the owner of the DR unit, and enables only the role PUBLIC. (The stored and new roles and values are not necessarily different.) When the DR unit is popped from the call stack, the database restores the stored roles and values. In contrast, when an IR unit is pushed onto, or popped from, the call stack, the values of CURRENT_USER and CURRENT_SCHEMA, and the currently enabled roles do not change.

For dynamic SQL statements issued by a PL/SQL unit, name resolution and privilege checking are done once, at run time. For static SQL statements, name resolution and privilege checking are done twice: first, when the PL/SQL unit is compiled, and then again at run time. At compilation time, the AUTHID property has no effect—both DR and IR units are treated like DR units. At run time, however, the AUTHID property determines whether a unit is IR or DR, and the unit is treated accordingly.

Topics:


- Choosing AUTHID CURRENT_USER or AUTHID DEFINER
- AUTHID and SQL Command SET ROLE
- Need for Template Objects in IR Units
- Overriding Default Name Resolution in IR Units
- IR Subprograms, Views, and Database Triggers
- IR Database Links
- IR ADTs
- IR Instance Methods

## Choosing AUTHID CURRENT_USER or AUTHID DEFINER

**Scenario:** Suppose that you want to create an API whose procedures have unrestricted access to its tables, but you want to prevent ordinary users from selecting table data directly, and from changing it with `INSERT`, `UPDATE`, and `DELETE` statements.

**Solution:** In a special schema, create the tables and the procedures that comprise the API. By default, each procedure is a DR unit, so you need not specify `AUTHID DEFINER` when you create it. To other users, grant the `EXECUTE` privilege, but do not grant any privileges that allow data access.

**Scenario:** Suppose that you want to write a PL/SQL procedure that presents compilation errors to a developer. The procedure is to join the static data dictionary views `ALL_SOURCE` and `ALL_ERRORS` and use the procedure `DBMS_OUTPUT.PUT_LINE` to show a window of numbered source lines around each error, following the list of errors for that window. You want all developers to be able to run the procedure, and you want the procedure to treat each developer as the `CURRENT_USER` for `ALL_SOURCE` and `ALL_ERRORS`.

**Solution:** When you create the procedure, specify `AUTHID CURRENT_USER`. Grant the `EXECUTE` privilege to `PUBLIC`. Because the procedure is an IR unit, `ALL_SOURCE` and `ALL_ERRORS` operate from the perspective of the user who invokes the procedure.

---

**Note:** Another solution is to make the procedure a DR unit and grant its owner the `SELECT` privilege on both `DBA_SOURCE` and `DBA_ERRORS`. However, this solution is harder to program, and far harder to audit for the criterion that a user must never see source code for units for which he or she does not have the `EXECUTE` privilege.

---

## AUTHID and SQL Command SET ROLE

The SQL command `SET ROLE` succeeds only if there are no DR units on the call stack. If at least one DR unit is on the call stack, issuing the `SET ROLE` command causes ORA-06565.

---

**Note:** To run the `SET ROLE` command from PL/SQL, you must use dynamic SQL, preferably the `EXECUTE IMMEDIATE` statement. For information about this statement, see "EXECUTE IMMEDIATE Statement" on page 7-2.

---

## Need for Template Objects in IR Units

The PL/SQL compiler must resolve all references to tables and other objects at compile time. The owner of an IR unit must have objects in the same schema with the right names and columns, even if they do not contain any data. At run time, the corresponding objects in the invoker's schema must have matching definitions. Otherwise, you get an error or unexpected results, such as ignoring table columns that exist in the invoker's schema but not in the schema that contains the unit.

## Overriding Default Name Resolution in IR Units

Sometimes, the run-time name resolution rules for an IR unit (that cause different invocations to resolve the same unqualified name to different objects) are not desired. Rather, it is required that a specific object be used on every invocation. Nevertheless, an IR unit is needed for other reasons. For example, it might be critical that privileges

are evaluated for the CURRENT_USER. Under these circumstances, qualify the name with the schema that owns the object.

An unqualified name for a public synonym is exposed to the risk of capture if the schema of the CURRENT_USER has a colliding name. A public synonym can be qualified with "PUBLIC". You must enclose PUBLIC in double quotation marks. For example:

```
DECLARE
  today  DATE;
BEGIN
  SELECT sysdate INTO today FROM "PUBLIC".DUAL;
END;
/
```

---

**Note:** Oracle recommends against issuing the SQL statement ALTER SESSION SET CURRENT_SCHEMA from in a stored PL/SQL unit.

---

## IR Subprograms, Views, and Database Triggers

For IR subprograms run in a view expression, the user who created the view, not the user who is querying the view, is considered to be the current user. This rule also applies to database triggers.

---

**Note:** If SYS_CONTEXT is used directly in the defining SQL statement of a view, then the value it returns for CURRENT_USER is the querying user and not the owner of the view.

---

## IR Database Links

You can create a database link to use invoker's rights:

```
CREATE DATABASE LINK link_name CONNECT TO CURRENT_USER
  USING connect_string;
```

A current-user link lets you connect to a remote database as another user, with that user's privileges. To connect, the database uses the user name of the current user (who must be a global user). Suppose an IR subprogram owned by user OE references this database link:

```
CREATE DATABASE LINK dallas CONNECT TO CURRENT_USER USING ...
```

If global user HR invokes the subprogram, it connects to the Dallas database as user HR, who is the current user. If it were a definer's rights subprogram, the current user would be OE, and the subprogram would connect to the Dallas database as global user OE.

## IR ADTs

To define ADTs for use in any schema, specify the AUTHID CURRENT_USER clause. For information about ADTs, see *Oracle Database Object-Relational Developer's Guide*.

Suppose that user HR creates the ADT in Example 8–19.

***Example 8–19 Creating an ADT with AUTHID CURRENT USER***

```
CREATE TYPE person_typ AUTHID CURRENT_USER AS OBJECT (
  person_id    NUMBER,
```

```
  person_name  VARCHAR2(30),
  person_job   VARCHAR2(10),

  STATIC PROCEDURE new_person_typ (
    person_id NUMBER,
    person_name VARCHAR2,
    person_job VARCHAR2,
    schema_name VARCHAR2,
    table_name VARCHAR2
  ),

  MEMBER PROCEDURE change_job (
   SELF IN OUT NOCOPY person_typ,
   new_job VARCHAR2
  )
);
/
CREATE TYPE BODY person_typ AS
  STATIC PROCEDURE new_person_typ (
    person_id NUMBER,
    person_name VARCHAR2,
    person_job VARCHAR2,
    schema_name VARCHAR2,
    table_name VARCHAR2
  )
  IS
    sql_stmt VARCHAR2(200);
  BEGIN
    sql_stmt := 'INSERT INTO ' || schema_name || '.'
       || table_name || ' VALUES (HR.person_typ(:1, :2, :3))';

    EXECUTE IMMEDIATE sql_stmt
      USING person_id, person_name, person_job;
  END;

  MEMBER PROCEDURE change_job (
    SELF IN OUT NOCOPY person_typ,
    new_job VARCHAR2
  )
  IS
  BEGIN
    person_job := new_job;
  END;
END;
/
```

Then user `HR` grants the `EXECUTE` privilege on `person_typ` to user `OE`:

```
GRANT EXECUTE ON person_typ TO OE;
```

User `OE` creates an object table to store objects of type `person_typ` and then invokes procedure `new_person_typ` to populate the table:

```
DROP TABLE person_tab;
CREATE TABLE person_tab OF hr.person_typ;

BEGIN
  hr.person_typ.new_person_typ(1001,
                               'Jane Smith',
                               'CLERK',
                               'oe',
```

```
                                'person_tab');
  hr.person_typ.new_person_typ(1002,
                                'Joe Perkins',
                                'SALES',
                                'oe',
                                'person_tab');
  hr.person_typ.new_person_typ(1003,
                                'Robert Lange',
                                'DEV',
                                'oe',
                                'person_tab');
END;
/
```

The calls succeed because the procedure runs with the privileges of its current user (`OE`), not its owner (`HR`).

For subtypes in an ADT hierarchy, these rules apply:

- If a subtype does not explicitly specify an `AUTHID` clause, it inherits the `AUTHID` of its supertype.
- If a subtype does specify an `AUTHID` clause, its `AUTHID` must match the `AUTHID` of its supertype. Also, if the `AUTHID` is `DEFINER`, both the supertype and subtype must have been created in the same schema.

## IR Instance Methods

An IR instance method runs with the privileges of the invoker, not the creator of the instance. Suppose that `person_typ` is the IR ADT created in Example 8–19 and user `HR` creates `p1`, an object of type `person_typ`. If user `OE` invokes instance method `change_job` to operate on object `p1`, the current user of the method is `OE`, not `HR`, as Example 8–20 shows.

***Example 8–20 Invoking an IR Instance Method***

```
-- OE creates procedure that invokes change_job:

CREATE OR REPLACE PROCEDURE reassign (
  p IN OUT NOCOPY hr.person_typ,
  new_job VARCHAR2
) AS
BEGIN
  p.change_job(new_job);  -- runs with privileges of OE
END;
/
-- OE grants EXECUTE privilege on procedure reassign to HR:

GRANT EXECUTE ON reassign to HR;

-- HR passes person_typ object to procedure reassign:

DECLARE
  p1  person_typ;
BEGIN
  p1 := person_typ(1004, 'June Washburn', 'SALES');
  oe.reassign(p1, 'CLERK');  -- current user is OE, not HR
END;
/
```

## Recursive Subprograms

A recursive subprogram is one that invokes itself. Each recursive call creates an instance of each item declared in the subprogram, including parameters, variables, cursors, and exceptions. Likewise, instances of SQL statements are created at each level in the recursive descent.

Be careful where you place a recursive call. If you place it inside a cursor `FOR` loop or between `OPEN` and `CLOSE` statements, another cursor is opened at each call, which might exceed the limit set by the database initialization parameter `OPEN_CURSORS`.

There must be at least two paths through a recursive subprogram: one that leads to the recursive call and one that does not. At least one path must lead to a terminating condition. Otherwise, the recursion continues until PL/SQL runs out of memory and raises the predefined exception `STORAGE_ERROR`.

Recursion is a powerful technique for simplifying the design of algorithms. Basically, recursion means self-reference. In a recursive mathematical sequence, each term is derived by applying a formula to preceding terms. The Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, 21, ...), is an example. Each term in the sequence (after the second) is the sum of the two terms that immediately precede it.

In a recursive definition, something is defined as simpler versions of itself. Consider the definition of *n* factorial (*n!*), the product of all integers from 1 to *n*:

```
n! = n * (n - 1)!
```

## External Subprograms

Although PL/SQL is a powerful, flexible language, some tasks are more easily done in another language, such as C or Java.

External C subprograms extend the functionality of the database server, and move computation-bound programs from client to server, where they run faster. External C subprograms are useful for interfacing with embedded systems, solving engineering problems, analyzing data, and controlling real-time devices and processes.

Java and other widely used languages have reusable libraries for common design patterns.

You can use PL/SQL call specifications to invoke external subprograms written in other languages, making their capabilities and libraries available from PL/SQL. For example, you can invoke Java stored procedures from any PL/SQL block, subprogram, or package.

If this Java class is stored in the database, it can be invoked as shown in Example 8–21:

```
import java.sql.*;
import oracle.jdbc.driver.*;
public class Adjuster {
  public static void raiseSalary (int empNo, float percent)
  throws SQLException {
    Connection conn = new OracleDriver().defaultConnection();
    String sql = "UPDATE employees SET salary = salary * ?
                  WHERE employee_id = ?";
    try {
      PreparedStatement pstmt = conn.prepareStatement(sql);
      pstmt.setFloat(1, (1 + percent / 100));
      pstmt.setInt(2, empNo);
      pstmt.executeUpdate();
      pstmt.close();
```

```
    } catch (SQLException e)
          {System.err.println(e.getMessage());}
    }
}
```

The class `Adjuster` has one method, which raises the salary of an employee by a given percentage. Because `raiseSalary` is a `void` method, you publish it as a procedure using the call specification shown in Example 8–21 and then invoke the procedure `raise_salary` from an anonymous PL/SQL block.

***Example 8–21 Invoking an External Procedure from PL/SQL***

```
CREATE OR REPLACE PROCEDURE raise_salary (
  empid NUMBER,
  pct NUMBER
)
AS LANGUAGE JAVA
NAME 'Adjuster.raiseSalary(int, float)';
/

DECLARE
  emp_id   NUMBER := 120;
  percent  NUMBER := 10;
BEGIN
  raise_salary(emp_id, percent);
END;
/
```

Java call specifications cannot be declared as nested procedures, but can be specified in ADT specifications and bodies, PL/SQL package specifications, PL/SQL package bodies, and as schema-level PL/SQL procedures and functions.

Example 8–22 creates a Java call specification and invokes the Java function from a PL/SQL procedure.

***Example 8–22 Invoking a Java Function from PL/SQL***

```
-- Java call specification:

CREATE PROCEDURE java_sleep (
  milli_seconds IN NUMBER
) AS LANGUAGE JAVA NAME 'java.lang.Thread.sleep(long)';
/

CREATE OR REPLACE PROCEDURE sleep (
  milli_seconds IN NUMBER
) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.get_time());
  java_sleep (milli_seconds);
  DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.get_time());
END;
/
```

> **See Also:** *Oracle Database Advanced Application Developer's Guide* for more information about multilanguage programs

## Subprogram Side Effects

The fewer side effects a function has, the better it can be optimized in a query, particularly when the PARALLEL_ENABLE or DETERMINISTIC options are used. For the syntax of these options, see "Function" on page 13-76.

To be callable from SQL statements, a stored function (and any subprograms that it invokes) must obey these purity rules, which are meant to control side effects:

- When invoked from a SELECT statement or a parallelized INSERT, UPDATE, or DELETE statement, the function cannot modify any database tables.
- When invoked from an INSERT, UPDATE, or DELETE statement, the function cannot query or modify any database tables modified by that statement.
- When invoked from a SELECT, INSERT, UPDATE, or DELETE statement, the function cannot run SQL transaction control statements (such as COMMIT), session control statements (such as SET ROLE), or system control statements (such as ALTER SYSTEM). Also, it cannot run DDL statements (such as CREATE) because they are followed by an automatic commit.

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

## Subprogram Parameter Aliasing

The NOCOPY hint (explained in "Subprogram Parameter Passing Methods" on page 8-7) increases the possibility of **aliasing**—that is, having two different names refer to the same memory location. This can occur when a global variable appears as an actual parameter in a subprogram call and then is referenced in the subprogram. The result is indeterminate, because the optimizer can either obey or disregard the NOCOPY hint.

In Example 8–23, procedure add_entry references varray lexicon both as a parameter and as a global variable. When add_entry is invoked, the identifiers word_list and lexicon point to the same varray.

***Example 8–23 Aliasing from Passing Global Variable with NOCOPY Hint***

```
DECLARE
  TYPE Definition IS RECORD (
    word     VARCHAR2(20),
    meaning  VARCHAR2(200)
  );

  TYPE Dictionary IS VARRAY(2000) OF Definition;

  lexicon  Dictionary := Dictionary();

  PROCEDURE add_entry (
    word_list IN OUT NOCOPY Dictionary
  ) IS
  BEGIN
    word_list(1).word := 'aardvark';
    lexicon(1).word := 'aardwolf';
  END;

BEGIN
  lexicon.EXTEND;
  add_entry(lexicon);
```

```
  DBMS_OUTPUT.PUT_LINE(lexicon(1).word);
END;
/
```

The program prints aardwolf if the compiler obeys the NOCOPY hint. The assignment to WORD_LIST is done immediately through a pointer, then is overwritten by the assignment to LEXICON.

The program prints aardvark if the NOCOPY hint is omitted, or if the compiler does not obey the hint. The assignment to WORD_LIST uses an internal copy of the varray, which is copied back to the actual parameter (overwriting the contents of LEXICON) when the procedure ends.

Aliasing can also occur when the same actual parameter appears more than once in a subprogram call. In Example 8–24, n2 is an IN OUT parameter, so the value of the actual parameter is not updated until the procedure exits. That is why the first PUT_LINE prints 10 (the initial value of n) and the third PUT_LINE prints 20. However, n3 is a NOCOPY parameter, so the value of the actual parameter is updated immediately. That is why the second PUT_LINE prints 30.

***Example 8–24 Aliasing Passing Same Parameter Multiple Times***

```
DECLARE
  n NUMBER := 10;

  PROCEDURE do_something (
    n1 IN NUMBER,
    n2 IN OUT NUMBER,
    n3 IN OUT NOCOPY NUMBER) IS
  BEGIN
    n2 := 20;
    DBMS_OUTPUT.put_line(n1);  -- prints 10
    n3 := 30;
    DBMS_OUTPUT.put_line(n1);  -- prints 30
  END;

BEGIN
  do_something(n, n, n);
  DBMS_OUTPUT.put_line(n);  -- prints 20
END;
/
```

Because they are pointers, cursor variables also increase the possibility of aliasing. In Example 8–25, after the assignment, emp_cv2 is an alias of emp_cv1; both point to the same query work area. The first fetch from emp_cv2 fetches the third row, not the first, because the first two rows were fetched from emp_cv1. The second fetch from emp_cv2 fails because emp_cv1 is closed.

***Example 8–25 Aliasing from Assigning Cursor Variables to Same Work Area***

```
DECLARE
  TYPE EmpCurTyp IS REF CURSOR;
  c1 EmpCurTyp;
  c2 EmpCurTyp;

  PROCEDURE get_emp_data (
    emp_cv1 IN OUT EmpCurTyp,
    emp_cv2 IN OUT EmpCurTyp
  )
  IS
```

```
    emp_rec employees%ROWTYPE;
  BEGIN
    OPEN emp_cv1 FOR SELECT * FROM employees;
    emp_cv2 := emp_cv1;
    FETCH emp_cv1 INTO emp_rec;
    DBMS_OUTPUT.put_line('Fetched first row');
    FETCH emp_cv1 INTO emp_rec;
    DBMS_OUTPUT.put_line('Fetched second row');
    FETCH emp_cv2 INTO emp_rec;
    DBMS_OUTPUT.put_line('Fetched third row');
    CLOSE emp_cv1;
    FETCH emp_cv2 INTO emp_rec; -- causes error when get_emp_data is invoked
  END;
BEGIN
  get_emp_data(c1, c2);
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-01001: invalid cursor
ORA-06512: at line 22
ORA-06512: at line 25
```

## PL/SQL Function Result Cache

The PL/SQL function result caching mechanism provides a language-supported and system-managed means for caching 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 relieves you of the burden of designing and developing your own caches and cache-management policies.

To enable result-caching for a function, use the `RESULT_CACHE` clause. When a result-cached function is invoked, the system checks the cache. If the cache contains the result from a previous call to the function with the same parameter values, the system returns the cached result to the invoker and does not reexecute the function body. If the cache does not contain the result, the system runs the function body and adds the result (for these parameter values) to the cache before returning control to the invoker.

> **Note:** If function execution results in an unhandled exception, the exception result is not stored in the cache.

The cache can accumulate very many results—one result for every unique combination of parameter values with which each result-cached function was invoked. If the system needs more memory, it **ages out** (deletes) one or more cached results.

Oracle Database automatically detects all data sources (tables and views) that are queried while a result-cached function is running. If changes to any of these data sources are committed, the cached result becomes invalid and must be recomputed. The best candidates for result-caching are functions that are invoked frequently but depend on information that changes infrequently or never.

Topics:

- Enabling Result-Caching for a Function
- Developing Applications with Result-Cached Functions
- Restrictions on Result-Cached Functions
- Examples of Result-Cached Functions
- Advanced Result-Cached Function Topics


## Enabling Result-Caching for a Function

To make a function result-cached, include the RESULT_CACHE clause in the function definition. (If you declare the function before defining it, you must also include the RESULT_CACHE option in the function declaration.) For syntax details, see "Function" on page 13-76.

In Example 8–26, the package department_pkg declares and then defines a result-cached function, get_dept_info, which returns the average salary and number of employees in a given department. get_dept_info depends on the database table EMPLOYEES.

***Example 8–26 Declaration and Definition of Result-Cached Function***

```
-- Package specification
CREATE OR REPLACE PACKAGE department_pkg IS
  TYPE dept_info_record IS RECORD (average_salary      NUMBER,
                                   number_of_employees NUMBER);
  -- Function declaration
  FUNCTION get_dept_info (dept_id NUMBER) RETURN dept_info_record
  RESULT_CACHE;
END department_pkg;
/
CREATE OR REPLACE PACKAGE BODY department_pkg AS
  -- Function definition
  FUNCTION get_dept_info (dept_id NUMBER) RETURN dept_info_record
    RESULT_CACHE;
  IS
    rec dept_info_record;
  BEGIN
    SELECT AVG(SALARY), COUNT(*) INTO rec
      FROM EMPLOYEES
        WHERE DEPARTMENT_ID = dept_id;
    RETURN rec;
  END get_dept_info;
END department_pkg;
/
DECLARE
  dept_id   NUMBER := 50;
  avg_sal   NUMBER;
  no_of_emp NUMBER;
BEGIN
  avg_sal   := department_pkg.get_dept_info(50).average_salary;
  no_of_emp := department_pkg.get_dept_info(50).number_of_employees;
  DBMS_OUTPUT.PUT_LINE('dept_id = ' ||dept_id);
  DBMS_OUTPUT.PUT_LINE('average_salary = '|| avg_sal);
  DBMS_OUTPUT.PUT_LINE('number_of_employees = ' ||no_of_emp);
END;
/
```

You invoke the function get_dept_info as you invoke any function. For example, this call returns the average salary and the number of employees in department number 10:

```
department_pkg.get_dept_info(10);
```

This call returns only the average salary in department number 10:

```
department_pkg.get_dept_info(10).average_salary;
```

If the result for get_dept_info(10) is in the result cache, the result is returned from the cache; otherwise, the result is computed and added to the cache. Because get_dept_info depends on the EMPLOYEES table, any committed change to EMPLOYEES invalidates all cached results for get_dept_info, relieving you of programming cache invalidation logic everywhere that EMPLOYEES might change.

## Developing Applications with Result-Cached Functions

When developing an application that uses a result-cached function, make no assumptions about the number of times the body of the function will run for a given set of parameter values.

Some situations in which the body of a result-cached function runs are:

- The first time a session on this database instance invokes the function with these parameter values
- When the cached result for these parameter values is **invalid**

  When a change to any data source on which the function depends is committed, the cached result becomes invalid.
- When the cached results for these parameter values have aged out

  If the system needs memory, it might discard the oldest cached values.
- When the function bypasses the cache (see "Result Cache Bypass" on page 8-35)

## Restrictions on Result-Cached Functions

To be result-cached, a function must meet all of these criteria:

- It is not defined in a module that has invoker's rights or in an anonymous block.
- It is not a pipelined table function.
- It does not reference dictionary tables, temporary tables, sequences, or nondeterministic SQL functions.

  For more information, see *Oracle Database Performance Tuning Guide*.
- It has no OUT or IN OUT parameters.
- No IN parameter has one of these types:
  - BLOB
  - CLOB
  - NCLOB
  - REF CURSOR
  - Collection
  - Object

  - Record
- The return type is none of these:
  - BLOB
  - CLOB
  - NCLOB
  - REF CURSOR
  - Object
  - Record or PL/SQL collection that contains an unsupported return type

It is recommended that a result-cached function also meet these criteria:

- It has no side effects.

  For example, it does not modify the database state, or modify the external state by invoking DBMS_OUTPUT or sending e-mail.
- It does not depend on session-specific settings.

  For more information, see "Making Result-Cached Functions Handle Session-Specific Settings" on page 8-35.
- It does not depend on session-specific application contexts.

  For more information, see "Making Result-Cached Functions Handle Session-Specific Application Contexts" on page 8-36.

## Examples of Result-Cached Functions

The best candidates for result-caching are functions that are invoked frequently but depend on information that changes infrequently (as might be the case in the first example). Result-caching avoids redundant computations in recursive functions.

Examples:

- Result-Cached Application Configuration Parameters
- Result-Cached Recursive Function

### Result-Cached Application Configuration Parameters

Consider an application that has configuration parameters that can be set at either the global level, the application level, or the role level. The application stores the configuration information in these tables:

```
-- Global Configuration Settings
DROP TABLE global_config_params;
CREATE TABLE global_config_params
  (name  VARCHAR2(20), -- parameter NAME
   value VARCHAR2(20), -- parameter VALUE
   PRIMARY KEY (name)
  );

-- Application-Level Configuration Settings
CREATE TABLE app_level_config_params
  (app_id  VARCHAR2(20), -- application ID
   name    VARCHAR2(20), -- parameter NAME
   value   VARCHAR2(20), -- parameter VALUE
   PRIMARY KEY (app_id, name)
  );
```

```
-- Role-Level Configuration Settings
CREATE TABLE role_level_config_params
  (role_id  VARCHAR2(20), -- application (role) ID
   name     VARCHAR2(20),  -- parameter NAME
   value    VARCHAR2(20),  -- parameter VALUE
   PRIMARY KEY (role_id, name)
  );
```

For each configuration parameter, the role-level setting overrides the application-level setting, which overrides the global setting. To determine which setting applies to a parameter, the application defines the PL/SQL function `get_value`. Given a parameter name, application ID, and role ID, `get_value` returns the setting that applies to the parameter.

The function `get_value` is a good candidate for result-caching if it is invoked frequently and if the configuration information changes infrequently.

Example 8–27 shows a possible definition for `get_value`. Suppose that for one set of parameter values, the global setting determines the result of `get_value`. While `get_value` is running, the database detects that three tables are queried—`role_level_config_params`, `app_level_config_params`, and `global_config_params`. If a change to any of these three tables is committed, the cached result for this set of parameter values is invalidated and must be recomputed.

Now suppose that, for a second set of parameter values, the role-level setting determines the result of `get_value`. While `get_value` is running, the database detects that only the `role_level_config_params` table is queried. If a change to `role_level_config_params` is committed, the cached result for the second set of parameter values is invalidated; however, committed changes to `app_level_config_params` or `global_config_params` do not affect the cached result.

***Example 8–27 Result-Cached Function that Returns Configuration Parameter Setting***

```
CREATE OR REPLACE FUNCTION get_value
  (p_param VARCHAR2,
   p_app_id  NUMBER,
   p_role_id NUMBER
  )
  RETURN VARCHAR2
  RESULT_CACHE
IS
  answer VARCHAR2(20);
BEGIN
  -- Is parameter set at role level?
  BEGIN
    SELECT value INTO answer
      FROM role_level_config_params
        WHERE role_id = p_role_id
          AND name = p_param;
    RETURN answer;  -- Found
    EXCEPTION
      WHEN no_data_found THEN
        NULL;  -- Fall through to following code
  END;
  -- Is parameter set at application level?
  BEGIN
    SELECT value INTO answer
      FROM app_level_config_params
        WHERE app_id = p_app_id
```

```
          AND name = p_param;
    RETURN answer;  -- Found
    EXCEPTION
      WHEN no_data_found THEN
        NULL;  -- Fall through to following code
  END;
  -- Is parameter set at global level?
    SELECT value INTO answer
     FROM global_config_params
      WHERE name = p_param;
    RETURN answer;
END;
```

### Result-Cached Recursive Function

A recursive function for finding the *n*th term of a Fibonacci series that mirrors the mathematical definition of the series might do many redundant computations. For example, to evaluate `fibonacci(7)`, the function must compute `fibonacci(6)` and `fibonacci(5)`. To compute `fibonacci(6)`, the function must compute `fibonacci(5)` and `fibonacci(4)`. Therefore, `fibonacci(5)` and several other terms are computed redundantly. Result-caching avoids these redundant computations.

> **Note:** The maximum number of recursive calls cached is 128.

```
CREATE OR REPLACE FUNCTION fibonacci (n NUMBER)
   RETURN NUMBER RESULT_CACHE IS
BEGIN
  IF (n =0) OR (n =1) THEN
    RETURN 1;
  ELSE
    RETURN fibonacci(n - 1) + fibonacci(n - 2);
  END IF;
END;
/
```

## Advanced Result-Cached Function Topics

Topics:

- Rules for a Cache Hit
- Result Cache Bypass
- Making Result-Cached Functions Handle Session-Specific Settings
- Making Result-Cached Functions Handle Session-Specific Application Contexts
- Choosing Result-Caching Granularity
- Result Caches in Oracle RAC Environment
- Result Cache Management
- Hot-Patching PL/SQL Units on Which Result-Cached Functions Depend

### Rules for a Cache Hit

Each time a result-cached function is invoked with different parameter values, those parameters and their result are stored in the cache. Subsequently, when the same

function is invoked with the same parameter values (that is, when there is a **cache hit**), the result is retrieved from the cache, instead of being recomputed.

The rules for parameter comparison for a cache hit differ from the rules for the PL/SQL "equal to" (=) operator, as follows:

| Cache Hit Rules | "Equal To" Operator Rules |
|---|---|
| `NULL` equals `NULL` | `NULL = NULL` evaluates to `NULL`. |
| Non-null scalars are the same if and only if their values are identical; that is, if and only if their values have identical bit patterns on the given platform. For example, `CHAR` values `'AA'` and `'AA   '` are different. (This rule is stricter than the rule for the "equal to" operator.) | Non-null scalars can be equal even if their values do not have identical bit patterns on the given platform; for example, `CHAR` values `'AA'` and `'AA   '` are equal. |

### Result Cache Bypass

In some situations, the cache is bypassed. When the cache is bypassed:

- The function computes the result instead of retrieving it from the cache.
- The result that the function computes is not added to the cache.

Some examples of situations in which the cache is bypassed are:

- The cache is unavailable to all sessions.

  For example, the database administrator has disabled the use of the result cache during application patching (as in "Hot-Patching PL/SQL Units on Which Result-Cached Functions Depend" on page 8-40).

- A session is performing a DML statement on a table or view on which a result-cached function depends.

  The session bypasses the result cache for that function until the DML statement is completed—either committed or rolled back. If the statement is rolled back, the session resumes using the cache for that function.

  Cache bypass ensures that:

  - The user of each session sees his or her own uncommitted changes.
  - The PL/SQL function result cache has only committed changes that are visible to all sessions, so that uncommitted changes in one session are not visible to other sessions.

### Making Result-Cached Functions Handle Session-Specific Settings

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.

Consider this function:

***Example 8–28 Function that Depends on Session-Specific Settings***

```
CREATE OR REPLACE FUNCTION get_hire_date (emp_id NUMBER) RETURN VARCHAR
 RESULT_CACHE
IS
  date_hired DATE;
BEGIN
  SELECT hire_date INTO date_hired
```

```
    FROM HR.EMPLOYEES
      WHERE EMPLOYEE_ID = emp_id;
  RETURN TO_CHAR(date_hired);
END;
/
```

The preceding function, `get_hire_date`, uses the `TO_CHAR` function to convert a `DATE` item to a `VARCHAR` item. The function `get_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 `get_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 is probably incorrect.

Some possible solutions to this problem are:

- Change the return type of `get_hire_date` to `DATE` and have each session invoke 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');
  ```

- Add a format mask parameter to `get_hire_date`. For example:

  ```
  CREATE OR REPLACE FUNCTION get_hire_date
    (emp_id NUMBER, fmt VARCHAR) RETURN VARCHAR
    RESULT_CACHE
  IS
    date_hired DATE;
  BEGIN
    SELECT hire_date INTO date_hired
      FROM HR.EMPLOYEES
        WHERE EMPLOYEE_ID = emp_id;
    RETURN TO_CHAR(date_hired, fmt);
  END;
  /
  ```

### Making Result-Cached Functions Handle Session-Specific Application Contexts

An **application context**, which can be either global or session-specific, is a set of attributes and their values. A PL/SQL function depends on session-specific application contexts if it does one or more of the following:

- Directly invokes the built-in function `SYS_CONTEXT`, which returns the value of a specified attribute in a specified context
- Indirectly invokes `SYS_CONTEXT` by using Virtual Private Database (VPD) mechanisms for fine-grained security

  (For information about VPD, see *Oracle Database Security Guide*.)

The PL/SQL function result-caching feature does not automatically handle dependence on session-specific application contexts. If you must cache the results of a function that depends on session-specific application contexts, you must pass the application context to the function as a parameter. You can give the parameter an initial value, so that not every user must specify it.

In Example 8–29, assume that a table, `config_tab`, has a VPD policy that translates this query:

```
SELECT value FROM config_tab WHERE name = param_name;
```

To this query:

```
SELECT value FROM config_tab
WHERE name = param_name
AND app_id = SYS_CONTEXT('Config', 'App_ID');
```

***Example 8–29 Result-Cached Function that Depends on Session-Specific Application Context***

```
CREATE OR REPLACE FUNCTION get_param_value
  (param_name VARCHAR,
   appctx     VARCHAR  := SYS_CONTEXT('Config', 'App_ID')
  )
  RETURN VARCHAR
  RESULT_CACHE
IS
  rec VARCHAR(2000);
BEGIN
  SELECT value INTO rec
    FROM config_tab
      WHERE Name = param_name;
END;
/
```

## Choosing Result-Caching Granularity

PL/SQL provides the function result cache, but you choose the caching granularity. To understand the concept of granularity, consider the `Product_Descriptions` table in the Order Entry (`OE`) sample schema:

```
NAME                      NULL?      TYPE
----------------------    --------   ---------------
PRODUCT_ID                NOT NULL   NUMBER(6)
LANGUAGE_ID               NOT NULL   VARCHAR2(3)
TRANSLATED_NAME           NOT NULL   NVARCHAR2(50)
TRANSLATED_DESCRIPTION    NOT NULL   NVARCHAR2(2000)
```

The table has the name and description of each product in several languages. The unique key for each row is `PRODUCT_ID,LANGUAGE_ID`.

Suppose that you want to define a function that takes a `PRODUCT_ID` and a `LANGUAGE_ID` and returns the associated `TRANSLATED_NAME`. You also want to cache the translated names. Some of the granularity choices for caching the names are:

- One name at a time (finer granularity)
- One language at a time (coarser granularity)

***Table 8–3 Comparison of Finer and Coarser Caching Granularity***

| Finer Granularity | Coarser Granularity |
|---|---|
| Each function result corresponds to one logical result. | Each function result contains many logical subresults. |
| Stores only data that is needed at least once. | Might store data that is never used. |
| Each data item ages out individually. | One aged-out data item ages out the whole set. |
| Does not allow bulk loading optimizations. | Allows bulk loading optimizations. |

In Example 8–30 and Example 8–31, the function productName takes a PRODUCT_ID and a LANGUAGE_ID and returns the associated TRANSLATED_NAME. Each version of productName caches translated names, but at a different granularity.

In Example 8–30, get_product_name_1 is a result-cached function. Whenever get_product_name_1 is invoked with a different PRODUCT_ID and LANGUAGE_ID, it caches the associated TRANSLATED_NAME. Each call to get_product_name_1 adds at most one TRANSLATED_NAME to the cache.

***Example 8–30 Caching One Name at a Time (Finer Granularity)***

```
CREATE OR REPLACE FUNCTION get_product_name_1 (prod_id NUMBER, lang_id VARCHAR2)
  RETURN NVARCHAR2
  RESULT_CACHE
IS
  result VARCHAR2(50);
BEGIN
  SELECT translated_name INTO result
    FROM Product_Descriptions
      WHERE PRODUCT_ID = prod_id
        AND LANGUAGE_ID = lang_id;
  RETURN result;
END;
```

In Example 8–31, get_product_name_2 defines a result-cached function, all_product_names. Whenever get_product_name_2 invokes all_product_names with a different LANGUAGE_ID, all_product_names caches every TRANSLATED_NAME associated with that LANGUAGE_ID. Each call to all_product_names adds every TRANSLATED_NAME of at most one LANGUAGE_ID to the cache.

***Example 8–31 Caching Translated Names One Language at a Time (Coarser Granularity)***

```
CREATE OR REPLACE FUNCTION get_product_name_2 (prod_id NUMBER, lang_id VARCHAR2)
  RETURN NVARCHAR2
IS
  TYPE product_names IS TABLE OF NVARCHAR2(50) INDEX BY PLS_INTEGER;

  FUNCTION all_product_names (lang_id NUMBER) RETURN product_names
    RESULT_CACHE
  IS
    all_names product_names;
  BEGIN
    FOR c IN (SELECT * FROM Product_Descriptions
      WHERE LANGUAGE_ID = lang_id) LOOP
        all_names(c.PRODUCT_ID) := c.TRANSLATED_NAME;
    END LOOP;
    RETURN all_names;
  END;
BEGIN
  RETURN all_product_names(lang_id)(prod_id);
END;
```

### Result Caches in Oracle RAC Environment

Cached results are stored in the system global area (SGA). In an Oracle RAC environment, each database instance manages its own local function result cache. However, the contents of the local result cache are accessible to sessions attached to other Oracle RAC instances. If a required result is missing from the result cache of the local instance, the result might be retrieved from the local cache of another instance, instead of being locally computed.

The access pattern and work load of an instance determine the set of results in its local cache; therefore, the local caches of different instances can have different sets of results.

Although each database instance might have its own set of cached results, the mechanisms for handling invalid results are Oracle RAC environment-wide. If results were invalidated only in the local instance's result cache, other instances might use invalid results. For example, consider a result cache of item prices that are computed from data in database tables. If any of these database tables is updated in a way that affects the price of an item, the cached price of that item must be invalidated in every database instance in the Oracle RAC environment.

### Result Cache Management

The PL/SQL function result cache shares its administrative and manageability infrastructure with the Result Cache. For information about the Result Cache, see *Oracle Database Performance Tuning Guide*.

The database administrator can use the following to manage the Result Cache:

- `RESULT_CACHE_MAX_SIZE` and `RESULT_CACHE_MAX_RESULT` initialization parameters

  `RESULT_CACHE_MAX_SIZE` specifies the maximum amount of SGA memory (in bytes) that the Result Cache can use, and `RESULT_CACHE_MAX_RESULT` specifies the maximum percentage of the Result Cache that any single result can use. For more information about these parameters, see *Oracle Database Reference* and *Oracle Database Performance Tuning Guide*.

  **See Also:**

  - *Oracle Database Reference* for more information about `RESULT_CACHE_MAX_SIZE`
  - *Oracle Database Reference* for more information about `RESULT_CACHE_MAX_RESULT`
  - *Oracle Database Performance Tuning Guide* for more information about Result Cache concepts

- `DBMS_RESULT_CACHE` package

  The `DBMS_RESULT_CACHE` package provides an interface to allow the DBA to administer that part of the shared pool that is used by the SQL result cache and the PL/SQL function result cache. For more information about this package, see *Oracle Database PL/SQL Packages and Types Reference*.

- Dynamic performance views:

  - `[G]V$RESULT_CACHE_STATISTICS`
  - `[G]V$RESULT_CACHE_MEMORY`
  - `[G]V$RESULT_CACHE_OBJECTS`
  - `[G]V$RESULT_CACHE_DEPENDENCY`

  See *Oracle Database Reference* for more information about `[G]V$RESULT_CACHE_STATISTICS`, `[G]V$RESULT_CACHE_MEMORY`, `[G]V$RESULT_CACHE_OBJECTS`, and `[G]V$RESULT_CACHE_DEPENDENCY`.

### Hot-Patching PL/SQL Units on Which Result-Cached Functions Depend

When you hot-patch a PL/SQL unit on which a result-cached function depends (directly or indirectly), the cached results associated with the result-cached function might not be automatically flushed in all cases.

For example, suppose that the result-cached function `P1.foo()` depends on the packaged subprogram `P2.bar()`. If a new version of the body of package `P2` is loaded, the cached results associated with `P1.foo()` are not automatically flushed.

Therefore, this is the recommended procedure for hot-patching a PL/SQL unit:

> **Note:** To follow these steps, you must have the `EXECUTE` privilege on the package `DBMS_RESULT_CACHE`.

1. Put the result cache in bypass mode and flush existing results:

   ```
   BEGIN
     DBMS_RESULT_CACHE.Bypass(TRUE);
     DBMS_RESULT_CACHE.Flush;
   END;
   /
   ```

   In an Oracle RAC environment, perform this step for each database instance.

2. Patch the PL/SQL code.

3. Resume using the result cache:

   ```
   BEGIN
     DBMS_RESULT_CACHE.Bypass(FALSE);
   END;
   /
   ```

   In an Oracle RAC environment, perform this step for each database instance.

# 9 PL/SQL Triggers

A **trigger** is a named PL/SQL unit that is stored in the database and run (**fired**) in response to a specified event that occurs in the database.

> **Note:** This chapter explains noncrossedition triggers. For information about crossedition triggers, see *Oracle Database Advanced Application Developer's Guide*.

Topics:


- Overview of Triggers
- Trigger Design Guidelines
- Trigger Creation
- Trigger Body
- Trigger Compilation
- Trigger Replacement
- Trigger Debugging
- Trigger Enabling
- Trigger Disabling
- Views for Information About Triggers
- Trigger Application Examples
- Triggers that Publish Database Events


## Overview of Triggers

A trigger is a named program unit that is stored in the database and **fired** (run) in response to a specified event. The specified **event** is associated with either a table, a view, a schema, or the database, and it is one of these:

- A database manipulation language (DML) statement (`DELETE`, `INSERT`, or `UPDATE`)
- A database definition language (DDL) statement (`CREATE`, `ALTER`, or `DROP`)
- A database operation (`SERVERERROR`, `LOGON`, `LOGOFF`, `STARTUP`, or `SHUTDOWN`)

The trigger is said to be **defined on** the table, view, schema, or database.

Topics:


- Trigger Types
- OLD and NEW Pseudorecords
- Trigger States
- Data Access for Triggers
- Trigger Uses


## Trigger Types

A **DML trigger** is fired by a DML statement, a **DDL trigger** is fired by a DDL statement, a **`DELETE` trigger** is fired by a `DELETE` statement, and so on.

An **`INSTEAD OF` trigger** is a DML trigger that is defined on a noneditioning view (not an editioning view or table). Oracle Database fires the `INSTEAD OF` trigger instead of running the triggering DML statement. For more information, see "INSTEAD OF Triggers" on page 9-8.

A **system trigger** is defined on a schema or the database. A trigger defined on a schema fires for each event associated with the owner of the schema (the current user). A trigger defined on a database fires for each event associated with all users.

A **simple trigger** can fire at exactly one of these **timing points**:

- Before the triggering event runs (statement-level `BEFORE` trigger)
- After the triggering event runs (statement-level `AFTER` trigger)
- Before each row that the event affects (row-level `BEFORE` trigger)
- After each row that the event affects (row-level `AFTER` trigger)

A **compound trigger** can fire at multiple timing points. Compound triggers make it easier to program an approach where you want the actions you implement for the various timing points to share common data. For more information, see "Compound Triggers" on page 9-14.

## OLD and NEW Pseudorecords

When a row-level trigger fires, the PL/SQL run-time system creates and populates the two pseudorecords `OLD` and `NEW`. They are called pseudorecords because they have some, but not all, of the properties of records. Their structure is *`table_name`*`%ROWTYPE`, where *`table_name`* is the name of the table on which the trigger is defined.

For the row that the trigger is processing:

- For an `INSERT` trigger, `OLD` contains no values, and `NEW` contains the new values.
- For an `UPDATE` trigger, `OLD` contains the old values, and `NEW` contains the new values.
- For a `DELETE` trigger, `OLD` contains the old values, and `NEW` contains no values.

To reference a pseudorecord, put a colon before its name—`:OLD` or `:NEW`—as in Example 9–1.

The ways in which pseudorecords are *not* like records are:

- You cannot change the values of `OLD`.

  (You can change the values of `NEW`.)

- You cannot pass OLD or NEW as a record parameter to a subprogram that the trigger calls.

  (You can pass fields of OLD and NEW as parameters to such subprograms.)

- You cannot perform record-level operations with OLD and NEW. For example, you cannot include this statement in a trigger:

  ```
  :NEW := NULL;
  ```

## Trigger States

A trigger can be in either of two states:

**Enabled**. An enabled trigger runs its trigger body if a triggering statement is entered and the trigger restriction (if any) evaluates to TRUE.

**Disabled**. A disabled trigger does not run its trigger body, even if a triggering statement is entered and the trigger restriction (if any) evaluates to TRUE.

By default, a trigger is created in enabled state. To create a trigger in disabled state, use the DISABLE clause of the CREATE TRIGGER statement.

**See Also:** "CREATE TRIGGER Statement" on page 14-53

## Data Access for Triggers

When a trigger is fired, the tables referenced in the trigger action might be undergoing changes by SQL statements in other users' transactions. In all cases, the SQL statements running in triggers follow the common rules used for standalone SQL statements. In particular, if an uncommitted transaction has modified values that a trigger being fired either must read (query) or write (update), then the SQL statements in the body of the trigger being fired use these guidelines:

- Queries see the current read-consistent materialized view of referenced tables and any data changed in the same transaction.
- Updates wait for existing data locks to be released before proceeding.

## Trigger Uses

Triggers supplement the standard capabilities of your database to provide a highly customized database management system. For example, you can use triggers to:

- Automatically generate derived column values
- Enforce referential integrity across nodes in a distributed database
- Enforce complex business rules
- Provide transparent event logging
- Provide auditing
- Maintain synchronous table replicates
- Gather statistics on table access
- Modify table data when DML statements are issued against views
- Publish information about database events, user events, and SQL statements to subscribing applications
- Restrict DML operations against a table to those issued during regular business hours

- Enforce security authorizations
- Prevent invalid transactions

---

**Caution:** Triggers are not reliable security mechanisms, because they are programmatic and easy to disable. For high assurance security, use Oracle Database Vault. For more information, see *Oracle Database Vault Administrator's Guide*.

---

## Trigger Design Guidelines

Use these guidelines when designing triggers:

- Use triggers to guarantee that when a specific operation is performed, related actions are performed.
- Do not define triggers that duplicate database features.

  For example, do not define triggers to reject bad data if you can do the same checking through constraints.

  Although you can use both triggers and integrity constraints to define and enforce any type of integrity rule, Oracle strongly recommends that you use triggers to constrain data input only in these situations:

  - To enforce referential integrity when child and parent tables are on different nodes of a distributed database
  - To enforce complex business rules not definable using integrity constraints
  - When a required referential integrity rule cannot be enforced using these integrity constraints:
    - `NOT NULL`, `UNIQUE`
    - `PRIMARY KEY`
    - `FOREIGN KEY`
    - `CHECK`
    - `DELETE CASCADE`
    - `DELETE SET NULL`
- Limit the size of triggers.

  If the logic for your trigger requires much more than 60 lines of PL/SQL code, put most of the code in a stored subprogram and invoke the subprogram from the trigger.

  The size of the trigger cannot exceed 32K.
- Use triggers only for centralized, global operations that must fire for the triggering statement, regardless of which user or database application issues the statement.
- Do not create recursive triggers.

  For example, if you create an `AFTER UPDATE` statement trigger on the `employees` table, and the trigger itself issues an `UPDATE` statement on the `employees` table, the trigger fires recursively until it runs out of memory.
- Use triggers on `DATABASE` judiciously. They run for every user every time the event occurs on which the trigger is created.

- If you use a `LOGON` trigger to monitor logons by users, include an exception-handling part in the trigger, and include a `WHEN OTHERS` exception in the exception-handling part. Otherwise, an unhandled exception might block all connections to the database.
- If you use a `LOGON` trigger only to run a package (for example, an application context-setting package), put the exception-handling part in the package instead of in the trigger.

# Trigger Creation

To create a trigger, use the `CREATE TRIGGER` statement. By default, a trigger is created in enabled state. To create a trigger in disabled state, use the `DISABLE` clause of the `CREATE TRIGGER` statement. For information about trigger states, see "Overview of Triggers" on page 9-1.

When using the `CREATE TRIGGER` statement with an interactive tool, such as SQL*Plus or Enterprise Manager, put a single slash (/) on the last line, as in Example 9–1, which creates a simple trigger for the `employees` table.

***Example 9–1 CREATE TRIGGER Statement***

```
CREATE OR REPLACE TRIGGER print_salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON employees
  FOR EACH ROW
  WHEN (NEW.employee_id > 0)
DECLARE
  sal_diff  NUMBER;
BEGIN
  sal_diff  := :NEW.salary  - :OLD.salary;
  dbms_output.put('Old salary: ' || :OLD.salary);
  dbms_output.put('New salary: ' || :NEW.salary);
  dbms_output.put_line('Difference: ' || sal_diff);
END;
/
```

**See Also:** "CREATE TRIGGER Statement" on page 14-53

The trigger in Example 9–1 fires when DML operations are performed on the table. You can choose what combination of operations must fire the trigger.

Because the trigger uses the `BEFORE` keyword, it can access the new values before they go into the table, and can change the values if there is an easily-corrected error by assigning to :`NEW.`*`column_name`*. You might use the `AFTER` keyword if you want the trigger to query or change the same table, because triggers can only do that after the initial changes are applied and the table is back in a consistent state.

Because the trigger uses the `FOR EACH ROW` clause, it might run multiple times, such as when updating or deleting multiple rows. You might omit this clause if you just want to record the fact that the operation occurred, but not examine the data for each row.

After the trigger is created, following SQL statement fires the trigger once for each row that is updated, in each case printing the new salary, the old salary, and the difference between them:

```
UPDATE employees
SET salary = salary + 500.00
WHERE department_ID = 10;
```

The CREATE (or CREATE OR REPLACE) statement fails if any errors exist in the PL/SQL block.

The following topics use Example 9–1 on page 9-5 to show how to specify the parts of a trigger. For additional examples of CREATE TRIGGER statements, see "Trigger Application Examples" on page 9-34.

Topics:

- Trigger Names
- Triggering Statements
- BEFORE and AFTER Triggers
- INSTEAD OF Triggers
- FOR EACH ROW Triggers
- Conditional Triggers
- Compound Triggers
- Order in Which Triggers Fire

## Trigger Names

Triggers in the same schema cannot have the same names. Triggers can have the same names as other schema objects—for example, a table and a trigger can have the same name—however, to avoid confusion, this is not recommended.

## Triggering Statements

A trigger fires based on a **triggering statement**, which specifies:

- The SQL statement, database event, or DDL event that fires the trigger body. The options include DELETE, INSERT, and UPDATE. One, two, or all three of these options can be included in the triggering statement specification.
- The table, view, DATABASE, or SCHEMA on which the trigger is defined.

**Note:** Exactly one table or view can be specified in the triggering statement. If the INSTEAD OF option is used, then the triggering statement must specify a view; conversely, if a view is specified in the triggering statement, then only the INSTEAD OF option can be used.

In Example 9–1 on page 9-5, the PRINT_SALARY_CHANGES trigger fires after any DELETE, INSERT, or UPDATE on the employees table.

Topics:

- SQL*Loader and Import
- How Column Lists Affect UPDATE Triggers

### SQL*Loader and Import

INSERT triggers fire during SQL*Loader conventional loads. (For direct loads, triggers are disabled before the load.)

The IGNORE parameter of the IMP statement determines whether triggers fire during import operations:

- If IGNORE=N (default) and the table exists, then import does not change the table and no existing triggers fire.
- If the table does not exist, then import creates and loads it before any triggers are defined, so again no triggers fire.
- If IGNORE=Y, then import loads rows into existing tables. Any existing triggers fire, and indexes are updated to account for the imported data.

### How Column Lists Affect UPDATE Triggers

An UPDATE statement might include a list of columns. If a triggering statement includes a column list, the trigger fires only when a specified column is updated. If a triggering statement omits a column list, the trigger fires when any column of the associated table is updated. A column list cannot be specified for INSERT or DELETE triggering statements.

The previous example of the PRINT_SALARY_CHANGES trigger can include a column list in the triggering statement. For example:

```
... BEFORE DELETE OR INSERT OR UPDATE OF last_name ON employees ...
```

---

**Note:**

- You cannot specify a column list for UPDATE with INSTEAD OF triggers.
- If the column specified in the UPDATE OF clause is an object column, then the trigger also fires if any of the attributes of the object are modified.
- You cannot specify UPDATE OF clauses on collection columns.

---

## BEFORE and AFTER Triggers

---

**Note:** This topic applies only to simple triggers. For the options of compound triggers, see "Compound Triggers" on page 9-14.

---

The BEFORE or AFTER option in the CREATE TRIGGER statement specifies exactly when to fire the trigger body in relation to the triggering statement that is being run. In a CREATE TRIGGER statement, the BEFORE or AFTER option is specified just before the triggering statement. For example, the PRINT_SALARY_CHANGES trigger in the previous example is a BEFORE trigger.

In general, you use BEFORE or AFTER triggers to achieve these results:

- Use BEFORE row triggers to modify the row before writing the row data to disk.
- Use AFTER row triggers to obtain, and perform operations, using the row ID.

  An AFTER row trigger fires when the triggering statement results in ORA-2292.

> **Note:** AFTER row triggers are slightly more efficient than BEFORE row triggers. With BEFORE row triggers, affected data blocks must be read (logical read, not physical read) once for the trigger and then again for the triggering statement. Alternatively, with AFTER row triggers, the data blocks must be read only once for both the triggering statement and the trigger.

If an UPDATE or DELETE statement detects a conflict with a concurrent UPDATE, then the database performs a transparent ROLLBACK to SAVEPOINT and restarts the update. This can occur many times before the statement completes successfully. Each time the statement is restarted, the BEFORE statement trigger fires again. The rollback to savepoint does not undo changes to any package variables referenced in the trigger. Include a counter variable in your package to detect this situation.

## INSTEAD OF Triggers

An **INSTEAD OF trigger** is a DML trigger defined on a noneditioning view, which Oracle Database fires instead of running the triggering DML statement.

An **updatable** view is one that lets you perform DML on the underlying table. Some views are inherently updatable, but others are not because they were created with one or more of the constructs listed in "Views that Require INSTEAD OF Triggers" on page 9-9.

Any noneditioning view that contains one of those constructs can be made updatable by using an INSTEAD OF trigger. INSTEAD OF triggers provide a transparent way of modifying noneditioning views that cannot be modified directly through UPDATE, INSERT, and DELETE statements. These triggers are called INSTEAD OF triggers because, unlike other types of triggers, Oracle Database fires the trigger instead of running the triggering statement. The trigger must determine what operation was intended and perform UPDATE, INSERT, or DELETE operations directly on the underlying tables.

With an INSTEAD OF trigger, you can write normal UPDATE, INSERT, and DELETE statements against the noneditioning view, and the INSTEAD OF trigger performs the correct actions.

INSTEAD OF triggers can only be activated for each row.

**See Also:** "FOR EACH ROW Triggers" on page 9-13

> **Note:**
>
> - The INSTEAD OF option can be used only for a trigger defined on a noneditioning view (not an editioning view or table).
> - The BEFORE and AFTER options cannot be used for triggers defined on noneditioning views.
> - The CHECK option for views is not enforced when inserts or updates to the view are done using INSTEAD OF triggers. The INSTEAD OF trigger body must enforce the check.

Topics:

- Views that Require INSTEAD OF Triggers
- Triggers on Nested Table View Columns
- INSTEAD OF Trigger Example


### Views that Require INSTEAD OF Triggers

A noneditioning view cannot be modified by `UPDATE`, `INSERT`, or `DELETE` statements if the view query contains any of these constructs:

- A set operator
- A `DISTINCT` operator
- An aggregate or analytic function
- A `GROUP BY`, `ORDER BY`, `MODEL`, `CONNECT BY`, or `START WITH` clause
- A collection expression in a `SELECT` list
- A subquery in a `SELECT` list
- A subquery designated `WITH READ ONLY`
- Joins, with some exceptions, as documented in *Oracle Database Administrator's Guide*

If a view contains pseudocolumns or expressions, then you can only update the view with an `UPDATE` statement that does not refer to any of the pseudocolumns or expressions.

`INSTEAD OF` triggers provide the means to modify object view instances on the client-side through OCI calls.

> **See Also:** *Oracle Call Interface Programmer's Guide*

To modify an object materialized by an object view in the client-side object cache and flush it back to the persistent store, you must specify `INSTEAD OF` triggers, unless the object view is modifiable. If the object is read only, then it is not necessary to define triggers to pin it.

### Triggers on Nested Table View Columns

`INSTEAD OF` triggers can also be created over nested table view columns. These triggers provide a way of updating elements of the nested table. They fire for each nested table element being modified. The row correlation variables inside the trigger correspond to the nested table element. This type of trigger also provides an additional correlation name for accessing the parent row that contains the nested table being modified.

> **Note:** These triggers:
>
> - Can be defined only over nested table columns in views.
> - Fire only when the nested table elements are modified using the `TABLE` clause, not when a DML statement is performed on the view.

For example, consider a department view, `dept_view`, which contains a nested table of employees:

```
-- Create type of nested table element:

CREATE OR REPLACE TYPE nte
AUTHID DEFINER IS
OBJECT (
  emp_id     NUMBER(6),
  lastname   VARCHAR2(25),
  job        VARCHAR2(10),
  sal        NUMBER(8,2)
);
/

-- Created type of nested table:

CREATE OR REPLACE TYPE emp_list_ IS
  TABLE OF nte;
/

-- Create view:

CREATE OR REPLACE VIEW dept_view AS
  SELECT d.department_id,
         d.department_name,
         CAST (MULTISET (SELECT e.employee_id, e.last_name, e.job_id, e.salary
                         FROM employees e
                         WHERE e.department_id = d.department_id
                        )
                        AS emp_list_
              ) emplist
  FROM departments d;
```

The `CAST` (`MULTISET`) operator creates a multiset of employees for each department. (For information about the `CAST` function, see *Oracle Database SQL Language Reference*.)

To modify the `emplist` column, which is the nested table of employees, define an `INSTEAD OF` trigger on the column to handle the operation. For example:

```
CREATE OR REPLACE TRIGGER dept_emplist_tr
  INSTEAD OF INSERT ON NESTED TABLE emplist OF dept_view
  REFERENCING NEW AS Employee
  PARENT AS Department
  FOR EACH ROW
BEGIN
  -- Insert on nested table translates to insert on base table:
  INSERT INTO employees (
    employee_id,
    last_name,
    email,
    hire_date,
    job_id,
    salary,
    department_id
  )
  VALUES (
    :Employee.emp_id,                      -- employee_id
    :Employee.lastname,                    -- last_name
    :Employee.lastname || '@company.com',  -- email
    SYSDATE,                               -- hire_date
    :Employee.job,                         -- job_id
    :Employee.sal,                         -- salary
    :Department.department_id              -- department_id
```

```
  );
END;
/
```

Any `INSERT` into the nested table fires the trigger, and the `employees` table is filled with the correct values. For example, query `employees`:

```
SELECT employee_id, last_name, job_id, salary
FROM employees
WHERE department_id=10;
```

Result:

```
EMPLOYEE_ID LAST_NAME                 JOB_ID         SALARY
----------- ------------------------- ---------- ----------
        200 Whalen                    AD_ASST          4288

1 row selected.
```

Insert a row into the nested table:

```
INSERT INTO TABLE (
  SELECT d.emplist
  FROM dept_view d
  WHERE department_id = 10
)
VALUES (1001, 'Glenn', 'AC_MGR', 10000);
```

Query `employees`:

```
EMPLOYEE_ID LAST_NAME                 JOB_ID         SALARY
----------- ------------------------- ---------- ----------
        200 Whalen                    AD_ASST          4288
       1001 Glenn                     AC_MGR          10000

2 rows selected.
```

### INSTEAD OF Trigger Example

---

**Note:** Example 9–2 needs these data structures:

```
DROP TABLE Project_tab;
CREATE TABLE Project_tab (
  Prj_level NUMBER,
  Projno    NUMBER,
  Resp_dept NUMBER);

DROP TABLE emp;
CREATE TABLE emp (
  Empno     NUMBER NOT NULL,
  Ename     VARCHAR2(10),
  Job       VARCHAR2(9),
  Mgr       NUMBER(4),
  Hiredate  DATE,
  Sal       NUMBER(7,2),
  Comm      NUMBER(7,2),
  Deptno    NUMBER(2) NOT NULL);

DROP TABLE dept;
CREATE TABLE dept (
  Deptno    NUMBER(2) NOT NULL,
  Dname     VARCHAR2(14),
  Loc       VARCHAR2(13),
  Mgr_no    NUMBER,
  Dept_type NUMBER);
```

---

Example 9–2 creates a view and an `INSTEAD OF` trigger for inserting rows into the view.

***Example 9–2 INSTEAD OF Trigger***

```
CREATE OR REPLACE VIEW manager_info AS
  SELECT e.ename, e.empno, d.dept_type, d.deptno, p.prj_level, p.projno
  FROM emp e, dept d, Project_tab p
  WHERE e.empno =  d.mgr_no
  AND d.deptno = p.resp_dept;

CREATE OR REPLACE TRIGGER manager_info_insert
  INSTEAD OF INSERT ON manager_info
    REFERENCING NEW AS n  -- new manager information
      FOR EACH ROW
DECLARE
  rowcnt number;
BEGIN
  SELECT COUNT(*) INTO rowcnt FROM emp WHERE empno = :n.empno;
  IF rowcnt = 0  THEN
    INSERT INTO emp (empno,ename) VALUES (:n.empno, :n.ename);
  ELSE
    UPDATE emp SET emp.ename = :n.ename WHERE emp.empno = :n.empno;
  END IF;
  SELECT COUNT(*) INTO rowcnt FROM dept WHERE deptno = :n.deptno;
  IF rowcnt = 0 THEN
    INSERT INTO dept (deptno, dept_type)
      VALUES(:n.deptno, :n.dept_type);
  ELSE
```

```
    UPDATE dept SET dept.dept_type = :n.dept_type
      WHERE dept.deptno = :n.deptno;
  END IF;
  SELECT COUNT(*) INTO rowcnt FROM Project_tab
    WHERE Project_tab.projno = :n.projno;
  IF rowcnt = 0 THEN
    INSERT INTO Project_tab (projno, prj_level)
      VALUES(:n.projno, :n.prj_level);
  ELSE
    UPDATE Project_tab SET Project_tab.prj_level = :n.prj_level
      WHERE Project_tab.projno = :n.projno;
  END IF;
END;
/
```

The actions shown for rows being inserted into the MANAGER_INFO view first test to see if appropriate rows exist in the base tables from which MANAGER_INFO is derived. The actions then insert or update rows, as appropriate. Similar triggers can specify appropriate actions for UPDATE and DELETE.

## FOR EACH ROW Triggers

---

**Note:** This topic applies only to simple triggers. For the options of compound triggers, see "Compound Triggers" on page 9-14.

---

If you specify FOR EACH ROW, the trigger is a **row-level trigger**; it fires once for each row of the table that the triggering statement affects. Without FOR EACH ROW, the trigger is a **statement-level trigger**; it fires only once for each triggering statement, not separately for each affected row. Statement-level triggers are useful for performing validation checks for the entire statement.

Example 9–3 creates a FOR EACH ROW trigger that inserts a row in a log table each time an UPDATE statement affects a row of the EMPLOYEES table.

***Example 9–3 FOR EACH ROW Trigger***

Create log table:

```
DROP TABLE Emp_log;
CREATE TABLE Emp_log (
  Emp_id     NUMBER,
  Log_date   DATE,
  New_salary NUMBER,
  Action     VARCHAR2(20));
```

Create trigger on EMPLOYEES table:

```
CREATE OR REPLACE TRIGGER log_salary_increase
  AFTER UPDATE ON employees
  FOR EACH ROW
  WHEN (NEW.salary > 1000)
BEGIN
  INSERT INTO Emp_log (Emp_id, Log_date, New_salary, Action)
  VALUES (:NEW.employee_id, SYSDATE, :NEW.salary, 'NEW SAL');
END;
/
```

Update EMPLOYEES table:

```
UPDATE employees SET salary = salary + 1000.0
WHERE Department_id = 20;
```

Result:

```
2 rows updated.
```

Show log table:

```
SELECT * FROM Emp_log;
```

Result:

```
    EMP_ID LOG_DATE  NEW_SALARY ACTION
---------- --------- ---------- --------------------
       201 10-MAR-09      14260 NEW SAL
       202 10-MAR-09       7120 NEW SAL

2 rows selected.
```

## Conditional Triggers

Optionally, a trigger restriction can be included in the definition of a row trigger by specifying a Boolean SQL expression in a WHEN clause.

> **Note:** A WHEN clause cannot be included in the definition of a statement trigger.

If included, 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 runs 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 run for that row. The evaluation of the WHEN clause does not affect 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).

For example, in the PRINT_SALARY_CHANGES trigger, the trigger body is not run if the new value of employee_id is zero, NULL, or negative. In more realistic examples, you might test if one column value is less than another.

The expression in a WHEN clause of a row trigger can include correlation names, which are explained in "Trigger Body" on page 9-21. The expression in a WHEN clause must be a SQL expression, and it cannot include a subquery. You cannot use a PL/SQL expression (including user-defined functions) in the WHEN clause.

> **Note:** You cannot specify the WHEN clause for INSTEAD OF triggers.

## Compound Triggers

A compound trigger can fire at multiple timing points.

Topics:

- Compound Trigger Uses
- Compound Trigger Sections

- Triggering Statements of Compound Triggers
- Compound Trigger Restrictions
- Compound Trigger Example
- Using Compound Triggers to Avoid Mutating-Table Error


### Compound Trigger Uses

The compound trigger makes it easier to program an approach where you want the actions you implement for the various timing points to share common data. To achieve the same effect with simple triggers, you had to model the common state with an ancillary package. This approach was both cumbersome to program and subject to memory leak when the triggering statement caused an error and the after-statement trigger did not fire.

A compound trigger has an optional declarative part and a section for each of its timing points (see Example 9–4). All of these sections can access a common PL/SQL state. The common state is established when the triggering statement starts and is destroyed when the triggering statement completes, even when the triggering statement causes an error.

***Example 9–4 Compound Trigger***

```
CREATE OR REPLACE TRIGGER compound_trigger
  FOR UPDATE OF salary ON employees
  COMPOUND TRIGGER

  -- Declarative part (optional)
  -- Variables declared here have firing-statement duration.
  threshold CONSTANT SIMPLE_INTEGER := 200;

  BEFORE STATEMENT IS
  BEGIN
    NULL;
  END BEFORE STATEMENT;

  BEFORE EACH ROW IS
  BEGIN
    NULL;
  END BEFORE EACH ROW;

  AFTER EACH ROW IS
  BEGIN
    NULL;
  END AFTER EACH ROW;

  AFTER STATEMENT IS
  BEGIN
    NULL;
  END AFTER STATEMENT;
END compound_trigger;
/
```

Two common reasons to use compound triggers are:

- To accumulate rows destined for a second table so that you can periodically bulk-insert them (as in "Compound Trigger Example" on page 9-17)
- To avoid the mutating-table error (ORA-04091) (as in "Using Compound Triggers to Avoid Mutating-Table Error" on page 9-19)

### Compound Trigger Sections

A compound trigger has an optional declarative part and at least one timing-point section. If a compound trigger has multiple timing-point sections, each section must be for a different timing point.

The optional **declarative part** declares variables and subprograms that all of the timing-point sections can use. When the trigger fires, the declarative part runs before any timing-point sections run. Variables and subprograms declared in this section have firing-statement duration.

A compound trigger defined on a noneditioning view has an `INSTEAD OF EACH ROW` timing-point section, and no other timing-point section.

A compound trigger defined on a table or editioning view has one or more of the timing-point sections described in Table 9–1. Timing-point sections must appear in the order shown in Table 9–1. If a timing-point section is absent, nothing happens at its timing point.

A timing-point section cannot be enclosed in a PL/SQL block.

Table 9–1 summarizes the timing point sections of a compound trigger that can be defined on a table or editioning view.

*Table 9–1 Timing-Point Sections of a Compound Trigger Defined*

| Timing Point | Section |
|---|---|
| Before the triggering statement runs | `BEFORE STATEMENT` |
| After the triggering statement runs | `AFTER STATEMENT` |
| Before each row that the triggering statement affects | `BEFORE EACH ROW` |
| After each row that the triggering statement affects | `AFTER EACH ROW` |

Any section can include the functions `Inserting`, `Updating`, `Deleting`, and `Applying`.

> **See Also:** "CREATE TRIGGER Statement" on page 14-53 for more information about the syntax of compound triggers

### Triggering Statements of Compound Triggers

The triggering statement of a compound trigger must be a DML statement.

If the triggering statement affects no rows, and the compound trigger has neither a `BEFORE STATEMENT` section nor an `AFTER STATEMENT` section, the trigger never fires.

It is when the triggering statement affects many rows that a compound trigger has a performance benefit. This is why it is important to use the `BULK COLLECT` clause with the `FORALL` statement. For example, without the `BULK COLLECT` clause, a `FORALL` statement that contains an `INSERT` statement simply performs a single-row insertion operation many times, and you get no benefit from using a compound trigger. For more information about using the `BULK COLLECT` clause with the `FORALL` statement, see "Using FORALL and BULK COLLECT Together" on page 12-27.

If the triggering statement of a compound trigger is an `INSERT` statement that includes a subquery, the compound trigger retains some of its performance benefit. For example, suppose that a compound trigger is triggered by this statement:

```
INSERT INTO Target
  SELECT c1, c2, c3
  FROM Source
```

```
WHERE Source.c1 > 0
```

For each row of Source whose column c1 is greater than zero, the BEFORE EACH ROW and AFTER EACH ROW sections of the compound trigger run. However, the BEFORE STATEMENT and AFTER STATEMENT sections each run only once (before and after the INSERT statement runs, respectively).

### Compound Trigger Restrictions

- The body of a compound trigger must be a compound trigger block.
- A compound trigger must be a DML trigger.
- A compound trigger must be defined on either a table or a view.
- The declarative part cannot include the AUTONOMOUS_TRANSACTION pragma.
- 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 runs exactly once before any other timing-point section runs.
- An exception that occurs in one section must be handled in that section. It cannot transfer control to another section.
- If a section includes a GOTO statement, the target of the GOTO statement must be in the same section.
- :OLD, :NEW, and :PARENT cannot appear in the declarative part, the BEFORE STATEMENT section, or the AFTER STATEMENT section.
- Only the BEFORE EACH ROW section can change the value of :NEW.
- If, after the compound trigger fires, a DML exception causes the triggering statement to roll back:
  - Local variables declared in the compound trigger sections are re-initialized, and any values computed thus far are lost.
  - Side effects from firing the compound trigger are not rolled back.
- The firing order of compound triggers is not guaranteed. Their firing can be interleaved with the firing of simple triggers. For information about firing order, see "Order in Which Triggers Fire" on page 9-20.

### Compound Trigger Example

**Scenario:** You want to record every change to hr.employees.salary in a new table, employee_salaries. A single UPDATE statement will update many rows of the table hr.employees; therefore, bulk-inserting rows into employee.salaries is more efficient than inserting them individually.

**Solution:** Define a compound trigger on updates of the table hr.employees, as in Example 9–5. You do not need a BEFORE STATEMENT section to initialize idx or salaries, because they are state variables, which are initialized each time the trigger fires (even when the triggering statement is interrupted and restarted).

---

**Note:** To run Example 9–5, you must have the EXECUTE privilege on the package DBMS_LOCK.

---

***Example 9–5 Compound Trigger Records Changes to One Table in Another Table***

```
CREATE TABLE employee_salaries (
  employee_id NUMBER NOT NULL,
  change_date DATE   NOT NULL,
  salary NUMBER(8,2) NOT NULL,
  CONSTRAINT pk_employee_salaries PRIMARY KEY (employee_id, change_date),
  CONSTRAINT fk_employee_salaries FOREIGN KEY (employee_id)
    REFERENCES employees (employee_id)
      ON DELETE CASCADE)
/
CREATE OR REPLACE TRIGGER maintain_employee_salaries
  FOR UPDATE OF salary ON employees
    COMPOUND TRIGGER

-- Declarative Part:
-- Choose small threshhold value to show how example works:
  threshhold CONSTANT SIMPLE_INTEGER := 7;

  TYPE salaries_t IS TABLE OF employee_salaries%ROWTYPE INDEX BY SIMPLE_INTEGER;
  salaries  salaries_t;
  idx       SIMPLE_INTEGER := 0;

  PROCEDURE flush_array IS
    n CONSTANT SIMPLE_INTEGER := salaries.count();
  BEGIN
    FORALL j IN 1..n
      INSERT INTO employee_salaries VALUES salaries(j);
    salaries.delete();
    idx := 0;
    DBMS_OUTPUT.PUT_LINE('Flushed ' || n || ' rows');
  END flush_array;

  -- AFTER EACH ROW Section:

  AFTER EACH ROW IS
  BEGIN
    idx := idx + 1;
    salaries(idx).employee_id := :NEW.employee_id;
    salaries(idx).change_date := SYSDATE();
    salaries(idx).salary := :NEW.salary;
    IF idx >= threshhold THEN
      flush_array();
    END IF;
  END AFTER EACH ROW;

  -- AFTER STATEMENT Section:

  AFTER STATEMENT IS
  BEGIN
    flush_array();
  END AFTER STATEMENT;
END maintain_employee_salaries;
/
/* Increase salary of every employee in department 50 by 10%: */

UPDATE employees
  SET salary = salary * 1.1
  WHERE department_id = 50
/
```

```
/* Wait two seconds: */

BEGIN
  DBMS_LOCK.SLEEP(2);
END;
/

/* Increase salary of every employee in department 50 by 5%: */

UPDATE employees
  SET salary = salary * 1.05
  WHERE department_id = 50
/
```

### Using Compound Triggers to Avoid Mutating-Table Error

You can use compound triggers to avoid the mutating-table error (ORA-04091) described in "Mutating Table Restriction" on page 9-28.

**Scenario:** A business rule states that an employee's salary increase must not exceed 10% of the average salary for the employee's department. This rule must be enforced by a trigger.

**Solution:** Define a compound trigger on updates of the table `hr.employees`, as in Example 9–6. The state variables are initialized each time the trigger fires (even when the triggering statement is interrupted and restarted).

***Example 9–6 Compound Trigger for Avoiding Mutating-Table Error***

```
CREATE OR REPLACE TRIGGER Check_Employee_Salary_Raise
  FOR UPDATE OF Salary ON Employees
COMPOUND TRIGGER
  Ten_Percent                 CONSTANT NUMBER := 0.1;
  TYPE Salaries_t             IS TABLE OF Employees.Salary%TYPE;
  Avg_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_Avg_Salaries     Department_Salaries_t;

  BEFORE STATEMENT IS
  BEGIN
    SELECT               AVG(e.Salary), NVL(e.Department_ID, -1)
      BULK COLLECT INTO  Avg_Salaries, Department_IDs
      FROM               Employees e
      GROUP BY           e.Department_ID;
    FOR j IN 1..Department_IDs.COUNT() LOOP
      Department_Avg_Salaries(Department_IDs(j)) := Avg_Salaries(j);
    END LOOP;
  END BEFORE STATEMENT;

  AFTER EACH ROW IS
  BEGIN
    IF :NEW.Salary - :Old.Salary >
      Ten_Percent*Department_Avg_Salaries(:NEW.Department_ID)
    THEN
      Raise_Application_Error(-20000, 'Raise too big');
    END IF;
  END AFTER EACH ROW;
```

```
END Check_Employee_Salary_Raise;
```

## Order in Which Triggers Fire

If two or more triggers *with different timing points* are defined for the same statement on the same table, then they fire in this order:

1. All BEFORE STATEMENT triggers
2. All BEFORE EACH ROW triggers
3. All AFTER EACH ROW triggers
4. All AFTER STATEMENT triggers

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 (see "FOLLOWS | PRECEDES" on page 14-63).

If multiple compound triggers are specified on a table, then all BEFORE STATEMENT sections run at the BEFORE STATEMENT timing point, BEFORE EACH ROW sections run at the BEFORE EACH ROW timing point, and so forth. If trigger execution order was specified using the FOLLOWS clause, then order of execution of compound trigger sections is determined by the FOLLOWS clause. If FOLLOWS is specified only for some triggers but not all triggers, then the order of execution of triggers is guaranteed only for those that are related using the FOLLOWS clause.

If multiple compound triggers are specified on a table, then all AFTER STATEMENT sections run at the AFTER STATEMENT timing point, AFTER EACH ROW sections run at the AFTER EACH ROW timing point, and so forth. If trigger execution order was specified using the PRECEDES clause, then order of execution of compound trigger sections is determined by the PRECEDES clause. If PRECEDES is specified only for some triggers but not all triggers, then the order of execution of triggers is guaranteed only for those that are related using the PRECEDES clause.

**Note:** PRECEDES applies only to reverse crossedition triggers.

A relational database does not guarantee the order of rows processed by a SQL statement. Therefore, do not create triggers that depend on the order in which rows are processed. For example, do not assign a value to a global package variable in a row trigger if the current value of the global variable depends on the row being processed by the row trigger. Also, if global package variables are updated in a trigger, initialize those variables in a BEFORE statement trigger.

When a statement in a trigger body causes another trigger to fire, the triggers are said to be **cascading**. The database allows up to 32 triggers to cascade at simultaneously. You can limit the number of trigger cascades by using the initialization parameter OPEN_CURSORS, because a cursor must be opened for every execution of a trigger.

Although any trigger can run a sequence of operations either inline or by invoking subprograms, using multiple triggers of the same type allows the modular installation of applications that have triggers on the same tables.

Each subsequent trigger sees the changes made by the previously fired triggers. Each trigger can see the old and new values. The old values are the original values, and the new values are the current values, as set by the most recently fired UPDATE or INSERT trigger.

# Trigger Body

> **Note:** This topic applies primarily to simple triggers. The body of a compound trigger has a different format (see "Compound Triggers" on page 9-14).

The trigger body is either a CALL subprogram (a PL/SQL subprogram, or a Java subprogram encapsulated in a PL/SQL wrapper) or a PL/SQL block, and as such, it can include SQL and PL/SQL statements. These statements run if the triggering statement is entered and if the trigger restriction (if any) evaluates to TRUE.

If the trigger body for a row trigger is a PL/SQL block (not a CALL subprogram), it can include these constructs:

- REFERENCING clause, which can specify the correlation names OLD, NEW, and PARENT
- Conditional predicates (see "Conditional Predicates for Detecting the DML Operation that Fired a Trigger" on page 9-24)

> **Note:** In a trigger body, this statement returns the owner of the trigger, not the name of user who is updating the table:
>
> ```
> SELECT Username FROM USER_USERS;
> ```

The LOGON trigger in Example 9–7 runs the procedure check_user after a user logs onto the database. The body of the trigger includes an exception-handling part, which includes a WHEN OTHERS exception that invokes RAISE_APPLICATION_ERROR.

***Example 9–7 Trigger for Monitoring Logons***

```
CREATE OR REPLACE TRIGGER check_user
  AFTER LOGON ON DATABASE
  BEGIN
    check_user;
  EXCEPTION
    WHEN OTHERS THEN
      RAISE_APPLICATION_ERROR
        (-20000, 'Unexpected error: '|| DBMS_Utility.Format_Error_Stack);
 END;
/
```

Although triggers are declared using PL/SQL, they can call subprograms in other languages. The trigger in Example 9–8 invokes a Java subprogram.

***Example 9–8 Trigger That Invokes Java Subprogram***

```
CREATE OR REPLACE PROCEDURE Before_delete (Id IN NUMBER, Ename VARCHAR2)
IS language Java
name 'thjvTriggers.beforeDelete (oracle.sql.NUMBER, oracle.sql.CHAR)';

CREATE OR REPLACE TRIGGER Pre_del_trigger BEFORE DELETE ON Tab
FOR EACH ROW
CALL Before_delete (:OLD.Id, :OLD.Ename)
/
```

The corresponding Java file is thjvTriggers.java:

```
import java.sql.*
import java.io.*
import oracle.sql.*
import oracle.oracore.*
public class thjvTriggers
{
public state void
beforeDelete (NUMBER old_id, CHAR old_name)
Throws SQLException, CoreException
   {
   Connection conn = JDBCConnection.defaultConnection();
   Statement stmt = conn.CreateStatement();
   String sql = "insert into logtab values
   ("+ old_id.intValue() +", '"+ old_ename.toString() + ", BEFORE DELETE');
   stmt.executeUpdate (sql);
   stmt.close();
   return;
   }
}
```

Topics:

- Column Values Accessible to Row Triggers
- OBJECT_VALUE Pseudocolumn
- Remote Exception Handling
- Trigger Body Restrictions

## Column Values Accessible to Row Triggers

In a trigger body of a row trigger, the PL/SQL code and SQL statements have access to the old and new column values of the current row affected by the triggering statement. Two correlation names exist for every column of the table being modified: one for the old column value, and one for the new column value. Depending on the type of triggering statement, certain correlation names might not have any meaning.

- A trigger fired by an `INSERT` statement has meaningful access to new column values only. Because the row is being created by the `INSERT`, the old values are null.
- A trigger fired by an `UPDATE` statement has access to both old and new column values for both `BEFORE` and `AFTER` row triggers.
- A trigger fired by a `DELETE` statement has meaningful access to :`OLD` column values only. Because the row no longer exists after the row is deleted, the :`NEW` values are `NULL`. However, you cannot modify :`NEW` values because `ORA-4084` is raised if you try to modify :`NEW` values.

The new column values are referenced using the `NEW` qualifier before the column name, while the old column values are referenced using the `OLD` qualifier before the column name. For example, if the triggering statement is associated with the `emp` table (with the columns `SAL`, `COMM`, and so on), then you can include statements in the trigger body. For example:

```
IF :NEW.Sal > 10000 ...
IF :NEW.Sal < :OLD.Sal ...
```

Old and new values are available in both `BEFORE` and `AFTER` row triggers. A `NEW` column value can be assigned in a `BEFORE` row trigger, but not in an `AFTER` row trigger (because the triggering statement takes effect before an `AFTER` row trigger fires). If a `BEFORE` row trigger changes the value of `NEW.column`, then an `AFTER` row

trigger fired by the same statement sees the change assigned by the BEFORE row trigger.

Correlation names can also be used in the Boolean expression of a WHEN clause. A colon (:) must precede the OLD and NEW qualifiers when they are used in a trigger body, but a colon is not allowed when using the qualifiers in the WHEN clause or the REFERENCING option.

Topics:

- Example: Trigger That Modifies LOB Columns
- INSTEAD OF Triggers on Nested Table View Columns
- Trigger Name Conflicts
- Conditional Predicates for Detecting the DML Operation that Fired a Trigger
- Exceptions Raised in the Trigger Body

### Example: Trigger That Modifies LOB Columns

You can treat LOB columns the same as other columns, using regular SQL and PL/SQL functions with CLOB columns, and calls to the DBMS_LOB package with BLOB columns, as in Example 9–9.

***Example 9–9 Trigger that Modifies LOB Columns***

```
DROP TABLE tab1;
CREATE TABLE tab1 (c1 CLOB);
INSERT INTO tab1 VALUES ('<h1>HTML Document Fragment</h1><p>Some text.');

CREATE OR REPLACE TRIGGER trg1
  BEFORE UPDATE ON tab1
  FOR EACH ROW
BEGIN
  DBMS_OUTPUT.PUT_LINE('Old value of CLOB column: '||:OLD.c1);
  DBMS_OUTPUT.PUT_LINE('Proposed new value of CLOB column: '||:NEW.c1);

  :NEW.c1 := :NEW.c1 || TO_CLOB('<hr><p>Standard footer paragraph.');

  DBMS_OUTPUT.PUT_LINE('Final value of CLOB column: '||:NEW.c1);
END;
/

SET SERVEROUTPUT ON;
UPDATE tab1 SET c1 = '<h1>Different Document Fragment</h1><p>Different text.';

SELECT * FROM tab1;
```

### INSTEAD OF Triggers on Nested Table View Columns

In the case of INSTEAD OF triggers on nested table view columns, the NEW and OLD qualifiers correspond to the new and old nested table elements. The parent row corresponding to this nested table element can be accessed using the parent qualifier. The parent correlation name is meaningful and valid only inside a nested table trigger.

### Trigger Name Conflicts

The REFERENCING option can be specified in a trigger body of a row trigger to avoid name conflicts among the correlation names and tables that might be named OLD or NEW. Because this is rare, this option is infrequently used.

Example 9–10 creates a table with the same name as a correlation name, `new`, and then creates a trigger on that table. To avoid conflict between the table name and the correlation name, the trigger references the correlation name as `Newest`.

***Example 9–10 REFERENCING Option***

```
CREATE TABLE new (
  field1  NUMBER,
  field2  VARCHAR2(20)
);

CREATE OR REPLACE TRIGGER Print_salary_changes
BEFORE UPDATE ON new
REFERENCING new AS Newest
FOR EACH ROW
BEGIN
  :Newest.Field2 := TO_CHAR (:newest.field1);
END;
/
```

## Conditional Predicates for Detecting the DML Operation that Fired a Trigger

If multiple types of DML statements can fire a trigger, the trigger body can use these **conditional predicates** to determine which type of statement fired the trigger:

- `INSERTING`, whose value is `TRUE` if and only if an `INSERT` statement fired the trigger.
- `UPDATING`, whose value is `TRUE` if and only if an `UPDATE` statement fired the trigger.
- `UPDATING ('`*`column_name`*`')`, whose value is `TRUE` if and only if an `UPDATE` statement statement that affected the specified column fired the trigger.
- `DELETING`, whose value is `TRUE` if and only if a `DELETE` statement fired the trigger.

In the code of the trigger body, you can use a conditional predicate wherever you can use a `BOOLEAN` expression. For example:

```
IF INSERTING OR UPDATING THEN ... ELSE ... END IF;

CASE
    WHEN INSERTING THEN ... ;
    WHEN UPDATING('salary') THEN ... ;
    WHEN UPDATING('department_id') THEN ... ;
    WHEN DELETING THEN ... ;
  END CASE;
```

## Exceptions Raised in the Trigger Body

If a predefined or user-defined error condition (exception) is raised during the execution of a trigger body, then all effects of both the trigger body and the triggering statement are rolled back (unless the error is trapped by an exception handler). Therefore, a trigger body can prevent the execution of the triggering statement by raising an exception. User-defined exceptions are commonly used in triggers that enforce complex security authorizations or constraints.

If the `LOGON` trigger raises an exception, logon fails except in these cases:

- Database startup and shutdown operations do not fail even if the system triggers for these events raise exceptions. Only the trigger action is rolled back. The error is logged in trace files and the alert log.
- If the system trigger is a DATABASE LOGON trigger and the user has ADMINISTER DATABASE TRIGGER privilege, then the user can log on successfully even if the trigger raises an exception. For SCHEMA LOGON triggers, if the user logging on is the trigger owner or has ALTER ANY TRIGGER privileges then logon is permitted. Only the trigger action is rolled back and an error is logged in the trace files and alert log.

## OBJECT_VALUE Pseudocolumn

You can use the OBJECT_VALUE pseudocolumn in a trigger on an object table because, as of 10*g* Release 1 (10.1), OBJECT_VALUE means the object as a whole. This is one example of its use. You can also invoke a PL/SQL function with OBJECT_VALUE as the data type of an IN formal parameter.

Example 9–11 creates an object table, tbl, and a history table, tbl_history, to keep track of updates to values in tbl. For tbl, the values 1 through 5 are inserted into n, while m is kept at 0. A row-level trigger runs once for each row that is affected by a DML statement. The trigger causes the old and new values of the object t in tbl to be written in tbl_history when tbl is updated. These old and new values are :OLD.OBJECT_VALUE and :NEW.OBJECT_VALUE. An update of the table tbl is done (each value of n is increased by 1).

***Example 9–11 Trigger with OBJECT_VALUE Pseudocolumn***

Create, populate, and show object table:

```
CREATE OR REPLACE TYPE t AS OBJECT (n NUMBER, m NUMBER)
/
CREATE TABLE tbl OF t
/
BEGIN
  FOR j IN 1..5 LOOP
    INSERT INTO tbl VALUES (t(j, 0));
  END LOOP;
END;
/
SELECT * FROM tbl;
```

Result:

```
         N          M
---------- ----------
         1          0
         2          0
         3          0
         4          0
         5          0

5 rows selected.
```

Create history table and trigger:

```
CREATE TABLE tbl_history ( d DATE, old_obj t, new_obj t)
/
CREATE OR REPLACE TRIGGER Tbl_Trg
  AFTER UPDATE ON tbl
```

```
  FOR EACH ROW
BEGIN
  INSERT INTO tbl_history (d, old_obj, new_obj)
  VALUES (SYSDATE, :OLD.OBJECT_VALUE, :NEW.OBJECT_VALUE);
END Tbl_Trg;
/
```

Update object table:

```
UPDATE tbl SET tbl.n = tbl.n+1
/
```

Result:

```
5 rows updated.
```

Show old and new values:

```
BEGIN
  FOR j IN (SELECT d, old_obj, new_obj FROM tbl_history) LOOP
    DBMS_OUTPUT.PUT_LINE (
      j.d ||
      ' -- old: ' || j.old_obj.n || ' ' || j.old_obj.m ||
      ' -- new: ' || j.new_obj.n || ' ' || j.new_obj.m
    );
  END LOOP;
END;
/
```

Result:

```
10-MAR-09 -- old: 1 0 -- new: 2 0
10-MAR-09 -- old: 2 0 -- new: 3 0
10-MAR-09 -- old: 3 0 -- new: 4 0
10-MAR-09 -- old: 4 0 -- new: 5 0
10-MAR-09 -- old: 5 0 -- new: 6 0
```

All values of column `n` were increased by 1. The value of `m` remains 0.

## Remote Exception Handling

Statements in a trigger body can reference remote schema objects. However, pay special attention when invoking remote subprograms from in a local trigger. If a timestamp or signature mismatch is found during execution of the trigger, then the remote subprogram is not run, and the trigger is invalidated.

A trigger that accesses a remote site cannot do remote exception handling if the network link is unavailable. A trigger is compiled when it is created. Thus, if a remote site is unavailable when the trigger must compile, then the database cannot validate the statement accessing the remote database, and the compilation fails.

The trigger in Example 9–12 cannot be compiled; therefore, its exception handler cannot run.

***Example 9–12 Remote Exception Handling Failure***

```
CREATE OR REPLACE TRIGGER employees_tr
  AFTER INSERT ON employees
  FOR EACH ROW
BEGIN
  -- When dblink is inaccessible, compilation fails here:
  INSERT INTO employees@remote (
```

```
    employee_id, first_name, last_name, email, hire_date, job_id
  )
  VALUES (
    99, 'Jane', 'Doe', 'jane.doe@acme.com', SYSDATE, 'ST_MAN'
  );
EXCEPTION
  WHEN OTHERS THEN
    INSERT INTO emp_log (Emp_id, Log_date, New_salary, Action)
    VALUES (99, SYSDATE, NULL, 'Could not insert');
END;
/
```

The work-around for the problem in Example 9–12 is to put the exception handler in a stored subprogram and have the trigger invoke the stored subprogram, as in Example 9–13. The stored subprogram is stored in compiled form, with a validated statement for accessing the remote database. Therefore, when the remote INSERT statement fails because the link is down, the exception is caught.

***Example 9–13 Workaround for Example 9–12***

```
CREATE OR REPLACE PROCEDURE insert_row_proc AS
BEGIN
  INSERT INTO employees@remote (
    employee_id, first_name, last_name, email, hire_date, job_id
  )
  VALUES (
    99, 'Jane', 'Doe', 'jane.doe@acme.com', SYSDATE, 'ST_MAN'
  );
EXCEPTION
  WHEN OTHERS THEN
    INSERT INTO emp_log (Emp_id, Log_date, New_salary, Action)
    VALUES (99, SYSDATE, NULL, 'Could not insert');
END;
/

CREATE OR REPLACE TRIGGER employees_tr
  AFTER INSERT ON employees
  FOR EACH ROW
BEGIN
  insert_row_proc;
END;
/
```

## Trigger Body Restrictions

In addition to the restrictions that apply to all PL/SQL units, trigger bodies have these restrictions:

- Size Restriction
- SQL Statement Restrictions
- LONG and LONG RAW Data Type Restrictions
- Mutating Table Restriction
- System Trigger Restrictions

### Size Restriction

The size of a trigger cannot exceed 32 K.

### SQL Statement Restrictions

A trigger cannot contain transaction control statements. Furthermore, a subprogram invoked by a trigger cannot run transaction control statements, because the subprogram runs in the context of the trigger body.

Only a system trigger can contain these SQL statements:

- CREATETABLE
- ALTERTABLE
- DROP TABLE
- ALTER ... COMPILE

A trigger can contain the SELECT statement only in a cursor declaration.

### LONG and LONG RAW Data Type Restrictions

LONG and LONG RAW data types in trigger bodies are subject to these restrictions:

- A SQL statement in a trigger can insert data into a column of LONG or LONG RAW data type.
- If data from a LONG or LONG RAW column can be converted to a constrained data type (such as CHAR and VARCHAR2), then a LONG or LONG RAW column can be referenced in a SQL statement in a trigger. The maximum length for these data types is 32000 bytes.
- Variables cannot be declared using the LONG or LONG RAW data types.
- :NEW and :PARENT cannot be used with LONG or LONG RAW columns.

### Mutating Table Restriction

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 INSTEAD OF triggers are not considered mutating.

When a trigger encounters a mutating table, a run-time error occurs, the effects of the trigger body and triggering statement are rolled back, and control is returned to the user or application. (You can use compound triggers to avoid the mutating-table error. For more information, see "Using Compound Triggers to Avoid Mutating-Table Error" on page 9-19.)

***Example 9–14 Row-Level Trigger Causes Mutating-Table Error***

```
CREATE OR REPLACE TRIGGER Emp_count
  AFTER DELETE ON employees
  FOR EACH ROW
DECLARE
  n  INTEGER;
BEGIN
  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || ' employees.');
END;
/
```

```
DELETE FROM employees WHERE employee_id = 197;
```

Result:

```
DELETE FROM employees WHERE employee_id = 197
            *
ERROR at line 1:
ORA-04091: table HR.EMPLOYEES is mutating, trigger/function may not see it
ORA-06512: at "HR.EMP_COUNT", line 4
ORA-04088: error during execution of trigger 'HR.EMP_COUNT'
```

Without FOR EACH ROW, the trigger is not subject to this restriction, as Example 9–15 shows.

***Example 9–15 Statement-Level Trigger Avoids Mutating-Table Error***

```
CREATE OR REPLACE TRIGGER Emp_count
  AFTER DELETE ON employees
DECLARE
  n  INTEGER;
BEGIN
  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || ' employees.');
END;
/

DELETE FROM employees WHERE employee_id = 197;
```

Result:

```
There are now 106 employees.

1 row deleted.
```

If you must update a mutating table, you can bypass these restrictions by using either a temporary table or a compound trigger that has a collection in its declarative part. For example, for a single AFTER row trigger that updates the original table, resulting in a mutating table error, you might use two triggers—an AFTER row trigger that updates a temporary table, and an AFTER statement trigger that updates the original table with the values from the temporary table.

Declarative constraints are checked at various times for row triggers.

Because declarative referential constraints are not supported between tables on different nodes of a distributed database, the mutating table restrictions do not apply to triggers that access remote nodes. These restrictions are also not enforced among tables in the same database that are connected by loop-back database links. A loop-back database link makes a local table appear remote by defining an Oracle Net path back to the database that contains the link.

### Restrictions on Mutating Tables Relaxed

The mutating error described in "Mutating Table Restriction" on page 9-28 prevents the trigger from reading or modifying the table that the parent statement is modifying. However, as of Oracle Database Release 8.1, a deletion from the parent table causes BEFORE and AFTER triggers to fire once. Therefore, you can create triggers (just not row triggers) to read and modify the parent and child tables.

This allows most foreign key constraint actions to be implemented through their obvious after-row trigger, if the constraint is not self-referential. Update cascade,

update set null, update set default, delete set default, inserting a missing parent, and maintaining a count of children can all be implemented easily.

***Example 9–16 Update Cascade***

```
DROP TABLE p;
CREATE TABLE p (p1 NUMBER CONSTRAINT pk_p_p1 PRIMARY KEY);

DROP TABLE f;
CREATE TABLE f (f1 NUMBER CONSTRAINT fk_f_f1 REFERENCES p);

CREATE TRIGGER pt
  AFTER UPDATE ON p
  FOR EACH ROW
BEGIN
  UPDATE f SET f1 = :NEW.p1 WHERE f1 = :OLD.p1;
END;
/
```

Cascades require care for multiple-row updates. For example, given the tables and trigger in Example 9–16: If table `p` has three rows with the values (1), (2), (3), and table `f` also has three rows with the values (1), (2), (3), then this statement updates `p` correctly but causes problems when the trigger updates `f`:

```
UPDATE p SET p1 = p1+1;
```

The statement first updates (1) to (2) in `p`, and the trigger updates (1) to (2) in `f`, leaving two rows of value (2) in `f`. Then the statement updates (2) to (3) in `p`, and the trigger updates both rows of value (2) to (3) in `f`. Finally, the statement updates (3) to (4) in `p`, and the trigger updates all three rows in f from (3) to (4). The relationship of the data in `p` and `f` is lost.

To avoid this problem, either forbid multiple-row updates to `p` that change the primary key and reuse existing primary key values, or track updates to foreign key values and modify the trigger to ensure that no row is updated twice.

That is the only problem with this technique for foreign key updates. The trigger cannot miss rows that were changed but not committed by another transaction, because the foreign key constraint guarantees that no matching foreign key rows are locked before the after-row trigger is invoked.

## System Trigger Restrictions

Depending on the event, different event attribute functions are available. For example, certain DDL operations might not be allowed on DDL events. Check "Event Attribute Functions" on page 9-52 before using an event attribute function, because its effects might be undefined rather than producing an error condition.

Only committed triggers fire. For example, if you create a trigger that fires after all `CREATE` events, then the trigger itself does not fire after the creation, because the correct information about this trigger was not committed at the time when the trigger on `CREATE` events fired.

For example, the following trigger does not fire after it is created:

```
CREATE OR REPLACE TRIGGER my_trigger
  AFTER CREATE ON DATABASE
BEGIN
  NULL;
END;
/
```

## Trigger Compilation

An important difference between triggers and PL/SQL anonymous blocks is their compilation. An anonymous block is compiled each time it is loaded into memory, and its compilation has three stages:

1. Syntax checking: PL/SQL syntax is checked, and a parse tree is generated.
2. Semantic checking: Type checking and further processing on the parse tree.
3. Code generation

A trigger is fully compiled when the `CREATE TRIGGER` statement runs. The trigger code is stored in the data dictionary. Therefore, it is unnecessary to open a shared cursor to run the trigger; the trigger runs directly.

If an error occurs during the compilation of a trigger, the trigger is still created. Therefore, if a DML statement fires the trigger, the DML statement fails (unless the trigger was created in the disabled state). To see trigger compilation errors, either use the `SHOW ERRORS` statement in SQL*Plus or Enterprise Manager, or `SELECT` the errors from the `USER_ERRORS` view.

Topics:

- Trigger Dependencies
- Trigger Recompilation

### Trigger Dependencies

Compiled triggers have dependencies. They become invalid if a depended-on object, such as a stored subprogram invoked from the trigger body, is modified. Triggers that are invalidated for dependency reasons are recompiled when next invoked.

You can examine the `ALL_DEPENDENCIES` view to see the dependencies for a trigger. For example, this statement shows the dependencies for the triggers in the `HR` schema:

```
SELECT NAME, REFERENCED_OWNER, REFERENCED_NAME, REFERENCED_TYPE
FROM ALL_DEPENDENCIES
WHERE OWNER = 'HR' and TYPE = 'TRIGGER';
```

Triggers might depend on other functions or packages. If the function or package specified in the trigger is dropped, then the trigger is marked invalid. An attempt is made to validate the trigger on occurrence of the event. If the trigger cannot be validated successfully, then it is marked `VALID WITH ERRORS`, and the event fails. For more information about dependencies between schema objects, see *Oracle Database Advanced Application Developer's Guide*.

---

**Note:**

- There is an exception for `STARTUP` events: `STARTUP` events succeed even if the trigger fails. There are also exceptions for `SHUTDOWN` events and for `LOGON` events if you login as `SYSTEM`.
- Because the `DBMS_AQ` package is used to enqueue a message, dependency between triggers and queues cannot be maintained.

---

### Trigger Recompilation

Use the `ALTER TRIGGER` statement to recompile a trigger manually. For example, this statement recompiles the `PRINT_SALARY_CHANGES` trigger:

```
ALTER TRIGGER Print_salary_changes COMPILE;
```

To recompile a trigger, you must own the trigger or have the ALTER ANY TRIGGER system privilege.

# Trigger Replacement

Like a stored subprogram, a trigger cannot be explicitly altered: It must be replaced with a new definition. (The ALTER TRIGGER statement is used only to recompile, enable, or disable a trigger.)

When replacing a trigger, you must include the OR REPLACE option in the CREATE TRIGGER statement. The OR REPLACE option is provided to allow a new version of an existing trigger to replace the older version, without affecting any grants made for the original version of the trigger.

Alternatively, the trigger can be dropped using the DROP TRIGGER statement, and you can rerun the CREATE TRIGGER statement.

To drop a trigger, the trigger must be in your schema, or you must have the DROP ANY TRIGGER system privilege.

# Trigger Debugging

You can debug a trigger using the same facilities available for stored subprograms. See *Oracle Database Advanced Application Developer's Guide*.

# Trigger Enabling

To enable a disabled trigger, use the ALTER TRIGGER statement with the ENABLE clause. For example, to enable the disabled trigger named Reorder, enter this statement:

```
ALTER TRIGGER Reorder ENABLE;
```

To enable all triggers defined for a specific table, use the ALTER TABLE statement with the ENABLE clause and the ALL TRIGGERS option. For example, to enable all triggers defined for the Inventory table, enter this statement:

```
ALTER TABLE Inventory ENABLE ALL TRIGGERS;
```

# Trigger Disabling

You might temporarily disable a trigger if:

- An object it references is not available.
- You must perform a large data load, and you want it to proceed quickly without firing triggers.
- You are reloading data.

To disable a trigger, use the ALTER TRIGGER statement with the DISABLE option. For example, to disable the trigger named Reorder, enter this statement:

```
ALTER TRIGGER Reorder DISABLE;
```

To disable all triggers defined for a specific table, use the ALTER TABLE statement with the DISABLE clause and the ALL TRIGGERS option. For example, to disable all triggers defined for the Inventory table, enter this statement:

```
ALTER TABLE Inventory DISABLE ALL TRIGGERS;
```

## Views for Information About Triggers

The *_TRIGGERS static data dictionary views reveal information about triggers.

The column BASE_OBJECT_TYPE specifies whether the trigger is based on DATABASE, SCHEMA, table, or view. The column TABLE_NAME is null if the base object is not table or view.

The column ACTION_TYPE specifies whether the trigger is a call type trigger or a PL/SQL trigger.

The column TRIGGER_TYPE specifies the type of the trigger; for example COMPOUND, BEFORE EVENT, or AFTER EVENT (the last two apply only to database events).

Each of the columns BEFORE_STATEMENT, BEFORE_ROW, AFTER_ROW, AFTER_STATEMENT, and INSTEAD_OF_ROW has the value YES or NO.

The column TRIGGERING_EVENT includes all system and DML events.

> **See Also:** *Oracle Database Reference* for information about *_TRIGGERS static data dictionary views

The query results in Example 9–17 were formatted by these SQL*Plus commands :

```
COLUMN Trigger_type FORMAT A15
COLUMN Triggering_event FORMAT A16
COLUMN Table_name FORMAT A11
COLUMN Trigger_body FORMAT A50
```

***Example 9–17 Viewing Information About Triggers***

```
CREATE OR REPLACE TRIGGER Emp_count
  AFTER DELETE ON employees
DECLARE
  n  INTEGER;
BEGIN
  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || ' employees.');
END;
/
```

Query:

```
SELECT Trigger_type, Triggering_event, Table_name
FROM USER_TRIGGERS
WHERE Trigger_name = 'EMP_COUNT';
```

Result:

```
TRIGGER_TYPE    TRIGGERING_EVENT TABLE_NAME
--------------- ---------------- -----------
AFTER STATEMENT DELETE           EMPLOYEES
```

Query:

```
SELECT Trigger_body
```

```
FROM USER_TRIGGERS
WHERE Trigger_name = 'EMP_COUNT';
```

Result:

```
TRIGGER_BODY
--------------------------------------------------
DECLARE
  n  INTEGER;
BEGIN
  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTP


1 row selected.
```

# Trigger Application Examples

You can use triggers in many ways to customize information management in the database. For example, triggers are commonly used to:

- Provide sophisticated auditing
- Prevent invalid transactions
- Enforce referential integrity (either those actions not supported by declarative constraints or across nodes in a distributed database)
- Enforce complex business rules
- Enforce complex security authorizations
- Provide transparent event logging
- Automatically generate derived column values
- Enable building complex views that are updatable
- Track database events

This section provides an example of each of these trigger applications. These examples are not meant to be used exactly as written: They are provided to assist you in designing your own triggers.

Topics:

- Triggers for Auditing
- Triggers and Constraints
- Triggers for Ensuring Referential Integrity

## Triggers for Auditing

Triggers are commonly used to supplement the built-in auditing features of the database. Although triggers can be written to record information similar to that recorded by the AUDIT statement, use triggers only when more detailed audit information is required. For example, use triggers to provide value-based auditing for each row.

Sometimes, the AUDIT statement is considered a *security* audit facility, while triggers can provide *financial* audit facility.

When deciding whether to create a trigger to audit database activity, consider what the database's auditing features provide, compared to auditing defined by triggers, as shown in Table 9–2.

***Table 9–2 Comparison of Built-in Auditing and Trigger-Based Auditing***

| Audit Feature | Description |
|---|---|
| DML and DDL Auditing | Standard auditing options permit auditing of DML and DDL statements regarding all types of schema objects and structures. Comparatively, *triggers* permit auditing of DML statements entered against tables, and DDL auditing at SCHEMA or DATABASE level. |
| Centralized Audit Trail | All database audit information is recorded centrally and automatically using the auditing features of the database. |
| Declarative Method | Auditing features enabled using the standard database features are easier to declare and maintain, and less prone to errors, when compared to auditing functions defined by triggers. |
| Auditing Options can be Audited | Any changes to existing auditing options can also be audited to guard against malicious database activity. |
| Session and Execution time Auditing | Using the database auditing features, records can be generated once every time an audited statement is entered (BY ACCESS) or once for every session that enters an audited statement (BY SESSION). Triggers cannot audit by session; an audit record is generated each time a trigger-audited table is referenced. |
| Auditing of Unsuccessful Data Access | Database auditing can be set to audit when unsuccessful data access occurs. However, unless autonomous transactions are used, any audit information generated by a trigger is rolled back if the triggering statement is rolled back. For more information about autonomous transactions, see *Oracle Database Concepts*. |
| Sessions can be Audited | Connections, disconnections, and session activity (physical I/Os, logical I/Os, deadlocks, and so on) can be recorded using standard database auditing. |

When using triggers to provide sophisticated auditing, AFTER triggers are normally used. The triggering statement is subjected to any applicable constraints. If no records are found, then the AFTER trigger does not fire, and audit processing is not carried out unnecessarily.

Choosing between AFTER row and AFTER statement triggers depends on the information being audited. For example, row triggers provide value-based auditing for each table row. Triggers can also require the user to supply a "reason code" for issuing the audited SQL statement, which can be useful in both row and statement-level auditing situations.

In Example 9–18, a trigger audits modifications to the Emp99 table for each row. The trigger requires that a reason code be stored in a global package variable before the update.

**Note:** Example 9–18 needs these data structures:

```
CREATE OR REPLACE PACKAGE Auditpackage
  AS
  Reason VARCHAR2(10);
  PROCEDURE Set_reason(Reason VARCHAR2);
END;
/
CREATE TABLE Emp (
   Empno              NUMBER(4),
   Ename              VARCHAR2(10),
   Job                VARCHAR2(9),
   Mgr                NUMBER(4),
   Hiredate           DATE,
   Sal                NUMBER(7,2),
   Comm               NUMBER(7,2),
   Deptno             NUMBER(2),
   Bonus              NUMBER,
   Ssn                NUMBER,
   Job_classification  NUMBER
);
CREATE TABLE Audit_employee (
   Oldssn             NUMBER,
   Oldname            VARCHAR2(10),
   Oldjob             VARCHAR2(2),
   Oldsal             NUMBER,
   Newssn             NUMBER,
   Newname            VARCHAR2(10),
   Newjob             VARCHAR2(2),
   Newsal             NUMBER,
   Reason             VARCHAR2(10),
   User1              VARCHAR2(10),
   Systemdate         DATE
);
```

***Example 9–18 Trigger for Auditing***

```
CREATE OR REPLACE TRIGGER audit_employee
  AFTER INSERT OR DELETE OR UPDATE ON Emp
  FOR EACH ROW
BEGIN
  /* AUDITPACKAGE is a package with a public package
     variable REASON. REASON can be set by the
     application by a statement such as EXECUTE
     AUDITPACKAGE.SET_REASON(reason_string).
     A package variable has state for the duration of a
     session and that each session has a separate copy of
     all package variables. */

  IF auditpackage.Reason IS NULL THEN
    raise_application_error(-20201, 'Must specify reason'
      || ' with AUDITPACKAGE.SET_REASON(Reason_string)');
  END IF;

  /* If preceding condition evaluates to TRUE,
     user-specified error number & message is raised,
     trigger stops execution, & effects of triggering statement are rolled back.
     Otherwise, row is inserted
```

```
     into predefined auditing table named AUDIT_EMPLOYEE
     containing existing & new values of the emp table
     & reason code defined by REASON variable of AUDITPACKAGE.
     "Old" values are NULL if triggering statement is INSERT
     & "new" values are NULL if triggering statement is DELETE. */

  INSERT INTO audit_employee (
   Oldssn, Oldname, Oldjob, Oldsal,
   Newssn, Newname, Newjob, Newsal,
   Reason, User1, Systemdate
  )
  VALUES (
    :OLD.Ssn, :OLD.Ename, :OLD.Job_classification, :OLD.Sal,
    :NEW.Ssn, :NEW.Ename, :NEW.Job_classification, :NEW.Sal,
    auditpackage.reason, user, sysdate
  );
END;
/
```

To force the reason code in Example 9–18 to be set for every update, use this trigger to set the reason code back to NULL with after the triggering statement runs:

```
CREATE OR REPLACE TRIGGER audit_employee_reset
  AFTER INSERT OR DELETE OR UPDATE ON emp
BEGIN
  auditpackage.set_reason(NULL);
END;
/
```

The triggers `audit_employee` and `audit_employee_reset` are fired by the same type of SQL statement. However, `audit_employee` fires once for each row of the table affected by the triggering statement, while `audit_employee_reset` fires only once after the triggering statement execution is completed.

Example 9–19 also uses triggers to do auditing. It tracks changes made to the `emp` table and stores this information in `audit_table` and `audit_table_values`.

---

**Note:** Example 9–19 needs these data structures:

```
CREATE TABLE audit_table (
   Seq      NUMBER,
   User_at  VARCHAR2(10),
   Time_now DATE,
   Term     VARCHAR2(10),
   Job      VARCHAR2(10),
   Proc     VARCHAR2(10),
   enum     NUMBER);
CREATE SEQUENCE audit_seq;
CREATE TABLE audit_table_values (
   Seq      NUMBER,
   Dept     NUMBER,
   Dept1    NUMBER,
   Dept2    NUMBER);
```

---

***Example 9–19 Trigger for Auditing***

```
CREATE OR REPLACE TRIGGER audit_emp
  AFTER INSERT OR UPDATE OR DELETE ON emp
  FOR EACH ROW
```

```
DECLARE
  Time_now DATE;
  Terminal CHAR(10);
BEGIN
  -- Get current time, & terminal of user:
  Time_now := SYSDATE;
  Terminal := USERENV('TERMINAL');

  -- Record new employee primary key:
  IF INSERTING THEN
    INSERT INTO audit_table VALUES (
      Audit_seq.NEXTVAL, User, Time_now,
      Terminal, 'emp', 'INSERT', :NEW.Empno
    );

    -- Record primary key of deleted row:
    ELSIF DELETING THEN
      INSERT INTO audit_table  VALUES (
        Audit_seq.NEXTVAL, User, Time_now,
        Terminal, 'emp', 'DELETE', :OLD.Empno
      );

    -- For updates, record primary key of row being updated:
    ELSE
      INSERT INTO audit_table (
        Seq, User_at, Time_now,
        Term, Job,  Proc, enum
      )
      VALUES (
        audit_seq.NEXTVAL, User, Time_now,
        Terminal, 'emp', 'UPDATE', :OLD.Empno
      );

       -- For SAL & DEPTNO, record old & new values:
       IF UPDATING ('SAL') THEN
         INSERT INTO audit_table_values (
           Seq, dept, dept1, dept2
         )
         VALUES (
           Audit_seq.CURRVAL, 'SAL', :OLD.Sal, :NEW.Sal
         );

       ELSIF UPDATING ('DEPTNO') THEN
         INSERT INTO audit_table_values (
           Seq, dept, dept1, dept2
         )
         VALUES (
           Audit_seq.CURRVAL, 'DEPTNO', :OLD.Deptno, :NEW.DEPTNO
         );
       END IF;
    END IF;
END;
/
```

## Triggers and Constraints

Triggers and constraints can both be used to constrain data input. However, triggers and constraints have significant differences.

Declarative constraints are statements about the database that are always true. A constraint applies to existing data in the table and any statement that manipulates the table.

**See Also:** *Oracle Database Advanced Application Developer's Guide*

Triggers constrain what a transaction can do. A trigger does not apply to data loaded before the definition of the trigger; therefore, it is not known if all data in a table conforms to the rules established by an associated trigger.

Although triggers can be written to enforce many of the same rules supported by declarative constraint features, use triggers only to enforce complex business rules that cannot be defined using standard constraints. The declarative constraint features provided with the database offer these advantages when compared to constraints defined by triggers:

- Centralized integrity checks

  All points of data access must adhere to the global set of rules defined by the constraints corresponding to each schema object.

- Declarative method

  Constraints defined using the standard constraint features are much easier to write and are less prone to errors, when compared with comparable constraints defined by triggers.

While most aspects of data integrity can be defined and enforced using declarative constraints, triggers can be used to enforce complex business constraints not definable using declarative constraints. For example, triggers can be used to enforce:

- `UPDATE SET NULL`, and `UPDATE` and `DELETE SET DEFAULT` referential actions.
- Referential integrity when the parent and child tables are on different nodes of a distributed database.
- Complex check constraints not definable using the expressions allowed in a `CHECK` constraint.

## Triggers for Ensuring Referential Integrity

Use triggers only when performing an action for which there is no declarative support.

When using triggers to maintain referential integrity, declare the `PRIMARY` (or `UNIQUE`) `KEY` constraint in the parent table. If referential integrity is being maintained between a parent and child table in the same database, then you can also declare the foreign key in the child table, but disable it. Disabling the trigger in the child table prevents the corresponding `PRIMARY KEY` constraint from being dropped (unless the `PRIMARY KEY` constraint is explicitly dropped with the `CASCADE` option).

To maintain referential integrity using triggers:

- For the child table, define a trigger that ensures that values inserted or updated in the foreign key correspond to values in the parent key.
- For the parent table, define one or more triggers that ensure the desired referential action (`RESTRICT`, `CASCADE`, or `SET NULL`) for values in the foreign key when values in the parent key are updated or deleted. No action is required for inserts into the parent table (no dependent foreign keys exist).

Topics:

- Foreign Key Trigger for Child Table

- UPDATE and DELETE RESTRICT Triggers for Parent Table
- UPDATE and DELETE SET NULL Triggers for Parent Table
- DELETE Cascade Trigger for Parent Table
- UPDATE Cascade Trigger for Parent Table
- Triggers for Complex Check Constraints
- Triggers for Complex Security Authorizations
- Triggers for Transparent Event Logging
- Triggers for Deriving Column Values
- Triggers for Building Complex Updatable Views
- Triggers for Fine-Grained Access Control

The examples in the following sections use the `emp` and `dept` table relationship. Several of the triggers include statements that lock rows (`SELECT FOR UPDATE`). This operation is necessary to maintain concurrency as the rows are being processed.

### Foreign Key Trigger for Child Table

The trigger in Example 9–20 guarantees that before an `INSERT` or `UPDATE` statement affects a foreign key value, the corresponding value exists in the parent key. The mutating table exception included in Example 9–20 allows the trigger `emp_dept_check` to be used with the `UPDATE_SET_DEFAULT` and `UPDATE_CASCADE` triggers. This exception can be removed if the trigger `emp_dept_check` is used alone.

***Example 9–20 Foreign Key Trigger for Child Table***

```
CREATE OR REPLACE TRIGGER emp_dept_check
  BEFORE INSERT OR UPDATE OF Deptno ON emp
  FOR EACH ROW WHEN (new.Deptno IS NOT NULL)

  -- Before row is inserted or DEPTNO is updated in emp table,
  -- fire this trigger to verify that new foreign key value (DEPTNO)
  -- is present in dept table.
DECLARE
  Dummy               INTEGER;  -- Use for cursor fetch
  Invalid_department  EXCEPTION;
  Valid_department    EXCEPTION;
  Mutating_table      EXCEPTION;
  PRAGMA EXCEPTION_INIT (Mutating_table, -4091);

  -- Cursor used to verify parent key value exists.
  -- If present, lock parent key's row so it cannot be deleted
  -- by another transaction until this transaction is
  -- committed or rolled back.

  CURSOR Dummy_cursor (Dn NUMBER) IS
    SELECT Deptno FROM dept
    WHERE Deptno = Dn
    FOR UPDATE OF Deptno;
BEGIN
  OPEN Dummy_cursor (:NEW.Deptno);
  FETCH Dummy_cursor INTO Dummy;

  -- Verify parent key.
  -- If not found, raise user-specified error number & message.
  -- If found, close cursor before allowing triggering statement to complete:
```

```
  IF Dummy_cursor%NOTFOUND THEN
    RAISE Invalid_department;
  ELSE
    RAISE valid_department;
  END IF;
  CLOSE Dummy_cursor;
EXCEPTION
  WHEN Invalid_department THEN
    CLOSE Dummy_cursor;
    Raise_application_error(-20000, 'Invalid Department'
      || ' Number' || TO_CHAR(:NEW.deptno));
  WHEN Valid_department THEN
    CLOSE Dummy_cursor;
  WHEN Mutating_table THEN
    NULL;
END;
/
```

### UPDATE and DELETE RESTRICT Triggers for Parent Table

The trigger in Example 9–21 enforces the UPDATE and DELETE RESTRICT referential action on the primary key of the dept table.

> **Caution:** The trigger dept_restrict does not work with self-referential tables (tables with both the primary/unique key and the foreign key). Also, this trigger does not allow triggers to cycle (such as when A fires B, which fires A).

***Example 9–21 UPDATE and DELETE RESTRICT Trigger for Parent Table***

```
CREATE OR REPLACE TRIGGER dept_restrict
  BEFORE DELETE OR UPDATE OF Deptno ON dept
  FOR EACH ROW

  -- Before row is deleted from dept or primary key (DEPTNO) of dept is updated,
  -- check for dependent foreign key values in emp;
  -- if any are found, roll back.

DECLARE
  Dummy                  INTEGER;  -- Use for cursor fetch
  Employees_present      EXCEPTION;
  employees_not_present  EXCEPTION;

  -- Cursor used to check for dependent foreign key values.
  CURSOR Dummy_cursor (Dn NUMBER) IS
    SELECT Deptno FROM emp WHERE Deptno = Dn;

BEGIN
  OPEN Dummy_cursor (:OLD.Deptno);
  FETCH Dummy_cursor INTO Dummy;

  -- If dependent foreign key is found, raise user-specified
  -- error number and message. If not found, close cursor
  -- before allowing triggering statement to complete.

  IF Dummy_cursor%FOUND THEN
    RAISE Employees_present;     -- Dependent rows exist
```

```
  ELSE
    RAISE Employees_not_present; -- No dependent rows exist
  END IF;
  CLOSE Dummy_cursor;

EXCEPTION
  WHEN Employees_present THEN
    CLOSE Dummy_cursor;
    Raise_application_error(-20001, 'Employees Present in'
      || ' Department ' || TO_CHAR(:OLD.DEPTNO));
  WHEN Employees_not_present THEN
    CLOSE Dummy_cursor;
END;
```

### UPDATE and DELETE SET NULL Triggers for Parent Table

The trigger in Example 9–22 enforces the `UPDATE` and `DELETE SET NULL` referential action on the primary key of the `dept` table.

***Example 9–22 UPDATE and DELETE SET NULL Triggers for Parent Table***

```
CREATE OR REPLACE TRIGGER dept_set_null
  AFTER DELETE OR UPDATE OF Deptno ON dept
  FOR EACH ROW

  -- Before row is deleted from dept or primary key (DEPTNO) of dept is updated,
  -- set all corresponding dependent foreign key values in emp to NULL:

BEGIN
  IF UPDATING AND :OLD.Deptno != :NEW.Deptno OR DELETING THEN
    UPDATE emp SET emp.Deptno = NULL
    WHERE emp.Deptno = :OLD.Deptno;
  END IF;
END;
/
```

### DELETE Cascade Trigger for Parent Table

The trigger in Example 9–23 enforces the `DELETE CASCADE` referential action on the primary key of the `dept` table.

***Example 9–23 DELETE Cascade Trigger for Parent Table***

```
CREATE OR REPLACE TRIGGER dept_del_cascade
  AFTER DELETE ON dept
  FOR EACH ROW

  -- Before row is deleted from dept,
  -- delete all rows from emp table whose DEPTNO is same as
  -- DEPTNO being deleted from dept table:

BEGIN
  DELETE FROM emp
  WHERE emp.Deptno = :OLD.Deptno;
END;
/
```

**Note:** Typically, the code for DELETE CASCADE is combined with the code for UPDATE SET NULL or UPDATE SET DEFAULT, to account for both updates and deletes.

### UPDATE Cascade Trigger for Parent Table

The triggers in Example 9–24 ensure that if a department number is updated in the dept table, then this change is propagated to dependent foreign keys in the emp table.

***Example 9–24 UPDATE Cascade Trigger for Parent Table***

```
-- Generate sequence number to be used as flag
-- for determining if update occurred on column:

CREATE SEQUENCE Update_sequence
  INCREMENT BY 1 MAXVALUE 5000 CYCLE;

CREATE OR REPLACE PACKAGE Integritypackage AS
  Updateseq NUMBER;
END Integritypackage;
/
CREATE OR REPLACE PACKAGE BODY Integritypackage AS
END Integritypackage;
/
-- Create flag col:

ALTER TABLE emp ADD Update_id NUMBER;

CREATE OR REPLACE TRIGGER dept_cascade1
  BEFORE UPDATE OF Deptno ON dept
DECLARE
  -- Before updating dept table (this is a statement trigger),
  -- generate sequence number
  -- & assign it to public variable UPDATESEQ of
  -- user-defined package named INTEGRITYPACKAGE:
BEGIN
  Integritypackage.Updateseq := Update_sequence.NEXTVAL;
END;
/
CREATE OR REPLACE TRIGGER dept_cascade2
  AFTER DELETE OR UPDATE OF Deptno ON dept
  FOR EACH ROW

  -- For each department number in dept that is updated,
  -- cascade update to dependent foreign keys in emp table.
  -- Cascade update only if child row was not updated by this trigger:
BEGIN
  IF UPDATING THEN
    UPDATE emp
    SET Deptno = :NEW.Deptno,
        Update_id = Integritypackage.Updateseq   --from 1st
    WHERE emp.Deptno = :OLD.Deptno
    AND Update_id IS NULL;

    /* Only NULL if not updated by 3rd trigger
       fired by same triggering statement */
  END IF;
  IF DELETING THEN
    -- Before row is deleted from dept,
```

```
    -- delete all rows from emp table whose DEPTNO is same as
    -- DEPTNO being deleted from dept table:
    DELETE FROM emp
    WHERE emp.Deptno = :OLD.Deptno;
  END IF;
END;
/
CREATE OR REPLACE TRIGGER dept_cascade3
  AFTER UPDATE OF Deptno ON dept
BEGIN UPDATE emp
  SET Update_id = NULL
  WHERE Update_id = Integritypackage.Updateseq;
END;
/
```

---

**Note:** Because the trigger `dept_cascade2` updates the `emp` table, the `emp_dept_check` trigger in Example 9–20, if enabled, also fires. The resulting mutating table error is trapped by the `emp_dept_check` trigger. Carefully test any triggers that require error trapping to succeed to ensure that they always work properly in your environment.

---

### Triggers for Complex Check Constraints

Triggers can enforce integrity rules other than referential integrity. The trigger in Example 9–25 performs a complex check before allowing the triggering statement to run.

---

**Note:** Example 9–25 needs this data structure:

```
CREATE TABLE Salgrade (
  Grade               NUMBER,
  Losal               NUMBER,
  Hisal               NUMBER,
  Job_classification  NUMBER);
```

---

***Example 9–25 Trigger for Complex Check Constraints***

```
CREATE OR REPLACE TRIGGER salary_check
  BEFORE INSERT OR UPDATE OF Sal, Job ON Emp
  FOR EACH ROW
DECLARE
  Minsal               NUMBER;
  Maxsal               NUMBER;
  Salary_out_of_range  EXCEPTION;

BEGIN
  /* Retrieve minimum & maximum salary for employee's new job classification
     from SALGRADE table into MINSAL and MAXSAL: */

  SELECT Minsal, Maxsal INTO Minsal, Maxsal
  FROM Salgrade
  WHERE Job_classification = :NEW.Job;

  /* If employee's new salary is less than or greater than
     job classification's limits, raise exception.
     Exception message is returned and pending INSERT or UPDATE statement
```

```
      that fired the trigger is rolled back: */

  IF (:NEW.Sal < Minsal OR :NEW.Sal > Maxsal) THEN
    RAISE Salary_out_of_range;
  END IF;
EXCEPTION
  WHEN Salary_out_of_range THEN
    Raise_application_error (
      -20300,
      'Salary '|| TO_CHAR(:NEW.Sal) ||' out of range for '
      || 'job classification ' ||:NEW.Job
      ||' for employee ' || :NEW.Ename
    );
  WHEN NO_DATA_FOUND THEN
    Raise_application_error(-20322, 'Invalid Job Classification');
END;
/
```

### Triggers for Complex Security Authorizations

Triggers are commonly used to enforce complex security authorizations for table data. Only use triggers to enforce complex security authorizations that cannot be defined using the database security features provided with the database. For example, a trigger can prohibit updates to salary data of the `emp` table during weekends, holidays, and nonworking hours.

When using a trigger to enforce a complex security authorization, it is best to use a `BEFORE` statement trigger. Using a `BEFORE` statement trigger has these benefits:

- The security check is done before the triggering statement is allowed to run, so that no wasted work is done by an unauthorized statement.
- The security check is performed only once for the triggering statement, not for each row affected by the triggering statement.

The trigger in Example 9–26 enforces security.

---

**Note:** Example 9–26 needs this data structure:

```
CREATE TABLE Company_holidays (Day DATE);
```

---

***Example 9–26 Trigger for Enforcing Security***

```
CREATE OR REPLACE TRIGGER Emp_permit_changes
  BEFORE INSERT OR DELETE OR UPDATE ON Emp
DECLARE
  Dummy              INTEGER;
  Not_on_weekends    EXCEPTION;
  Not_on_holidays    EXCEPTION;
  Non_working_hours  EXCEPTION;
BEGIN
   /* Check for weekends: */
   IF (TO_CHAR(Sysdate, 'DY') = 'SAT' OR
     TO_CHAR(Sysdate, 'DY') = 'SUN') THEN
       RAISE Not_on_weekends;
   END IF;

   /* Check for company holidays: */
   SELECT COUNT(*) INTO Dummy FROM Company_holidays
     WHERE TRUNC(Day) = TRUNC(Sysdate); -- Discard time parts of dates
```

```
    IF dummy > 0 THEN
      RAISE Not_on_holidays;
    END IF;

   /* Check for work hours (8am to 6pm): */
   IF (TO_CHAR(Sysdate, 'HH24') < 8 OR
     TO_CHAR(Sysdate, 'HH24') > 18) THEN
       RAISE Non_working_hours;
   END IF;
EXCEPTION
  WHEN Not_on_weekends THEN
    Raise_application_error(-20324,'Might not change '
      ||'employee table during the weekend');
  WHEN Not_on_holidays THEN
    Raise_application_error(-20325,'Might not change '
      ||'employee table during a holiday');
  WHEN Non_working_hours THEN
    Raise_application_error(-20326,'Might not change '
     ||'emp table during nonworking hours');
END;
/
```

**See Also:** *Oracle Database Security Guide* for detailed information about database security features

### Triggers for Transparent Event Logging

Triggers are very useful when you want to transparently perform a related change in the database following certain events.

The REORDER trigger example shows a trigger that reorders parts as necessary when certain conditions are met. (In other words, a triggering statement is entered, and the PARTS_ON_HAND value is less than the REORDER_POINT value.)

### Triggers for Deriving Column Values

Triggers can derive column values automatically, based upon a value provided by an INSERT or UPDATE statement. This type of trigger is useful to force values in specific columns that depend on the values of other columns in the same row. BEFORE row triggers are necessary to complete this type of operation for these reasons:

- The dependent values must be derived before the INSERT or UPDATE occurs, so that the triggering statement can use the derived values.
- The trigger must fire for each row affected by the triggering INSERT or UPDATE statement.

The trigger in Example 9–27 derives new column values for a table whenever a row is inserted or updated.

---

**Note:** Example 9–27 needs this change to this data structure:

```
ALTER TABLE Emp ADD(
   Uppername   VARCHAR2(20),
   Soundexname VARCHAR2(20));
```

---

***Example 9–27 Trigger That Derives New Column Values for Table***

```
CREATE OR REPLACE TRIGGER Derived
BEFORE INSERT OR UPDATE OF Ename ON Emp
```

```
/* Before updating the ENAME field, derive the values for
   the UPPERNAME and SOUNDEXNAME fields. Restrict users
   from updating these fields directly: */
FOR EACH ROW
BEGIN
  :NEW.Uppername := UPPER(:NEW.Ename);
  :NEW.Soundexname := SOUNDEX(:NEW.Ename);
END;
/
```

### Triggers for Building Complex Updatable Views

Views are an excellent mechanism to provide logical windows over table data. However, when the view query gets complex, the system implicitly cannot translate the DML on the view into those on the underlying tables. `INSTEAD OF` triggers help solve this problem. These triggers can be defined over views, and they fire instead of the actual DML.

Consider a library system where books are arranged by title. The library consists of a collection of book type objects:

```
CREATE OR REPLACE TYPE Book_t AS OBJECT (
  Booknum    NUMBER,
  Title      VARCHAR2(20),
  Author     VARCHAR2(20),
  Available  CHAR(1)
);
/
CREATE OR REPLACE TYPE Book_list_t AS TABLE OF Book_t;
/
```

The table `Book_table` is created and populated like this:

```
DROP TABLE Book_table;
CREATE TABLE Book_table (
  Booknum    NUMBER,
  Section    VARCHAR2(20),
  Title      VARCHAR2(20),
  Author     VARCHAR2(20),
  Available  CHAR(1)
);

INSERT INTO Book_table (
  Booknum, Section, Title, Author, Available
)
VALUES (
  121001, 'Classic', 'Iliad', 'Homer', 'Y'
);

INSERT INTO Book_table (
  Booknum, Section, Title, Author, Available
)
VALUES (
  121002, 'Novel', 'Gone with the Wind', 'Mitchell M', 'N'
);

SELECT * FROM Book_table;
```

Result:

```
   BOOKNUM SECTION              TITLE                AUTHOR               A
---------- -------------------- -------------------- -------------------- -
    121001 Classic              Iliad                Homer                Y
    121002 Novel                Gone with the Wind   Mitchell M           N

2 rows selected.
```

The table `Library_table` is created and populated like this:

```
DROP TABLE Library_table;
CREATE TABLE Library_table (Section VARCHAR2(20));

INSERT INTO Library_table (Section)
VALUES ('Novel');

INSERT INTO Library_table (Section)
VALUES ('Classic');

SELECT * FROM Library_table;
```

Result:

```
SECTION
--------------------
Novel
Classic

2 rows selected.
```

You can define a complex view over the tables `Book_table` and `Library_table` to create a logical view of the library with sections and a collection of books in each section:

```
CREATE OR REPLACE VIEW Library_view AS
  SELECT i.Section, CAST (
    MULTISET (
      SELECT b.Booknum, b.Title, b.Author, b.Available
      FROM Book_table b
      WHERE b.Section = i.Section
    ) AS Book_list_t
  ) BOOKLIST
  FROM Library_table i;
```

(For information about the `CAST` function, see *Oracle Database SQL Language Reference*.)

Make `Library_view` updatable by defining an `INSTEAD OF` trigger on it:

```
CREATE OR REPLACE TRIGGER Library_trigger
  INSTEAD OF
  INSERT ON Library_view
  FOR EACH ROW
DECLARE
  Bookvar  Book_t;
  i        INTEGER;
BEGIN
  INSERT INTO Library_table
  VALUES (:NEW.Section);

  FOR i IN 1..:NEW.Booklist.COUNT LOOP
    Bookvar := :NEW.Booklist(i);

    INSERT INTO Book_table (
```

```
      Booknum, Section, Title, Author, Available
    )
    VALUES (
      Bookvar.booknum, :NEW.Section, Bookvar.Title,
      Bookvar.Author, bookvar.Available
    );
  END LOOP;
END;
/
```

Insert a new row into `Library_view`:

```
INSERT INTO Library_view (Section, Booklist)
VALUES (
  'History',
  book_list_t (book_t (121330, 'Alexander', 'Mirth', 'Y'))
);
```

See the effect on `Library_view`:

```
SELECT * FROM Library_view;
```

Result:

```
SECTION
--------------------
BOOKLIST(BOOKNUM, TITLE, AUTHOR, AVAILABLE)
--------------------------------------------------------------------

Novel
BOOK_LIST_T(BOOK_T(121002, 'Gone with the Wind', 'Mitchell M', 'N'))

Classic
BOOK_LIST_T(BOOK_T(121001, 'Iliad', 'Homer', 'Y'))

History
BOOK_LIST_T(BOOK_T(121330, 'Alexander', 'Mirth', 'Y'))


3 rows selected.
```

See the effect on `Book_table`:

```
SELECT * FROM Book_table;
```

Result:

```
   BOOKNUM SECTION              TITLE                AUTHOR               A
---------- -------------------- -------------------- -------------------- -
    121001 Classic              Iliad                Homer                Y
    121002 Novel                Gone with the Wind   Mitchell M           N
    121330 History              Alexander            Mirth                Y

3 rows selected.
```

See the effect on `Library_table`:

```
SELECT * FROM Library_table;
```

Result:

```
SECTION
--------------------
```

```
Novel
Classic
History

3 rows selected.
```

Similarly, you can also define triggers on the nested table `booklist` to handle modification of the nested table element.

### Triggers for Fine-Grained Access Control

You can use `LOGON` triggers to run the package associated with an application context. An application context captures session-related information about the user who is logging in to the database. From there, your application can control how much access this user has, based on his or her session information.

> **Note:** If you have very specific logon requirements, such as preventing users from logging in from outside the firewall or after work hours, consider using Oracle Database Vault instead of `LOGON` triggers. With Oracle Database Vault, you can create custom rules to strictly control user access.

**See Also:**

- *Oracle Database Security Guide* for information about creating a `LOGON` trigger to run a database session application context package
- *Oracle Database Vault Administrator's Guide* for information about Oracle Database Vault

## Triggers that Publish Database Events

> **Note:** This topic applies only to simple triggers.

Database event publication lets applications subscribe to database events, just like they subscribe to messages from other applications. The database events publication framework includes these features:

- Infrastructure for publish/subscribe, by making the database an active publisher of events.
- Integration of data cartridges in the server. The database events publication can be used to notify cartridges of state changes in the server.
- Integration of fine-grained access control in the server.

By creating a trigger, you can specify a subprogram that runs when an event occurs. DML events are supported on tables, and database events are supported on `DATABASE` and `SCHEMA`. You can turn notification on and off by enabling and disabling the trigger using the `ALTER TRIGGER` statement.

This feature is integrated with the Advanced Queueing engine. Publish/subscribe applications use the `DBMS_AQ.ENQUEUE` procedure, and other applications such as cartridges use callouts.

**See Also:**

- "ALTER TRIGGER Statement" on page 14-14
- *Oracle Streams Advanced Queuing User's Guide* for details on how to subscribe to published events

Topics:


- How Triggers Publish Events
- Publication Context
- Error Handling
- Execution Model
- Event Attribute Functions
- Database Events
- Client Events


## How Triggers Publish Events

When the database detects an event, the trigger mechanism runs the action specified in the trigger. The action can include publishing the event to a queue so that subscribers receive notifications. To publish events, use the `DBMS_AQ` package.

---

**Note:** The database can detect only system-defined events. You cannot define your own events.

---

When it detects an event, the database fires all triggers that are enabled on that event, except:

- Any trigger that is the target of the triggering event.

  For example, a trigger for all `DROP` events does not fire when it is dropped itself.

- Any trigger that was modified, but not committed, in the same transaction as the triggering event.

  For example, recursive DDL in a system trigger might modify a trigger, which prevents the modified trigger from being fired by events in the same transaction.

**See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_AQ` package

## Publication Context

When an event is published, certain run-time context and attributes, as specified in the parameter list, are passed to the callout subprogram. A set of functions called event attribute functions are provided.

**See Also:** "Event Attribute Functions" on page 9-52 for information about event-specific attributes

For each supported database event, you can identify and predefine event-specific attributes for the event. You can choose the parameter list to be any of these attributes, along with other simple expressions. For callouts, these are passed as `IN` arguments.

## Error Handling

Return status from publication callout functions for all events are ignored. For example, with SHUTDOWN events, the database cannot do anything with the return status.

## Execution Model

Traditionally, triggers run as the definer of the trigger. The trigger action of an event runs as the definer of the action (as the definer of the package or function in callouts, or as owner of the trigger in queues). Because the owner of the trigger must have EXECUTE privileges on the underlying queues, packages, or subprograms, this action is consistent.

## Event Attribute Functions

When the database fires a trigger, you can retrieve certain attributes about the event that fired the trigger. You can retrieve each attribute with a function call. Table 9–3 describes the system-defined event attributes.

**Note:**

- The trigger dictionary object maintains metadata about events to be published and their corresponding attributes.
- In earlier releases, these functions were accessed through the SYS package. Oracle recommends you use these public synonyms whose names begin with ora_.
- ora_name_list_t is defined in package DBMS_STANDARD as

```
TYPE ora_name_list_t IS TABLE OF VARCHAR2(64);
```

***Table 9–3 System-Defined Event Attributes***

| Attribute | Type | Description | Example |
|---|---|---|---|
| ora_client_ip_address | VARCHAR2 | Returns IP address of the client in a LOGON event when the underlying protocol is TCP/IP | `DECLARE`<br>`  v_addr VARCHAR2(11);`<br>`BEGIN`<br>`  IF (ora_sysevent = 'LOGON') THEN`<br>`    v_addr := ora_client_ip_address;`<br>`  END IF;`<br>`END;`<br>`/` |
| ora_database_name | VARCHAR2(50) | Database name. | `DECLARE`<br>`  v_db_name VARCHAR2(50);`<br>`BEGIN`<br>`  v_db_name := ora_database_name;`<br>`END;`<br>`/` |
| ora_des_encrypted_password | VARCHAR2 | The DES-encrypted password of the user being created or altered. | `IF (ora_dict_obj_type = 'USER') THEN`<br>`  INSERT INTO event_table`<br>`    VALUES (ora_des_encrypted_password);`<br>`END IF;` |
| ora_dict_obj_name | VARCHAR(30) | Name of the dictionary object on which the DDL operation occurred. | `INSERT INTO event_table`<br>`  VALUES ('Changed object is ' \|\|`<br>`          ora_dict_obj_name);` |

***Table 9–3 (Cont.) System-Defined Event Attributes***

<table>
<tr><th>Attribute</th><th>Type</th><th>Description</th><th>Example</th></tr>
<tr><td>ora_dict_obj_name_list<br>(name_list OUT ora_name_list_<br>t)</td><td>PLS_INTEGER</td><td>Return the list of object names of objects being modified in the event.</td><td>DECLARE<br>name_list   DBMS_STANDARD.ora_name_list_t;<br>number_modified  PLS_INTEGER;<br>BEGIN<br>IF (ora_sysevent='ASSOCIATE STATISTICS')<br>  THEN number_modified :=<br>       ora_dict_obj_name_list(name_list);<br>END IF;<br>END;</td></tr>
<tr><td>ora_dict_obj_owner</td><td>VARCHAR(30)</td><td>Owner of the dictionary object on which the DDL operation occurred.</td><td>INSERT INTO event_table<br>  VALUES ('object owner is' ||<br>          ora_dict_obj_owner);</td></tr>
<tr><td>ora_dict_obj_owner_list<br>(owner_list OUT ora_name_<br>list_t)</td><td>PLS_INTEGER</td><td>Returns the list of object owners of objects being modified in the event.</td><td>DECLARE<br>owner_list<br>   DBMS_STANDARD.ora_name_list_t;<br>number_modified   PLS_INTEGER;<br>BEGIN<br>IF (ora_sysevent='ASSOCIATE STATISTICS')<br>  THEN number_modified :=<br>       ora_dict_obj_name_list(owner_list);<br>END IF;<br>END;</td></tr>
<tr><td>ora_dict_obj_type</td><td>VARCHAR(20)</td><td>Type of the dictionary object on which the DDL operation occurred.</td><td>INSERT INTO event_table<br>  VALUES ('This object is a ' ||<br>          ora_dict_obj_type);</td></tr>
<tr><td>ora_grantee<br>(user_list OUT ora_name_list_<br>t)</td><td>PLS_INTEGER</td><td>Returns the grantees of a grant event in the OUT parameter; returns the number of grantees in the return value.</td><td>DECLARE<br>user_list   DBMS_STANDARD.ora_name_list_t;<br>number_of_grantees   PLS_INTEGER;<br>BEGIN<br>IF (ora_sysevent = 'GRANT') THEN<br>number_of_grantees :=<br> ora_grantee(user_list);<br>END IF;<br>END;</td></tr>
<tr><td>ora_instance_num</td><td>NUMBER</td><td>Instance number.</td><td>IF (ora_instance_num = 1) THEN<br>  INSERT INTO event_table VALUES ('1');<br>END IF;</td></tr>
<tr><td>ora_is_alter_column<br>(column_name IN VARCHAR2)</td><td>BOOLEAN</td><td>Returns true if the specified column is altered.</td><td>IF (ora_sysevent = 'ALTER' AND<br>    ora_dict_obj_type = 'TABLE') THEN<br> alter_column := ora_is_alter_column('C');<br>END IF;</td></tr>
<tr><td>ora_is_creating_nested_table</td><td>BOOLEAN</td><td>Returns true if the current event is creating a nested table</td><td>IF (ora_sysevent = 'CREATE' and<br>    ora_dict_obj_type = 'TABLE' and<br>    ora_is_creating_nested_table) THEN<br>  INSERT INTO event_table<br>    VALUES ('A nested table is created');<br>END IF;</td></tr>
<tr><td>ora_is_drop_column<br>(column_name IN VARCHAR2)</td><td>BOOLEAN</td><td>Returns true if the specified column is dropped.</td><td>IF (ora_sysevent = 'ALTER' AND<br>    ora_dict_obj_type = 'TABLE') THEN<br> drop_column := ora_is_drop_column('C');<br>END IF;</td></tr>
<tr><td>ora_is_servererror</td><td>BOOLEAN</td><td>Returns TRUE if given error is on error stack, FALSE otherwise.</td><td>IF ora_is_servererror(error_number) THEN<br> INSERT INTO event_table<br>   VALUES ('Server error!!');<br>END IF;</td></tr>
<tr><td>ora_login_user</td><td>VARCHAR2(30)</td><td>Login user name.</td><td>SELECT ora_login_user<br>FROM DUAL;</td></tr>
</table>

*Table 9–3 (Cont.) System-Defined Event Attributes*

| Attribute | Type | Description | Example |
|---|---|---|---|
| `ora_partition_pos` | `PLS_INTEGER` | In an `INSTEAD OF` trigger for `CREATE TABLE`, the position in the SQL text where you can insert a `PARTITION` clause. | `-- Retrieve ora_sql_txt into`<br>`-- sql_text variable first.`<br>`v_n := ora_partition_pos;`<br>`v_new_stmt := SUBSTR(sql_text,1,v_n - 1)`<br>`         \|\| ' ' \|\| my_partition_clause`<br>`         \|\| ' ' \|\| SUBSTR(sql_text, v_n));` |
| `ora_privilege_list (privilege_list OUT ora_name_list_t)` | `PLS_INTEGER` | Returns the list of privileges being granted by the grantee or the list of privileges revoked from the revokees in the `OUT` parameter; returns the number of privileges in the return value. | `DECLARE`<br>`privelege_list`<br>`        DBMS_STANDARD.ora_name_list_t;`<br>`number_of_privileges   PLS_INTEGER;`<br>`BEGIN`<br>`IF (ora_sysevent = 'GRANT' OR`<br>`    ora_sysevent = 'REVOKE') THEN`<br>`  number_of_privileges :=`<br>`    ora_privilege_list(privilege_list);`<br>`END IF;`<br>`END;` |
| `ora_revokee (user_list OUT ora_name_list_t)` | `PLS_INTEGER` | Returns the revokees of a revoke event in the `OUT` parameter; returns the number of revokees in the return value. | `DECLARE`<br>`user_list   DBMS_STANDARD.ora_name_list_t;`<br>`number_of_users   PLS_INTEGER;`<br>`BEGIN`<br>`IF (ora_sysevent = 'REVOKE') THEN`<br>`number_of_users := ora_revokee(user_list);`<br>`END IF;`<br>`END;` |
| `ora_server_error` | `NUMBER` | Given a position (1 for top of stack), it returns the error number at that position on error stack | `INSERT INTO event_table`<br>`  VALUES ('top stack error ' \|\|`<br>`           ora_server_error(1));` |
| `ora_server_error_depth` | `PLS_INTEGER` | Returns the total number of error messages on the error stack. | `n := ora_server_error_depth;`<br>`-- This value is used with other functions`<br>`-- such as ora_server_error` |
| `ora_server_error_msg (position in pls_integer)` | `VARCHAR2` | Given a position (1 for top of stack), it returns the error message at that position on error stack | `INSERT INTO event_table`<br>`  VALUES ('top stack error message' \|\|`<br>`           ora_server_error_msg(1));` |
| `ora_server_error_num_params (position in pls_integer)` | `PLS_INTEGER` | Given a position (1 for top of stack), it returns the number of strings that were substituted into the error message using a format like `%s`. | `n := ora_server_error_num_params(1);` |
| `ora_server_error_param (position in pls_integer, param in pls_integer)` | `VARCHAR2` | Given a position (1 for top of stack) and a parameter number, returns the matching substitution value (`%s`, `%d`, and so on) in the error message. | `-- For example, the second %s in a`<br>`-- message: "Expected %s, found %s"`<br>`param := ora_server_error_param(1,2);` |

*Table 9–3 (Cont.) System-Defined Event Attributes*

| Attribute | Type | Description | Example |
|---|---|---|---|
| `ora_sql_txt (sql_text out ora_name_list_t)` | `PLS_INTEGER` | Returns the SQL text of the triggering statement in the `OUT` parameter. If the statement is long, it is broken into multiple collection elements. The function return value shows the number of elements are in the PL/SQL table. | `--...`<br>`-- Create table event_table`<br>`create table event_table (col VARCHAR2(2030));`<br>`--...`<br>`DECLARE`<br>`sql_text   DBMS_STANDARD.ora_name_list_t;`<br>`n   PLS_INTEGER;`<br>`v_stmt VARCHAR2(2000);`<br>`BEGIN`<br>`n := ora_sql_txt(sql_text);`<br><br>`FOR i IN 1..n LOOP`<br>`v_stmt := v_stmt \|\| sql_text(i);`<br>`END LOOP;`<br><br>`INSERT INTO event_table VALUES ('text of triggering statement: ' \|\| v_stmt);`<br>`END;` |
| `ora_sysevent` | `VARCHAR2(20)` | Database event firing the trigger: Event name is same as that in the syntax. | `INSERT INTO event_table`<br>`  VALUES (ora_sysevent);` |
| `ora_with_grant_option` | `BOOLEAN` | Returns true if the privileges are granted with grant option. | `IF (ora_sysevent = 'GRANT' and`<br>`    ora_with_grant_option = TRUE) THEN`<br>`  INSERT INTO event_table`<br>`    VALUES ('with grant option');`<br>`END IF;` |
| `space_error_info (error_number OUT NUMBER, error_type OUT VARCHAR2, object_owner OUT VARCHAR2, table_space_name OUT VARCHAR2, object_name OUT VARCHAR2, sub_object_name OUT VARCHAR2)` | `BOOLEAN` | Returns true if the error is related to an out-of-space condition, and fills in the `OUT` parameters with information about the object that caused the error. | `IF (space_error_info(eno,typ,owner,ts,obj, subobj) = TRUE) THEN`<br>`  DBMS_OUTPUT.PUT_LINE('The object '\|\| obj \|\| ' owned by ' \|\| owner \|\| ' has run out of space.');`<br>`END IF;` |

## Database Events

Database events are related to entire instances or schemas, not individual tables or rows. Triggers associated with startup and shutdown events must be defined on the database instance. Triggers associated with on-error and suspend events can be defined on either the database instance or a particular schema.

**Table 9–4 Database Events**

| Event | When Trigger Fires | Conditions | Restrictions | Transaction | Attribute Functions |
|---|---|---|---|---|---|
| STARTUP | When the database is opened. | None allowed | No database operations allowed in the trigger.<br>Return status ignored. | Starts a separate transaction and commits it after firing the triggers. | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name |
| SHUTDOWN | Just before the server starts the shutdown of an instance.<br>This lets the cartridge shutdown completely. For abnormal instance shutdown, this trigger might not fire. | None allowed | No database operations allowed in the trigger.<br>Return status ignored. | Starts a separate transaction and commits it after firing the triggers. | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name |
| DB_ROLE_CHANGE | When the database is opened for the first time after a role change. | None allowed | Return status ignored. | Starts a separate transaction and commits it after firing the triggers. | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name |
| SERVERERROR | When the error eno occurs. If no condition is given, then this trigger fires whenever an error occurs.<br>The trigger does not fire on ORA-1034, ORA-1403, ORA-1422, ORA-1423, and ORA-4030 because they are not true errors or are too serious to continue processing. It also fails to fire on ORA-18 and ORA-20 because a process is not available to connect to the database to record the error. | ERRNO = eno | Depends on the error.<br>Return status ignored. | Starts a separate transaction and commits it after firing the triggers. | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name<br>ora_server_error<br>ora_is_servererror<br>space_error_info |

## Client Events

Client events are the events related to user logon/logoff, DML, and DDL operations.

The LOGON and LOGOFF events allow simple conditions on UID and USER. All other events allow simple conditions on the type and name of the object, and functions like UID and USER.

The LOGON event starts a separate transaction and commits it after firing the triggers. All other events fire the triggers in the existing user transaction.

The LOGON and LOGOFF events can operate on any objects. For all other events, the corresponding trigger cannot perform any DDL operations, such as DROP and ALTER, on the object that caused the event to be generated.

The DDL allowed inside these triggers is altering, creating, or dropping a table, creating a trigger, and compile operations.

If an event trigger becomes the target of a DDL operation (such as CREATE TRIGGER), it cannot fire later during the same transaction

*Table 9–5 Client Events*

| Event | When Trigger Fires | Attribute Functions |
|---|---|---|
| BEFORE ALTER<br><br>AFTER ALTER | When a catalog object is altered. | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name<br>ora_dict_obj_type<br>ora_dict_obj_name<br>ora_dict_obj_owner<br>ora_des_encrypted_password<br>(for ALTER USER events)<br>ora_is_alter_column<br>(for ALTER TABLE events)<br>ora_is_drop_column<br>(for ALTER TABLE events) |
| BEFORE DROP<br><br>AFTER DROP | When a catalog object is dropped. | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name<br>ora_dict_obj_type<br>ora_dict_obj_name<br>ora_dict_obj_owner |
| BEFORE ANALYZE<br><br>AFTER ANALYZE | When an analyze statement is issued | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name<br>ora_dict_obj_name<br>ora_dict_obj_type<br>ora_dict_obj_owner |
| BEFORE ASSOCIATE STATISTICS<br><br>AFTER ASSOCIATE STATISTICS | When an associate statistics statement is issued | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name<br>ora_dict_obj_name<br>ora_dict_obj_type<br>ora_dict_obj_owner<br>ora_dict_obj_name_list<br>ora_dict_obj_owner_list |
| BEFORE AUDIT<br>AFTER AUDIT<br><br>BEFORE NOAUDIT<br>AFTER NOAUDIT | When an audit or noaudit statement is issued | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name |
| BEFORE COMMENT<br><br>AFTER COMMENT | When an object is commented | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name<br>ora_dict_obj_name<br>ora_dict_obj_type<br>ora_dict_obj_owner |
| BEFORE CREATE<br><br>AFTER CREATE | When a catalog object is created. | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name<br>ora_dict_obj_type<br>ora_dict_obj_name<br>ora_dict_obj_owner<br>ora_is_creating_nested_table<br>(for CREATE TABLE events) |

*Table 9–5 (Cont.) Client Events*

| Event | When Trigger Fires | Attribute Functions |
|---|---|---|
| `BEFORE DDL`<br><br>`AFTER DDL` | When most SQL DDL statements are issued. Not fired for `ALTER DATABASE`, `CREATE CONTROLFILE`, `CREATE DATABASE`, and DDL issued through the PL/SQL subprogram interface, such as creating an advanced queue. | `ora_sysevent`<br>`ora_login_user`<br>`ora_instance_num`<br>`ora_database_name`<br>`ora_dict_obj_name`<br>`ora_dict_obj_type`<br>`ora_dict_obj_owner` |
| `BEFORE DISASSOCIATE STATISTICS`<br><br>`AFTER DISASSOCIATE STATISTICS` | When a disassociate statistics statement is issued | `ora_sysevent`<br>`ora_login_user`<br>`ora_instance_num`<br>`ora_database_name`<br>`ora_dict_obj_name`<br>`ora_dict_obj_type`<br>`ora_dict_obj_owner`<br>`ora_dict_obj_name_list`<br>`ora_dict_obj_owner_list` |
| `BEFORE GRANT`<br><br>`AFTER GRANT` | When a grant statement is issued | `ora_sysevent`<br>`ora_login_user`<br>`ora_instance_num`<br>`ora_database_name`<br>`ora_dict_obj_name`<br>`ora_dict_obj_type`<br>`ora_dict_obj_owner`<br>`ora_grantee`<br>`ora_with_grant_option`<br>`ora_privileges` |
| `BEFORE LOGOFF` | At the start of a user logoff | `ora_sysevent`<br>`ora_login_user`<br>`ora_instance_num`<br>`ora_database_name` |
| `AFTER LOGON` | After a successful logon of a user. | `ora_sysevent`<br>`ora_login_user`<br>`ora_instance_num`<br>`ora_database_name`<br>`ora_client_ip_address` |
| `BEFORE RENAME`<br><br>`AFTER RENAME` | When a rename statement is issued. | `ora_sysevent`<br>`ora_login_user`<br>`ora_instance_num`<br>`ora_database_name`<br>`ora_dict_obj_name`<br>`ora_dict_obj_owner`<br>`ora_dict_obj_type` |

***Table 9–5 (Cont.) Client Events***

| Event | When Trigger Fires | Attribute Functions |
|---|---|---|
| BEFORE REVOKE<br><br>AFTER REVOKE | When a revoke statement is issued | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name<br>ora_dict_obj_name<br>ora_dict_obj_type<br>ora_dict_obj_owner<br>ora_revokee<br>ora_privileges |
| AFTER SUSPEND | After a SQL statement is suspended because of an out-of-space condition. The trigger must correct the condition so the statement can be resumed. | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name<br>ora_server_error<br>ora_is_servererror<br>space_error_info |
| BEFORE TRUNCATE<br><br>AFTER TRUNCATE | When an object is truncated | ora_sysevent<br>ora_login_user<br>ora_instance_num<br>ora_database_name<br>ora_dict_obj_name<br>ora_dict_obj_type<br>ora_dict_obj_owner |

# 10

# PL/SQL Packages

This chapter explains how to bundle related PL/SQL code and data into a package. A package is compiled and stored in the database, where many applications can share its contents.

Topics:


- What is a Package?
- What Goes in a Package?
- Advantages of Packages
- Serially Reusable Packages
- Package Specification
- Referencing Package Contents
- Package Body
- Examples of Package Features
- Private and Public Items in Packages
- How STANDARD Package Defines the PL/SQL Environment
- Overview of Product-Specific Packages
- Packing Writing Guidelines
- Separating Cursor Specifications and Bodies with Packages


## What is a Package?

A package is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. Packages usually have two parts, a specification ("spec") and a body; sometimes the body is unnecessary.

The specification is the interface to the package. It declares the items that can be referenced from outside the package. The body defines the queries for the cursors and the code for the subprograms.

You can think of the specification as an interface and of the body as a black box. You can debug, enhance, or replace a package body without changing the package specification.

To create a package specification, use the "CREATE PACKAGE Statement" on page 14-42. To create a package body, use the "CREATE PACKAGE BODY Statement" on page 14-45.

The specification holds public declarations, which are visible to stored subprograms and other code outside the package. You must declare subprograms in the specification after all other items (except pragmas that name a specific function; such pragmas must follow the function specification).

The body holds implementation details and private declarations, which are hidden from code outside the package. Following the declarative part of the package body is the optional **package initialization** part, which holds statements that initialize package variables and do any other one-time setup steps.

The `AUTHID` clause determines whether the subprograms and cursors in the package run with the privileges of their definer (the default) or invoker, and whether their unqualified references to schema objects are resolved in the schema of the definer or invoker. For more information, see "Invoker's Rights and Definer's Rights (AUTHID Property)" on page 8-19.

A call specification lets you map a package subprogram to a Java method or external C function. The call specification maps the Java or C name, parameter types, and return type to their SQL counterparts.

**See Also:**

- *Oracle Database Java Developer's Guide* to learn how to write Java call specifications
- *Oracle Database Advanced Application Developer's Guide* to learn how to write C call specifications
- *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL packages provided by Oracle

## What Goes in a Package?

A PL/SQL package contains:

- `Get` and `Set` methods for the package variables, to avoid letting other subprograms read and write them directly.
- Cursor declarations with the text of SQL queries. Reusing the same query text in multiple locations is faster than retyping the same query each time with slight differences. It is also easier to maintain if you must change a query that is used in many places.
- Declarations for exceptions. Typically, you must be able to reference these from different subprograms, so that you can handle exceptions in invoked subprograms.
- Declarations for subprograms that invoke each other. You need not worry about compilation order for packaged subprograms, making them more convenient than standalone stored subprograms when they invoke back and forth to each other.
- Declarations for overloaded subprograms. You can create multiple variations of a subprogram, using the same names but different sets of parameters.
- Variables to remain available between subprogram calls in the same session. You can treat variables in a package like global variables.
- Type declarations for PL/SQL collection types. To pass a collection as a parameter between stored subprograms, you must declare the type in a package so that both the invoking and invoked subprogram can refer to it.

For more information, see "CREATE PACKAGE Statement" on page 14-42. Only the declarations in the package specification are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible. You can change the body (implementation) without having to recompile invoking programs.

## Advantages of Packages

Packages have a long history in software engineering, offering important features for reliable, maintainable, reusable code, often in team development efforts for large systems.

### Modularity

Packages let you encapsulate logically related types, items, and subprograms in a named PL/SQL module. Each package is easy to understand, and the interfaces between packages are simple, clear, and well defined. This aids application development.

### Easier Application Design

When designing an application, all you need initially is the interface information in the package specs. You can code and compile a specification without its body. Then, stored subprograms that reference the package can be compiled as well. You need not define the package bodies fully until you are ready to complete the application.

### Information Hiding

With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible). For example, if a package contains four subprograms, three might be public and one private. The package hides the implementation of the private subprogram so that only the package (not your application) is affected if the implementation changes. This simplifies maintenance and enhancement. Also, by hiding implementation details from users, you protect the integrity of the package.

### Added Functionality

Packaged public variables and cursors persist for the duration of a session. They can be shared by all subprograms that run in the environment. They let you maintain data across transactions without storing it in the database.

### Better Performance

When you invoke a packaged subprogram for the first time, the whole package is loaded into memory. Later calls to related subprograms in the package require no disk I/O.

Packages stop cascading dependencies and avoid unnecessary recompiling. For example, if you change the body of a packaged function, the database does not recompile other subprograms that invoke the function; these subprograms only depend on the parameters and return value that are declared in the specification, so they are only recompiled if the specification changes.

## Serially Reusable Packages

PL/SQL packages usually consume user global area (UGA) memory corresponding to the number of package variables and cursors in the package. This limits scalability,

because the memory increases linearly with the number of users. The solution is to allow some packages to be marked as SERIALLY_REUSABLE.

For serially reusable packages, the package global memory is not kept in the UGA for each user; rather, it is kept in a small pool and reused for different users. Therefore, the global memory for such a package is only used in a unit of work. After that unit of work, the memory can be released to the pool to be reused by another user (after running the initialization code for all the global variables).

The unit of work for serially reusable packages is implicitly a call to the server; for example, an OCI call to the server, or a PL/SQL RPC call from a client to a server, or an RPC call from a server to another server.

Topics:

- Package States
- Why Serially Reusable Packages?
- Syntax of Serially Reusable Packages
- Semantics of Serially Reusable Packages
- Examples of Serially Reusable Packages

## Package States

The state of a nonreusable package (one not marked SERIALLY_REUSABLE) persists for the lifetime of a session. A package **state** includes global variables, cursors, and so on.

The state of a serially reusable package persists only for the lifetime of a call to the server. On a subsequent call to the server, if a reference is made to the serially reusable package, then Oracle Database creates an instantiation of the serially reusable package and initializes all the global variables to NULL or to the initial values provided. Any changes made to the serially reusable package state in the previous calls to the server are not visible.

> **Note:** Creating an instantiation of a serially reusable package on a call to the server does not necessarily mean that Oracle Database allocates memory or configures the instantiation object. Oracle Database looks for an available instantiation work area (which is allocated and configured) for this package in a least-recently used (LRU) pool in the SGA.
>
> After the call to the server, this work area returns to the LRU pool. The reason for keeping the pool in the SGA is that the work area can be reused across users who have requests for the same package.

## Why Serially Reusable Packages?

Because the state of a nonreusable package persists for the lifetime of the session, this locks up UGA memory for the whole session. In some applications, such as Oracle Office, a log-on session typically exists for days. Applications often need certain packages only for short periods of the session. Ideally, such applications could de-instantiate the package state in after they finish using the package (the middle of the session).

SERIALLY_REUSABLE packages enable you to design applications that manage memory better for scalability. Package states that matter only for the duration of a call to the server can be captured in SERIALLY_REUSABLE packages.

## Syntax of Serially Reusable Packages

A package can be marked serially reusable by a pragma. The syntax of the pragma is:

```
PRAGMA SERIALLY_REUSABLE;
```

A package specification can be marked serially reusable, regardless of whether it has a corresponding package body. If the package has a body, then the body must have the serially reusable pragma, if its corresponding specification has the pragma; it cannot have the serially reusable pragma unless the specification also has the pragma.

**See Also:** "SERIALLY_REUSABLE Pragma" on page 13-122

## Semantics of Serially Reusable Packages

A package that is marked SERIALLY_REUSABLE has these properties:

- Its package variables are meant for use only in the work boundaries, which correspond to calls to the server (either OCI call boundaries or PL/SQL RPC calls to the server).

> **Note:** If the application programmer makes a mistake and depends on a package variable that is set in a previous unit of work, then the application program can fail. PL/SQL cannot check for such cases.

- A pool of package instantiations is kept, and whenever a "unit of work" needs this package, one instantiation is "reused", as follows:
  - The package variables are reinitialized (for example, if the package variables have initial values, then those values are reinitialized).
  - The initialization code in the package body is run again.
- At the "end work" boundary, cleanup is done.
  - If any cursors were left open, then they are silently closed.
  - Some nonreusable secondary memory is freed (such as memory for collection variables or long VARCHAR2s).
  - This package instantiation is returned back to the pool of reusable instantiations kept for this package.
- Serially reusable packages cannot be accessed from database triggers or other PL/SQL subprograms that are invoked from SQL statements. If you try, then Oracle Database generates an error.

## Examples of Serially Reusable Packages

The two packages specified in Example 10–1 are the same, except that one is serially reusable and the other is not. Neither package has a body.

***Example 10–1 Serially Reusable Package Specification***

```
CREATE OR REPLACE PACKAGE pkg IS
```

```
  n NUMBER := 5;
END pkg;
/
CREATE OR REPLACE PACKAGE sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  n NUMBER := 5;
END sr_pkg;
/
BEGIN
  pkg.n := 10;
  sr_pkg.n := 10;
END;
/
BEGIN
  DBMS_OUTPUT.PUT_LINE('pkg.n: ' || pkg.n);
  DBMS_OUTPUT.PUT_LINE('sr_pkg.n: ' || sr_pkg.n);
END;
/
```

Result:

```
pkg.n: 10
sr_pkg.n: 5
```

The package in Example 10–2 has a body. Because the package specification is serially reusable, the package body must also be serially reusable.

***Example 10–2 Serially Reusable Package Specification and Package Body***

```
CREATE OR REPLACE PACKAGE sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  TYPE str_table_type IS TABLE OF VARCHAR2(200) INDEX BY PLS_INTEGER;
  num      NUMBER        := 10;
  str      VARCHAR2(200) := 'default-init-str';
  str_tab  STR_TABLE_TYPE;

  PROCEDURE print_pkg;

    PROCEDURE init_and_print_pkg (
      n NUMBER,
      v VARCHAR2
    );
END sr_pkg;
/
CREATE OR REPLACE PACKAGE BODY sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;

  PROCEDURE print_pkg IS
    num      NUMBER;
    str      VARCHAR2(200);
    str_tab  STR_TABLE_TYPE;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('num: ' || sr_pkg.num);
    DBMS_OUTPUT.PUT_LINE('str: ' || sr_pkg.str);
    DBMS_OUTPUT.PUT_LINE
      ('number of table elements: ' || sr_pkg.str_tab.count);
    FOR i IN 1..sr_pkg.str_tab.count LOOP
      DBMS_OUTPUT.PUT_LINE(sr_pkg.str_tab(i));
    END LOOP;
  END print_pkg;
```

```
  PROCEDURE init_and_print_pkg (
    n NUMBER,
    v VARCHAR2
  ) IS
  BEGIN
    sr_pkg.num := n;
    sr_pkg.str := v;

    FOR i IN 1..n LOOP
      sr_pkg.str_tab(i) := v || ' ' || i;
    END LOOP;

    Print_pkg;
  END  init_and_print_pkg;
END sr_pkg;
/
BEGIN
  DBMS_OUTPUT.PUT_LINE('Initializing and printing package state ...');
  sr_pkg.init_and_print_pkg(4, 'abracadabra');
  DBMS_OUTPUT.PUT_LINE('Printing package state in the same CALL ...');
  sr_pkg.print_pkg;
END;
/
```

Result:

```
Initializing and printing package state ...
num: 4
str: abracadabra
number of table elements: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4
Printing package state in the same CALL ...
num: 4
str: abracadabra
number of table elements: 4
abracadabra 1
abracadabra 2
abracadabra 3
abracadabra 4
```

Open cursors in serially reusable packages are closed automatically after a server call, and must be reopened in a new server call. A server call can be different from a subprogram call, as Example 10–3 shows.

***Example 10–3 Open Cursors in Serially Reusable Packages at Call Boundaries***

```
DROP TABLE people;
CREATE TABLE people (name VARCHAR2(20));

INSERT INTO people (name) VALUES ('John Smith');
INSERT INTO people (name) VALUES ('Mary Jones');
INSERT INTO people (name) VALUES ('Joe Brown');
INSERT INTO people (name) VALUES ('Jane White');

CREATE OR REPLACE PACKAGE sr_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  CURSOR c IS SELECT name FROM people;
END sr_pkg;
```

```
/

CREATE OR REPLACE PROCEDURE fetch_from_cursor IS
  name_  VARCHAR2(200);
BEGIN
  IF sr_pkg.c%ISOPEN THEN
    DBMS_OUTPUT.PUT_LINE('Cursor is open.');
  ELSE
    DBMS_OUTPUT.PUT_LINE('Cursor is closed; opening now.');
    OPEN sr_pkg.c;
  END IF;

  FETCH sr_pkg.c INTO name_;
  DBMS_OUTPUT.PUT_LINE('Fetched: ' || name_);

  FETCH sr_pkg.c INTO name;
    DBMS_OUTPUT.PUT_LINE('Fetched: ' || name_);
  END fetch_from_cursor;
/
```

First call to server:

```
BEGIN
  fetch_from_cursor;
  fetch_from_cursor;
END;
/
```

Result:

```
Cursor is closed; opening now.
Fetched: John Smith
Fetched: Mary Jones
Cursor is open.
Fetched: Joe Brown
Fetched: Jane White
```

New call to server:

```
BEGIN
  fetch_from_cursor;
  fetch_from_cursor;
END;
/
```

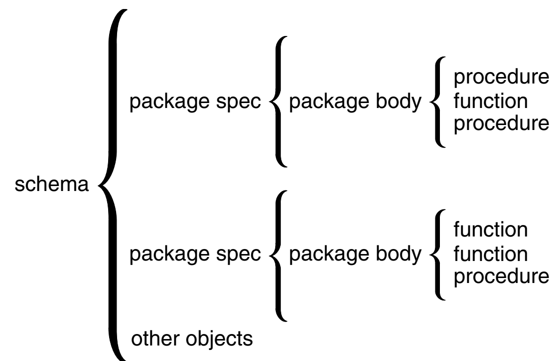
Result:

```
Cursor is closed; opening now.
Fetched: John Smith
Fetched: Mary Jones
Cursor is open.
Fetched: Joe Brown
Fetched: Jane White
```

## Package Specification

The package specification contains public declarations. The declared items are accessible from anywhere in the package and to any other subprograms in the same schema. Figure 10–1 illustrates the scoping.

***Figure 10–1 Package Scope***




The specification lists the package resources available to applications. All the information your application must use the resources is in the specification. For example, this declaration shows that the function named `factorial` takes one argument of type `INTEGER` and returns a value of type `INTEGER`:

```
FUNCTION factorial (n INTEGER) RETURN INTEGER; -- returns n!
```

That is all the information needed to invoke the function. You need not consider its underlying implementation (whether it is iterative or recursive for example).

If a specification declares only types, constants, variables, exceptions, and call specifications, the package body is unnecessary. Only subprograms and cursors have an underlying implementation. In Example 10–4, the package needs no body because it declares types, exceptions, and variables, but no subprograms or cursors. Such packages let you define global variables, usable by stored subprograms and triggers, that persist throughout a session.

***Example 10–4 Simple Package Specification Without Body***

```
CREATE OR REPLACE PACKAGE trans_data AS
  TYPE TimeRec IS RECORD (
    minutes SMALLINT,
    hours   SMALLINT);
  TYPE TransRec IS RECORD (
    category VARCHAR2(10),
    account  INT,
    amount   REAL,
    time_of  TimeRec);
  minimum_balance     CONSTANT REAL := 10.00;
  number_processed    INT;
  insufficient_funds  EXCEPTION;
END trans_data;
/
```

# Referencing Package Contents

To reference the types, items, subprograms, and call specifications declared in a package specification, use dot notation:

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
package_name.call_spec_name
```

You can reference package contents from database triggers, stored subprograms, 3GL application programs, and various Oracle tools.

### Restrictions

You cannot reference remote packaged variables, either directly or indirectly. For example, you cannot invoke a subprogram through a database link if the subprogram refers to a packaged variable.

Inside a package, you cannot reference host variables.

## Package Body

The package body contains the implementation of every cursor and subprogram declared in the package specification. Subprograms defined in a package body are accessible outside the package only if their specs also appear in the package specification. If a subprogram specification is not included in the package specification, that subprogram can only be invoked by other subprograms in the same package. A package body must be in the same schema as the package specification.

To match subprogram specs and bodies, PL/SQL does a token-by-token comparison of their headers. Except for white space, the headers must match word for word. Otherwise, PL/SQL raises an exception, as Example 10–5 shows.

***Example 10–5 Matching Package Specification and Body***

```
CREATE PACKAGE emp_bonus AS
  PROCEDURE calc_bonus (date_hired employees.hire_date%TYPE);
END emp_bonus;
/
CREATE PACKAGE BODY emp_bonus AS
  -- DATE does not match employees.hire_date%TYPE
  PROCEDURE calc_bonus (date_hired DATE) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE
      ('Employees hired on ' || date_hired || ' get bonus.');
  END;
END emp_bonus;
/
```

Result:

```
Warning: Package Body created with compilation errors.
```

Show errors (in SQL*Plus):

```
SHOW ERRORS
```

Result:

```
Errors for PACKAGE BODY EMP_BONUS:

LINE/COL ERROR
-------- -----------------------------------------------------------------
2/13     PLS-00323: subprogram or cursor 'CALC_BONUS' is declared in a
         package specification and must be defined in the package body
```

Correct problem:

```
CREATE PACKAGE BODY emp_bonus AS
  PROCEDURE calc_bonus
```

```
    (date_hired employees.hire_date%TYPE) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE
      ('Employees hired on ' || date_hired || ' get bonus.');
  END;
END emp_bonus;
/
```

Result:

```
Package body created.
```

The package body can also contain private declarations, which define types and items necessary for the internal workings of the package. The scope of these declarations is local to the package body. Therefore, the declared types and items are inaccessible except from in the package body. Unlike a package specification, the declarative part of a package body can contain subprogram bodies.

Following the declarative part of a package body is the optional initialization part, which typically holds statements that initialize some variables previously declared in the package.

The initialization part of a package plays a minor role because, unlike subprograms, a package cannot be invoked or passed parameters. As a result, the initialization part of a package is run once, the first time you reference the package.

Remember, if a package 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.

# Examples of Package Features

Consider the package `emp_admin` in Example 10–6. The package specification declares these types, items, and subprograms:

- Type `EmpRecTyp`
- Cursor `desc_salary`
- Exception `invalid_salary`
- Functions `hire_employee` and `nth_highest_salary`
- Procedures `fire_employee` and `raise_salary`

After writing the package, you can develop applications that reference its types, invoke its subprograms, use its cursor, and raise its exception. When you create the package, it is stored in the database for use by any application that has run privilege on the package.

***Example 10–6 Creating emp_admin Package***

```
-- Audit table to track changes:

DROP PACKAGE emp_admin;
CREATE TABLE emp_audit (
  date_of_action  DATE,
  user_id         VARCHAR2(20),
  package_name    VARCHAR2(30)
);

-- Package specification:
```

```
CREATE OR REPLACE PACKAGE emp_admin AS
  -- Declare externally visible type, cursor, and exception:

  TYPE EmpRecTyp IS RECORD (emp_id NUMBER, sal NUMBER);
  CURSOR desc_salary RETURN EmpRecTyp;
  invalid_salary EXCEPTION;

  -- Declare externally callable subprograms:

  FUNCTION hire_employee (
    last_name       VARCHAR2,
    first_name      VARCHAR2,
    email           VARCHAR2,
    phone_number    VARCHAR2,
    job_id          VARCHAR2,
    salary          NUMBER,
    commission_pct  NUMBER,
    manager_id      NUMBER,
    department_id   NUMBER
  ) RETURN NUMBER;

  -- Overloaded subprogram:
  PROCEDURE fire_employee (emp_id NUMBER);
  PROCEDURE fire_employee (emp_email VARCHAR2);

  PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
  FUNCTION nth_highest_salary (n NUMBER) RETURN EmpRecTyp;
END emp_admin;
/
-- Package body:

CREATE OR REPLACE PACKAGE BODY emp_admin AS
  number_hired  NUMBER;  -- visible only in this package

  -- Define cursor declared in package specification:

  CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT employee_id, salary
    FROM employees
    ORDER BY salary DESC;

  -- Define subprograms declared in package specification:

  FUNCTION hire_employee (
    last_name       VARCHAR2,
    first_name      VARCHAR2,
    email           VARCHAR2,
    phone_number    VARCHAR2,
    job_id          VARCHAR2,
    salary          NUMBER,
    commission_pct  NUMBER,
    manager_id      NUMBER,
    department_id   NUMBER
  ) RETURN NUMBER
  IS
    new_emp_id NUMBER;
  BEGIN
    new_emp_id := employees_seq.NEXTVAL;
    INSERT INTO employees (
```

```
    employee_id,
    last_name,
    first_name,
    email,
    phone_number,
    hire_date,
    job_id,
    salary,
    commission_pct,
    manager_id,
    department_id
  )
  VALUES (
    new_emp_id,
    hire_employee.last_name,
    hire_employee.first_name,
    hire_employee.email,
    hire_employee.phone_number,
    SYSDATE,
    hire_employee.job_id,
    hire_employee.salary,
    hire_employee.commission_pct,
    hire_employee.manager_id,
    hire_employee.department_id
  );
  number_hired := number_hired + 1;
  DBMS_OUTPUT.PUT_LINE('The number of employees hired is '
                       || TO_CHAR(number_hired) );
  RETURN new_emp_id;
END hire_employee;

PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
  DELETE FROM employees WHERE employee_id = emp_id;
END fire_employee;

PROCEDURE fire_employee (emp_email VARCHAR2) IS
BEGIN
  DELETE FROM employees WHERE email = emp_email;
END fire_employee;

-- Define local function, available only inside package:

FUNCTION sal_ok (
  jobid VARCHAR2,
  sal NUMBER
) RETURN BOOLEAN
IS
  min_sal NUMBER;
  max_sal NUMBER;
BEGIN
  SELECT MIN(salary), MAX(salary)
  INTO min_sal, max_sal
  FROM employees
  WHERE job_id = jobid;

  RETURN (sal >= min_sal) AND (sal <= max_sal);
END sal_ok;

PROCEDURE raise_salary (
```

```
    emp_id NUMBER,
    amount NUMBER
  )
  IS
    sal NUMBER(8,2);
    jobid VARCHAR2(10);
  BEGIN
    SELECT job_id, salary INTO jobid, sal
    FROM employees
    WHERE employee_id = emp_id;

    IF sal_ok(jobid, sal + amount) THEN
      UPDATE employees
      SET salary = salary + amount
      WHERE employee_id = emp_id;
    ELSE
      RAISE invalid_salary;
    END IF;
  EXCEPTION
    WHEN invalid_salary THEN
      DBMS_OUTPUT.PUT_LINE ('The salary is out of the specified range.');
  END raise_salary;

  FUNCTION nth_highest_salary (
    n NUMBER
  ) RETURN EmpRecTyp
  IS
    emp_rec  EmpRecTyp;
  BEGIN
    OPEN desc_salary;
    FOR i IN 1..n LOOP
      FETCH desc_salary INTO emp_rec;
    END LOOP;
    CLOSE desc_salary;
    RETURN emp_rec;
  END nth_highest_salary;

BEGIN  -- initialization part of package body
   INSERT INTO emp_audit (date_of_action, user_id, package_name)
   VALUES (SYSDATE, USER, 'EMP_ADMIN');
   number_hired := 0;
END emp_admin;
/
-- Invoking package procedures:

DECLARE
  new_emp_id NUMBER(6);
BEGIN
  new_emp_id := emp_admin.hire_employee (
    'Belden',
    'Enrique',
    'EBELDEN',
    '555.111.2222',
    'ST_CLERK',
    2500,
    .1,
    101,
    110
  );
  DBMS_OUTPUT.PUT_LINE ('The employee id is ' || TO_CHAR(new_emp_id));
```

```
  EMP_ADMIN.raise_salary (new_emp_id, 100);

  DBMS_OUTPUT.PUT_LINE (
    'The 10th highest salary is '||
    TO_CHAR (emp_admin.nth_highest_salary(10).sal) ||
             ', belonging to employee: ' ||
             TO_CHAR (emp_admin.nth_highest_salary(10).emp_id)
  );

  emp_admin.fire_employee(new_emp_id);
  -- You can also delete the newly added employee as follows:
  -- emp_admin.fire_employee('EBELDEN');
END;
/
```

Result:

```
The number of employees hired is 1
The employee id is 207
The 10th highest salary is 11730, belonging to employee: 168
There are now 107 employees.
```

Remember, the initialization part of a package is run once, the first time you reference the package. In the last example, only one row is inserted into the database table emp_audit, and the variable number_hired is initialized once.

Every time the procedure hire_employee is invoked, the variable number_hired is updated. However, the count kept by number_hired is session specific. That is, the count reflects the number of employees processed by one user, not the number processed by all users.

PL/SQL allows two or more packaged subprograms to have the same name. This option is useful when you want a subprogram to accept similar sets of parameters that have different data types. For example, the emp_admin package in Example 10–6 defines two procedures named fire_employee. The first procedure accepts a number, while the second procedure accepts string. Each procedure handles the data appropriately. For the rules that apply to overloaded subprograms, see "Overloaded Subprograms" on page 8-13.

## Private and Public Items in Packages

In the package emp_admin, the package body declares a variable named number_hired, which is initialized to zero. Items declared in the body are restricted to use in the package. PL/SQL code outside the package cannot reference the variable number_hired. Such items are called private.

Items declared in the specification of emp_admin, such as the exception invalid_salary, are visible outside the package. Any PL/SQL code can reference the exception invalid_salary. Such items are called public.

To maintain items throughout a session or across transactions, place them in the declarative part of the package body. For example, the value of number_hired is kept between calls to hire_employee in the same session. The value is lost when the session ends.

To make the items public, place them in the package specification. For example, emp_rec declared in the specification of the package is available for general use.

# How STANDARD Package Defines the PL/SQL Environment

A package named STANDARD defines the PL/SQL environment. The package specification globally declares types, exceptions, and subprograms, which are available automatically to PL/SQL programs. For example, package STANDARD declares function ABS, which returns the absolute value of its argument, as follows:

```
FUNCTION ABS (n NUMBER) RETURN NUMBER;
```

The contents of package STANDARD are directly visible to applications. You need not qualify references to its contents by prefixing the package name. For example, you might invoke ABS from a database trigger, stored subprogram, Oracle tool, or 3GL application, as follows:

```
abs_diff := ABS(x - y);
```

If you declare your own version of ABS, your local declaration overrides the global declaration. You can still invoke the built-in function by specifying its full name:

```
abs_diff := STANDARD.ABS(x - y);
```

Most built-in functions are overloaded. For example, package STANDARD contains these declarations:

```
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

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

# Overview of Product-Specific Packages

Various Oracle tools are supplied with product-specific packages that define application programming interfaces (APIs) that you can invoke from PL/SQL, SQL, Java, and other programming environments. This section briefly describes these widely used product-specific packages:

- DBMS_ALERT Package
- DBMS_OUTPUT Package
- DBMS_PIPE Package
- DBMS_CONNECTION_POOL Package
- HTF and HTP Packages
- UTL_FILE Package
- UTL_HTTP Package
- UTL_SMTP Package

For more information about these and other product-specific packages, see *Oracle Database PL/SQL Packages and Types Reference*.

## DBMS_ALERT Package

DBMS_ALERT package lets you use database triggers to alert an application when specific database values change. The alerts are transaction based and asynchronous (that is, they operate independently of any timing mechanism). For example, a

company might use this package to update the value of its investment portfolio as stock and bond quotes arrive.

## DBMS_OUTPUT Package

DBMS_OUTPUT package enables you to display output from PL/SQL blocks, subprograms, packages, and triggers. The package is especially useful for displaying PL/SQL debugging information. The procedure PUT_LINE outputs information to a buffer that can be read by another trigger, subprogram, or package. You display the information by invoking the procedure GET_LINE or by setting SERVEROUTPUT ON in SQL*Plus. Example 10–7 shows how to display output from a PL/SQL block.

***Example 10–7 DBMS_OUTPUT.PUT_LINE Procedure***

```
-- Set server output to ON to display output from DBMS_OUTPUT
SET SERVEROUTPUT ON
BEGIN
  DBMS_OUTPUT.PUT_LINE
    ('These are the tables that ' || USER || ' owns:');
  FOR item IN (SELECT table_name FROM user_tables)
    LOOP
      DBMS_OUTPUT.PUT_LINE(item.table_name);
    END LOOP;
END;
/
```

Result:

```
These are the tables that HR owns:
AUDIT_TRAIL
DEPT_TEMP
ACCOUNTS
AUDIT_TABLE_VALUES
DEPT
EMP
COMPANY_HOLIDAYS
EMP_AUDIT
COUNTRIES
DEPARTMENTS
EMPLOYEES
JOBS
REGIONS
JOB_HISTORY
LOCATIONS
```

## DBMS_PIPE Package

DBMS_PIPE package allows different sessions to communicate over named pipes. (A **pipe** is an area of memory used by one process to pass information to another.) You can use the procedures PACK_MESSAGE and SEND_MESSAGE to pack a message into a pipe, then send it to another session in the same instance or to a waiting application such as a Linux or UNIX program.

At the other end of the pipe, you can use the procedures RECEIVE_MESSAGE and UNPACK_MESSAGE to receive and unpack (read) the message. Named pipes are useful in many ways. For example, you can write a C program to collect data, then send it through pipes to stored subprograms in the database.

## DBMS_CONNECTION_POOL Package

DBMS_CONNECTION_POOL package is meant for managing the Database Resident Connection Pool, which is shared by multiple middle-tier processes. The database administrator uses procedures in DBMS_CONNECTION_POOL to start and stop the database resident connection pool and to configure pool parameters such as size and time limit.

**See Also:**

- *Oracle Database PL/SQL Packages and Types Reference* for a detailed description of the DBMS_CONNECTION_POOL package
- *Oracle Database Administrator's Guide* for information about managing the Database Resident Connection Pool

## HTF and HTP Packages

HTF and HTP packages enable your PL/SQL programs to generate HTML tags.

## UTL_FILE Package

UTL_FILE package lets PL/SQL programs read and write operating system text files. It provides a restricted version of standard operating system stream file I/O, including open, put, get, and close operations.

When you want to read or write a text file, you invoke the function FOPEN, which returns a file handle for use in subsequent subprogram calls. For example, the procedure PUT_LINE writes a text string and line terminator to an open file, and the procedure GET_LINE reads a line of text from an open file into an output buffer.

## UTL_HTTP Package

UTL_HTTP package enables your PL/SQL programs to make hypertext transfer protocol (HTTP) callouts. It can retrieve data from the Internet or invoke Oracle Web Server cartridges. The package has multiple entry points, each of which accepts a URL (uniform resource locator) string, contacts the specified site, and returns the requested data, which is usually in hypertext markup language (HTML) format.

## UTL_SMTP Package

UTL_SMTP package enables your PL/SQL programs to send electronic mails (e-mails) over Simple Mail Transfer Protocol (SMTP). The package provides interfaces to the SMTP commands for an e-mail client to dispatch e-mails to a SMTP server.

# Packing Writing Guidelines

When writing packages, keep them general so they can be reused in future applications. Become familiar with the packages that Oracle Database supplies, and avoid writing packages that duplicate their features.

Design and define package specs before the package bodies. Place in a specification only those things that must be visible to invoking programs. That way, other developers cannot build unsafe dependencies on your implementation details.

To reduce the need for recompiling when code is changed, place as few items as possible in a package specification. Changes to a package body do not require

recompiling invoking subprograms. Changes to a package specification require the database to recompile every stored subprogram that references the package.

## Separating Cursor Specifications and Bodies with Packages

You can separate a cursor specification ("spec") from its body for placement in a package. That way, you can change the cursor body without having to change the cursor specification. For information about the cursor syntax, see "Explicit Cursor" on page 13-54.

In Example 10–8, you use the %ROWTYPE attribute to provide a record type that represents a row in the database table employees.

***Example 10–8 Separating Cursor Specifications with Packages***

```
CREATE PACKAGE emp_stuff AS
  CURSOR c1 RETURN employees%ROWTYPE;
END emp_stuff;
/
CREATE PACKAGE BODY emp_stuff AS
  CURSOR c1 RETURN employees%ROWTYPE IS
    SELECT * FROM employees WHERE salary > 2500;
END emp_stuff;
/
```

The cursor specification has no SELECT statement because the RETURN clause specifies the data type of the return value. However, the cursor body must have a SELECT statement and the same RETURN clause as the cursor specification. Also, the number and data types of items in the SELECT list and the RETURN clause must match.

Packaged cursors increase flexibility. For example, you can change the cursor body in the last example, without having to change the cursor specification.

From a PL/SQL block or subprogram, you use dot notation to reference a packaged cursor, as in Example 10–9.

***Example 10–9 Referencing Packaged Cursor***

```
DECLARE
  emp_rec employees%ROWTYPE;
BEGIN
  OPEN emp_stuff.c1;
  LOOP
    FETCH emp_stuff.c1 INTO emp_rec;
    -- do processing here
    EXIT WHEN emp_stuff.c1%NOTFOUND;
  END LOOP;
  CLOSE emp_stuff.c1;
END;
/
```

The scope of a packaged cursor is not limited to a PL/SQL block. When you open a packaged cursor, it remains open until you close it or you disconnect from the session.

# 11

# PL/SQL Error Handling

This chapter explains how to handle PL/SQL compile-time warnings and PL/SQL run-time errors. The latter are called **exceptions**.

Topics:


- Compile-Time Warnings
- DBMS_WARNING Package
- Overview of Exception Handling
- Predefined Exceptions
- User-Defined Exceptions
- How Exceptions Are Raised
- How Exceptions Propagate
- Reraising an Exception
- Exception Handlers
- Unhandled Exceptions


> **Tip:** If you have problems creating or running PL/SQL code, check the Oracle Database trace files. The `USER_DUMP_DEST` initialization parameter specifies the current location of the trace files. You can find the value of this parameter by issuing `SHOW PARAMETER USER_DUMP_DEST`. For more information about trace files, see *Oracle Database Performance Tuning Guide*.

## Compile-Time Warnings

While compiling stored PL/SQL units, the PL/SQL compiler generates warnings for conditions that are not serious enough to produce errors and prevent compilation—for example, using a compiler parameter or language feature that is deprecated.

To see warnings (and errors) generated during compilation, either use the SQL*Plus command `SHOW ERRORS` or query the static data dictionary view `*_ERRORS`. For information about the view, see *Oracle Database Reference*.

The message code of a PL/SQL warning has the form PLW-*nnnnn*. For the message codes of all PL/SQL warnings, see *Oracle Database Error Messages*.

Table 11–1 summarizes the categories of warnings.

*Table 11–1 Compile-Time Warning Categories*

| Category | Description | Example |
|---|---|---|
| SEVERE | Condition might cause unexpected action or wrong results. | Aliasing problems with parameters |
| PERFORMANCE | Condition might cause performance problems. | Passing a VARCHAR2 value to a NUMBER column in an INSERT statement |
| INFORMATIONAL | Condition does not affect performance or correctness, but you might want to change it to make the code more maintainable. | Code that can never run |

By setting the compilation parameter PLSQL_WARNINGS, you can:

- Enable and disable all warnings, one or more categories of warnings, or specific warnings
- Treat specific warnings as errors (so that those conditions must be corrected before the PL/SQL unit can be compiled)

You can set the value of PLSQL_WARNINGS for:

- Your Oracle database instance

  Use the ALTER SYSTEM statement. For information about this statement, see *Oracle Database SQL Language Reference*.

- Your session

  Use the ALTER SESSION statement. For information about this statement, see *Oracle Database SQL Language Reference*.

- A stored PL/SQL unit

  Use an ALTER statement from "ALTER Statements" on page 14-1 with its *compiler_parameters_clause*. For more information about PL/SQL units and compiler parameters, see "PL/SQL Units and Compilation Parameters" on page 1-11.

In any of the preceding ALTER statements, you set the value of PLSQL_WARNINGS with this syntax:

```
PLSQL_WARNINGS = 'value_clause' [, 'value_clause' ] ...
```

For the syntax of *value_clause*, see *Oracle Database Reference*.

Example 11–1 shows several ALTER statements that set the value of PLSQL_WARNINGS.

***Example 11–1 Setting Value of PLSQL_WARNINGS Compilation Parameter***

For the session, enable all warnings—highly recommended during development:

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';
```

For the session, enable PERFORMANCE warnings:

```
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:PERFORMANCE';
```

For the procedure loc_var, enable PERFORMANCE warnings, and reuse settings:

```
ALTER PROCEDURE loc_var
```

```
  COMPILE PLSQL_WARNINGS='ENABLE:PERFORMANCE'
  REUSE SETTINGS;
```

For the session, enable SEVERE warnings, disable PERFORMANCE warnings, and treat PLW-06002 warnings as errors:

```
ALTER SESSION
  SET PLSQL_WARNINGS='ENABLE:SEVERE', 'DISABLE:PERFORMANCE', 'ERROR:06002';
```

For the session, disable all warnings:

```
ALTER SESSION SET PLSQL_WARNINGS='DISABLE:ALL';
```

To display the current value of PLSQL_WARNINGS, query the static data dictionary view ALL_PLSQL_OBJECT_SETTINGS. For information about this view, see *Oracle Database Reference*.

## DBMS_WARNING Package

If you are writing PL/SQL subprograms in a development environment that compiles them, you can control PL/SQL compile-time warning messages by invoking subprograms in the DBMS_WARNING package. You can also use this package when compiling a complex application, composed of several nested SQL*Plus scripts, where different warning settings apply to different subprograms. You can save the current state of the PLSQL_WARNINGS parameter with one call to the package, change the parameter to compile a particular set of subprograms, then restore the original parameter value.

Unreachable code could represent a mistake or be intentionally hidden by a debug flag, so you might want a warning message for it.

The procedure in Example 11–2 is unreachable.

***Example 11–2 DBMS_WARNING Package***

Disable all warning messages for this session:

```
ALTER SESSION SET PLSQL_WARNINGS='DISABLE:ALL';
```

With warnings disabled, this procedure compiles with no warnings:

```
CREATE OR REPLACE PROCEDURE unreachable_code AS
  x CONSTANT BOOLEAN := TRUE;
BEGIN
  IF x THEN
    DBMS_OUTPUT.PUT_LINE('TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE('FALSE');
  END IF;
END unreachable_code;
/
```

Enable all warning messages for this session:

```
CALL DBMS_WARNING.set_warning_setting_string ('ENABLE:ALL' ,'SESSION');
```

Check warning setting:

```
SELECT DBMS_WARNING.get_warning_setting_string() FROM DUAL;
```

Result:

```
DBMS_WARNING.GET_WARNING_SETTING_STRING()
-----------------------------------------

ENABLE:ALL

1 row selected.
```

Recompile procedure:

```
ALTER PROCEDURE unreachable_code COMPILE;
```

Result:

```
SP2-0805: Procedure altered with compilation warnings
```

Show errors:

```
SHOW ERRORS
```

Result:

```
Errors for PROCEDURE UNREACHABLE_CODE:

LINE/COL ERROR
-------- -----------------------------------------------------------------
1/1      PLW-05018: unit UNREACHABLE_CODE omitted optional AUTHID clause;
         default value DEFINER used

7/5      PLW-06002: Unreachable code
```

For more information, see DBMS_WARNING package in *Oracle Database PL/SQL Packages and Types Reference* and PLW- messages in *Oracle Database Error Messages*

## Overview of Exception Handling

Exceptions can arise from design faults, coding mistakes, hardware failures, and many other sources. You cannot anticipate all possible exceptions, but you can code exception handlers that allow your program to continue to operate in their presence.

An exception can be either internally defined (by the run-time system) or user-defined. Examples of internally defined exceptions are ORA-22056 (value *string* is divided by zero) and ORA-27102 (out of memory). Some common internal exceptions have predefined names, such as ZERO_DIVIDE and STORAGE_ERROR. The other internal exceptions can be given names.

You can define your own exceptions in the declarative part of any PL/SQL block, subprogram, or package. For example, you might define an exception named insufficient_funds to flag overdrawn bank accounts. User-defined exceptions must be given names.

When an error occurs, an exception is raised. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by RAISE statements or invocations of the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

To handle raised exceptions, you write separate exception handlers. After an exception handler runs, the current block stops running and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment. For information about managing errors when using BULK COLLECT, see "Handling FORALL Exceptions" on page 12-17.

Example 11–3 calculates a price-to-earnings ratio for a company. If the company has zero earnings, the division operation raises the predefined exception ZERO_DIVIDE, the execution of the block is interrupted, and control transfers to the exception handlers. The optional OTHERS handler handles all exceptions that the block does not name specifically.

***Example 11–3 Anonymous Block with Exception Handlers***

```
DECLARE
  stock_price   NUMBER := 9.73;
  net_earnings  NUMBER := 0;
  pe_ratio      NUMBER;
BEGIN
  pe_ratio := stock_price / net_earnings;  -- causes division-by-zero error
  DBMS_OUTPUT.PUT_LINE('Price/earnings ratio = ' || pe_ratio);
EXCEPTION
  WHEN ZERO_DIVIDE THEN  -- handles division-by-zero error
    DBMS_OUTPUT.PUT_LINE('Company had zero earnings.');
    pe_ratio := NULL;
  WHEN OTHERS THEN  -- handles all other errors
    DBMS_OUTPUT.PUT_LINE('Another kind of error occurred.');
    pe_ratio := NULL;
END;
/
```

Result:

```
Company had zero earnings.
```

Example 11–4 avoids the exception that Example 11–3 handles.

***Example 11–4 Avoiding the Exception that Example 11–3 Handles***

```
DECLARE
  stock_price   NUMBER := 9.73;
  net_earnings  NUMBER := 0;
  pe_ratio      NUMBER;
BEGIN
  pe_ratio :=
    CASE net_earnings
      WHEN 0 THEN NULL
      ELSE stock_price / net_earnings
    END;
END;
/
```

Topics:

- Advantages of Exceptions
- Guidelines for Avoiding and Handling Exceptions

## Advantages of Exceptions

With exceptions, you can reliably handle potential errors from many statements with a single exception handler, as in Example 11–5.

***Example 11–5 Managing Multiple Errors with a Single Exception Handler***

```
DECLARE
  emp_column  VARCHAR2(30) := 'last_name';
```

```
  table_name  VARCHAR2(30) := 'emp';
  temp_var    VARCHAR2(30);
BEGIN
  temp_var := emp_column;

  SELECT COLUMN_NAME INTO temp_var
  FROM USER_TAB_COLS
  WHERE TABLE_NAME = 'EMPLOYEES'
  AND COLUMN_NAME = UPPER(emp_column);

  temp_var := table_name;

  SELECT OBJECT_NAME INTO temp_var
  FROM USER_OBJECTS
  WHERE OBJECT_NAME = UPPER(table_name)
  AND OBJECT_TYPE = 'TABLE';

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('No Data found for SELECT on ' || temp_var);
END;
/
```

Instead of checking for an error at every point where it might occur, add an exception handler to your PL/SQL block. If the exception is ever raised in that block (including inside a sub-block), it is handled.

Sometimes the error is not immediately obvious, and cannot be detected until later when you perform calculations using bad data. Again, a single exception handler can trap all division-by-zero errors, bad array subscripts, and so on.

If you must check for errors at a specific spot, you can enclose a single statement or a group of statements inside its own `BEGIN` `END` block with its own exception handler. You can make the checking as general or as precise as you like.

Isolating exception handlers makes the rest of the program easier to read and understand.

## Guidelines for Avoiding and Handling Exceptions

Because reliability is crucial for database programs, use both error checking and exception handling to ensure your program can handle all possibilities:

- Add exception handlers whenever errors can occur.

  Errors are especially likely during arithmetic calculations, string manipulation, and database operations. Errors can also occur at other times, for example if a hardware failure with disk storage or memory causes a problem that has nothing to do with your code; but your code still must take corrective action.

- Add error-checking code whenever bad input data can cause an error.

  Expect that at some time, your code will be passed incorrect or null parameters, that your queries will return no rows or more rows than you expect.

  Test your code with different combinations of bad data to see what potential errors arise.

- Make your programs robust enough to work even if the database is not in the state you expect.

For example, perhaps a table you query will have columns added or deleted, or their types changed. You can avoid such problems by declaring individual variables with `%TYPE` qualifiers, and declaring records to hold query results with `%ROWTYPE` qualifiers.

- Handle named exceptions whenever possible, instead of using `WHEN OTHERS` in exception handlers.

  Learn the names and causes of the predefined exceptions. If your database operations might cause particular ORA-*n* errors, associate names with these errors so you can write handlers for them.

- Write out debugging information in your exception handlers.

  You might store such information in a separate table. If so, do it by invoking a subprogram declared with the `PRAGMA AUTONOMOUS_TRANSACTION`, so that you can commit your debugging information, even if you roll back the work that the main subprogram was doing.

- Carefully consider whether each exception handler should commit the transaction, roll it back, or let it continue.

  No matter how severe the error is, you want to leave the database in a consistent state and avoid storing any bad data.

## Predefined Exceptions

An internal exception is raised automatically if your PL/SQL program violates a database rule or exceeds a system-dependent limit. PL/SQL predefines some common ORA-*n* errors as exceptions. For example, PL/SQL raises the predefined exception `NO_DATA_FOUND` if a `SELECT INTO` statement returns no rows.

You can use the pragma `EXCEPTION_INIT` to associate exception names with other Oracle Database error codes that you can anticipate. To handle unexpected Oracle Database errors, you can use the `OTHERS` handler. In this handler, you can invoke the functions `SQLCODE` and `SQLERRM` to return the Oracle Database error code and message text. Once you know the error code, you can use it with pragma `EXCEPTION_INIT` and write a handler specifically for that error.

PL/SQL declares predefined exceptions globally in package `STANDARD`. You need not declare them yourself. You can write handlers for predefined exceptions using the names in Table 11–2.

***Table 11–2 Predefined Exceptions***

| Exception Name | ORA Error | SQLCODE | Raised When ... |
|---|---|---|---|
| ACCESS_INTO_NULL | 06530 | -6530 | A program attempts to assign values to the attributes of an uninitialized object |
| CASE_NOT_FOUND | 06592 | -6592 | None of the choices in the `WHEN` clauses of a `CASE` statement is selected, and there is no `ELSE` clause. |
| COLLECTION_IS_NULL | 06531 | -6531 | A program attempts to apply collection methods other than `EXISTS` to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray. |
| CURSOR_ALREADY_OPEN | 06511 | -6511 | A program attempts to open an open cursor. A cursor must be closed before it can be reopened. A cursor `FOR` loop automatically opens the cursor to which it refers, so your program cannot open that cursor inside the loop. |

***Table 11–2 (Cont.) Predefined Exceptions***

| Exception Name | ORA Error | SQLCODE | Raised When ... |
|---|---|---|---|
| DUP_VAL_ON_INDEX | 00001 | -1 | A program attempts to store duplicate values in a column that is constrained by a unique index. |
| INVALID_CURSOR | 01001 | -1001 | A program attempts a cursor operation that is not allowed, such as closing an unopened cursor. |
| INVALID_NUMBER | 01722 | -1722 | n a SQL statement, the conversion of a character string into a number fails because the string does not represent a valid number. (In procedural statements, VALUE_ERROR is raised.) This exception is also raised when the LIMIT-clause expression in a bulk FETCH statement does not evaluate to a positive number. |
| LOGIN_DENIED | 01017 | -1017 | A program attempts to log on to the database with an invalid user name or password. |
| NO_DATA_FOUND | 01403 | +100 | A SELECT INTO statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an associative array.<br>Because this exception is used internally by some SQL functions to signal completion, you must not rely on this exception being propagated if you raise it in a function that is invoked as part of a query. |
| NOT_LOGGED_ON | 01012 | -1012 | A program issues a database call without being connected to the database. |
| PROGRAM_ERROR | 06501 | -6501 | PL/SQL has an internal problem. |
| ROWTYPE_MISMATCH | 06504 | -6504 | The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. When an open host cursor variable is passed to a stored subprogram, the return types of the actual and formal parameters must be compatible. |
| SELF_IS_NULL | 30625 | -30625 | A program attempts to invoke a MEMBER method, but the instance of the ADT was not initialized. The built-in parameter SELF points to the object, and is always the first parameter passed to a MEMBER method. |
| STORAGE_ERROR | 06500 | -6500 | PL/SQL ran out of memory or memory was corrupted. |
| SUBSCRIPT_BEYOND_COUNT | 06533 | -6533 | A program references a nested table or varray element using an index number larger than the number of elements in the collection. |
| SUBSCRIPT_OUTSIDE_LIMIT | 06532 | -6532 | A program references a nested table or varray element using an index number (-1 for example) that is outside the legal range. |
| SYS_INVALID_ROWID | 01410 | -1410 | The conversion of a character string into a universal rowid fails because the character string does not represent a valid rowid. |
| TIMEOUT_ON_RESOURCE | 00051 | -51 | A time out occurs while the database is waiting for a resource. |

*Table 11–2 (Cont.) Predefined Exceptions*

| Exception Name | ORA Error | SQLCODE | Raised When ... |
|---|---|---|---|
| TOO_MANY_ROWS | 01422 | -1422 | A SELECT INTO statement returns multiple rows. |
| VALUE_ERROR | 06502 | -6502 | An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL stops the assignment and raises VALUE_ERROR. In procedural statements, VALUE_ERROR is raised if the conversion of a character string into a number fails. (In SQL statements, INVALID_NUMBER is raised.) |
| ZERO_DIVIDE | 01476 | -1476 | A program attempts to divide a number by zero. |

# User-Defined Exceptions

PL/SQL lets you define exceptions of your own. Unlike a predefined exception, a user-defined exception must be declared and then raised explicitly, using either a RAISE statement or the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR. The latter lets you associate an error message with the user-defined exception.

Topics:

- Exception Declarations
- Exception Scope
- Associating Exceptions with Numbers
- User-Defined Error Messages
- Redeclaring Predefined Exceptions

## Exception Declarations

Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. You declare an exception by introducing its name, followed by the keyword EXCEPTION. For example:

```
DECLARE
  past_due  EXCEPTION;
BEGIN
  NULL;
END;
/
```

Exception and variable declarations are similar. But remember, an exception is an error condition, not a data item. Unlike variables, exceptions cannot appear in assignment statements or SQL statements. However, the same scope rules apply to variables and exceptions.

## Exception Scope

You cannot declare an exception twice in the same block. You can, however, declare the same exception in two different blocks.

Exceptions declared in a block are considered local to that block and global to all its sub-blocks. Because a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions declared in a sub-block.

If you redeclare a global exception in a sub-block, the local declaration prevails. The sub-block cannot reference the global exception, unless the exception is declared in a labeled block and you qualify its name with the block label *block_label.exception_name*.

Example 11–6 shows the scope rules.

***Example 11–6 Scope of Exceptions***

```
DECLARE
  past_due  EXCEPTION;
  acct_num  NUMBER;
BEGIN
  DECLARE  -- sub-block begins
    past_due EXCEPTION;  -- this declaration prevails
    acct_num NUMBER;
    due_date DATE := SYSDATE - 1;
    todays_date DATE := SYSDATE;
  BEGIN
    IF due_date < todays_date THEN
      RAISE past_due;  -- this is not handled
    END IF;
  END;  -- sub-block ends
EXCEPTION
  -- Does not handle raised exception
  WHEN past_due THEN
    DBMS_OUTPUT.PUT_LINE ('Handling PAST_DUE exception.');
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE
      ('Could not recognize PAST_DUE_EXCEPTION in this scope.');
END;
/
```

Result:

```
Could not recognize PAST_DUE_EXCEPTION in this scope.
```

The enclosing block does not handle the raised exception because the declaration of `past_due` in the sub-block prevails. Though they share the same name, the two `past_due` exceptions are different, just as the two `acct_num` variables share the same name but are different variables. Thus, the `RAISE` statement and the `WHEN` clause refer to different exceptions. To have the enclosing block handle the raised exception, you must remove its declaration from the sub-block or define an `OTHERS` handler.

## Associating Exceptions with Numbers

To handle an error condition (typically an ORA-*n* message) that has no predefined name, you must use the `OTHERS` handler or the pragma `EXCEPTION_INIT`. A pragma is a compiler directive that is processed at compile time, not at run time.

In PL/SQL, the pragma `EXCEPTION_INIT` tells the compiler to associate an exception name with an Oracle Database error number. That lets you refer to any internal exception by name and to write a specific handler for it. When you see an error stack, or sequence of error messages, the one on top is the one that you can trap and handle.

You code the pragma `EXCEPTION_INIT` in the declarative part of a PL/SQL block, subprogram, or package using this syntax:

```
PRAGMA EXCEPTION_INIT(exception_name, -Oracle_error_number);
```

where *exception_name* is the name of a previously declared exception and the number is a negative value corresponding to an ORA-*n* error. The pragma must appear somewhere after the exception declaration in the same declarative section, as shown in Example 11–7.

***Example 11–7 PRAGMA EXCEPTION_INIT***

```
DECLARE
  deadlock_detected EXCEPTION;
  PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
  NULL; -- Some operation that causes an ORA-00060 error
EXCEPTION
  WHEN deadlock_detected THEN
    NULL; -- handle the error
END;
/
```

**See Also:** "EXCEPTION_INIT Pragma" on page 13-43

## User-Defined Error Messages

The RAISE_APPLICATION_ERROR procedure lets you issue user-defined ORA-*n* error messages from stored subprograms. That way, you can report errors to your application and avoid returning unhandled exceptions.

To invoke RAISE_APPLICATION_ERROR, use this syntax:

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

where *error_number* is a negative integer in the range -20000..-20999 and *message* is a character string of at most 2048 bytes. If the optional third parameter is TRUE, the error is placed on the stack of previous errors. If the parameter is FALSE (the default), the error replaces all previous errors.

---

**Note:** RAISE_APPLICATION_ERROR is part of package DBMS_STANDARD; therefore, you need not qualify references to it.

---

An application can invoke RAISE_APPLICATION_ERROR only from an running stored subprogram or method. When invoked, RAISE_APPLICATION_ERROR ends the subprogram and returns a user-defined error number and message to the application. The error number and message can be trapped like any Oracle Database error.

Example 11–8 invokes RAISE_APPLICATION_ERROR if the current schema owns fewer than 1000 tables.

***Example 11–8 RAISE_APPLICATION_ERROR Procedure***

```
DECLARE
  num_tables NUMBER;
BEGIN
  SELECT COUNT(*) INTO num_tables FROM USER_TABLES;
  IF num_tables < 1000 THEN
    /* Issue your own error code (ORA-20101)
       with your own error message. You need not
       qualify RAISE_APPLICATION_ERROR with
       DBMS_STANDARD */
```

```
    RAISE_APPLICATION_ERROR
      (-20101, 'Expecting at least 1000 tables');
  ELSE
    -- Do rest of processing (for nonerror case)
    NULL;
  END IF;
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-20101: Expecting at least 1000 tables
ORA-06512: at line 10
```

The invoking application gets a PL/SQL exception, which it can process using the error-reporting functions SQLCODE and SQLERRM in an OTHERS handler. Also, it can use the pragma EXCEPTION_INIT to map specific error numbers returned by RAISE_APPLICATION_ERROR to exceptions of its own, as this Pro*C example shows:

```
EXEC SQL EXECUTE
  /* Execute embedded PL/SQL block using host
     variables v_emp_id and v_amount, which were
     assigned values in the host environment. */
DECLARE
  null_salary EXCEPTION;
  /* Map error number returned by RAISE_APPLICATION_ERROR
     to user-defined exception. */
  PRAGMA EXCEPTION_INIT(null_salary, -20101);
  BEGIN
    raise_salary(:v_emp_id, :v_amount);
  EXCEPTION
    WHEN null_salary THEN
      INSERT INTO emp_audit (employee_id, ...) VALUES (:v_emp_id, ...);
  END;
END-EXEC;
```

This technique allows the invoking application to handle error conditions in specific exception handlers.

## Redeclaring Predefined Exceptions

PL/SQL declares predefined exceptions globally in package STANDARD, so you need not declare them yourself. Redeclaring predefined exceptions is error prone because your local declaration overrides the global declaration. For example, if you declare an exception named invalid_number and then PL/SQL raises the predefined exception INVALID_NUMBER internally, a handler written for INVALID_NUMBER cannot catch the internal exception. In such cases, you must use dot notation to specify the predefined exception. For example:

```
BEGIN
  NULL;
EXCEPTION
  WHEN invalid_number OR STANDARD.INVALID_NUMBER THEN
    NULL;  -- handle the error
END;
/
```

## How Exceptions Are Raised

Internal exceptions are raised implicitly by the run-time system, as are user-defined exceptions that you have associated with an Oracle Database error number using EXCEPTION_INIT. Other user-defined exceptions must be raised explicitly, with either RAISE statements or invocations of the procedure DBMS_STANDARD.RAISE_APPLICATION_ERROR.

Raise an exception in a PL/SQL block or subprogram only when an error makes it undesirable or impossible to finish processing. You can explicitly raise a given exception anywhere in the scope of that exception. In Example 11–9, you alert your PL/SQL block to a user-defined exception named out_of_stock.

***Example 11–9 Raising a User-Defined Exception***

```
DECLARE
  out_of_stock   EXCEPTION;
  number_on_hand  NUMBER := 0;
BEGIN
  IF number_on_hand < 1 THEN
    RAISE out_of_stock;
  END IF;
EXCEPTION
  WHEN out_of_stock THEN
    DBMS_OUTPUT.PUT_LINE('Out-of-stock error.');
END;
/
```

Result:

```
Out-of-stock error.
```

You can also raise a predefined exception explicitly. That way, an exception handler written for the predefined exception can process other errors, as Example 11–10 shows.

***Example 11–10 Raising a Predefined Exception***

```
DECLARE
  acct_type INTEGER := 7;
BEGIN
  IF acct_type NOT IN (1, 2, 3) THEN
    RAISE INVALID_NUMBER;  -- raise predefined exception
  END IF;
EXCEPTION
  WHEN INVALID_NUMBER THEN
     DBMS_OUTPUT.PUT_LINE
       ('HANDLING INVALID INPUT BY ROLLING BACK.');
     ROLLBACK;
END;
/
```

Result:

```
HANDLING INVALID INPUT BY ROLLING BACK.
```

## How Exceptions Propagate

When an exception is raised, if PL/SQL cannot find a handler for it in the current block or subprogram, the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to

search. If no handler is found, PL/SQL returns an unhandled exception error to the host environment.

Exceptions cannot propagate across remote subprogram calls done through database links. A PL/SQL block cannot catch an exception raised by a remote subprogram. For a workaround, see "User-Defined Error Messages" on page 11-11.

Figure 11–1, Figure 11–2, and Figure 11–3 illustrate the basic propagation rules.

***Figure 11–1 Propagation Rules: Example 1***

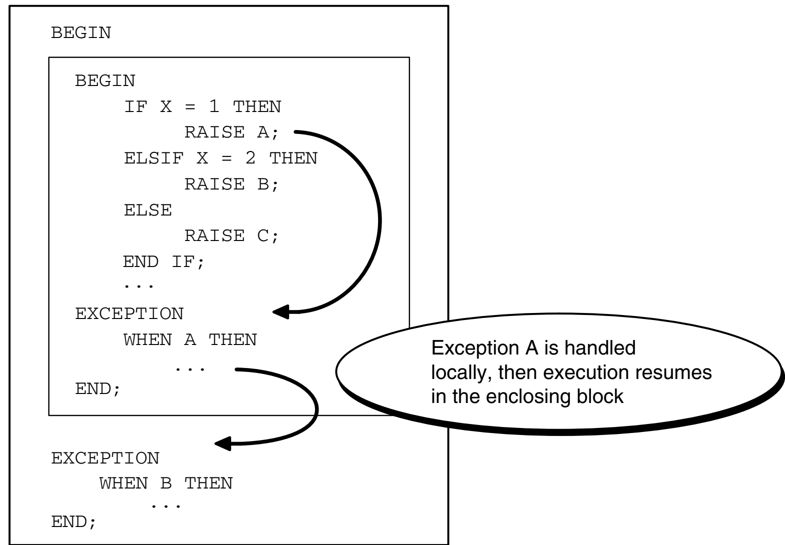



***Figure 11–2 Propagation Rules: Example 2***

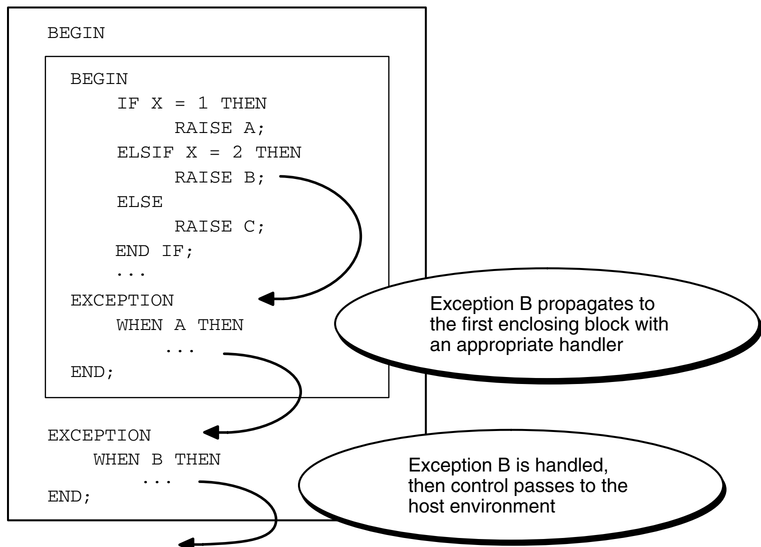
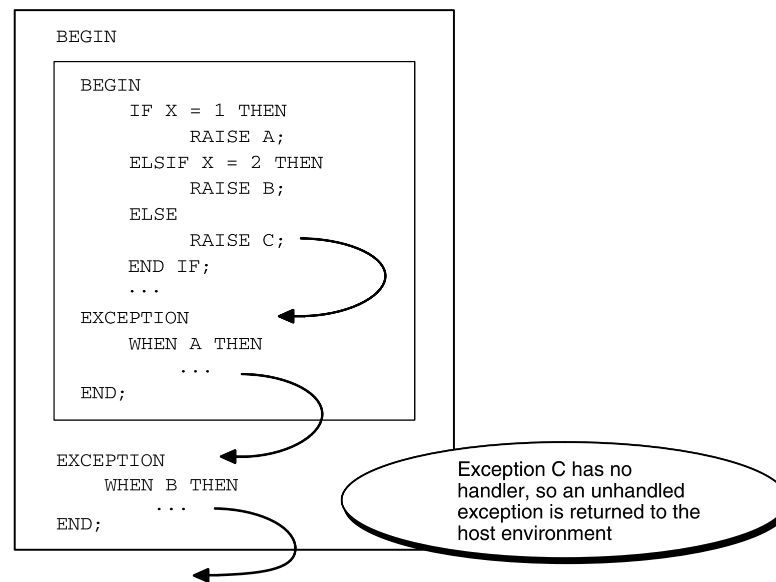


*Figure 11–3 Propagation Rules: Example 3*




An exception can propagate beyond its scope, that is, beyond the block in which it was declared, as shown in Example 11–11.

*Example 11–11 Scope of an Exception*

```
BEGIN
  DECLARE  -- sub-block begins
    past_due     EXCEPTION;
    due_date     DATE := trunc(SYSDATE) - 1;
    todays_date  DATE := trunc(SYSDATE);
  BEGIN
    IF due_date < todays_date THEN
      RAISE past_due;
    END IF;
  END;  -- sub-block ends
EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
/
```

Because the block that declares the exception `past_due` has no handler for it, the exception propagates to the enclosing block. But the enclosing block cannot reference the name `PAST_DUE`, because the scope where it was declared no longer exists. Once the exception name is lost, only an `OTHERS` handler can catch the exception. If there is no handler for a user-defined exception, the invoking application gets ORA-06510.

# Reraising an Exception

Sometimes, you want to reraise an exception, that is, handle it locally, then pass it to an enclosing block. For example, you might want to roll back a transaction in the current block, then log the error in an enclosing block.

To reraise an exception, use a `RAISE` statement without an exception name, which is allowed only in an exception handler, as in Example 11–12.

***Example 11–12 Reraising an Exception***

```
DECLARE
  salary_too_high   EXCEPTION;
  current_salary    NUMBER := 20000;
  max_salary        NUMBER := 10000;
  erroneous_salary  NUMBER;
BEGIN
  BEGIN  -- sub-block begins
    IF current_salary > max_salary THEN
      RAISE salary_too_high;  -- raise the exception
    END IF;
  EXCEPTION
    WHEN salary_too_high THEN
      -- first step in handling the error
      DBMS_OUTPUT.PUT_LINE('Salary ' || erroneous_salary ||' is out of range.');
      DBMS_OUTPUT.PUT_LINE ('Maximum salary is ' || max_salary || '.');
      RAISE;  -- reraise exception
  END;  -- sub-block ends
EXCEPTION
  WHEN salary_too_high THEN
    -- handle error more thoroughly
    erroneous_salary := current_salary;
    current_salary := max_salary;

    DBMS_OUTPUT.PUT_LINE (
      'Revising salary from ' || erroneous_salary ||
      ' to ' || current_salary || '.'
    );
END;
/
```

Result:

```
Salary  is out of range.
Maximum salary is 10000.
Revising salary from 20000 to 10000.
```

# Exception Handlers

When an exception is raised, normal execution of your PL/SQL block or subprogram stops and control transfers to its exception-handling part, which is formatted as follows:

```
EXCEPTION
  WHEN exception1 THEN -- handler for exception1
    sequence_of_statements1
  WHEN exception2 THEN -- another handler for exception2
    sequence_of_statements2
  ...
  WHEN OTHERS THEN -- optional handler for all other errors
    sequence_of_statements3
END;
```

To catch raised exceptions, you write exception handlers. Each handler consists of a `WHEN` clause, which specifies an exception, followed by a sequence of statements to run when that exception is raised. These statements complete execution of the block or subprogram; control does not return to where the exception was raised. In other words, you cannot resume processing where you left off.

The optional OTHERS exception handler, which is always the last handler in a block or subprogram, acts as the handler for all exceptions not named specifically. Thus, a block or subprogram can have only one OTHERS handler. Use of the OTHERS handler guarantees that no exception is unhandled.

If you want two or more exceptions to run the same sequence of statements, list the exception names in the WHEN clause, separating them by the keyword OR, as follows:

```
EXCEPTION
  WHEN over_limit OR under_limit OR VALUE_ERROR THEN
    -- handle the error
```

If any of the exceptions in the list is raised, the associated sequence of statements runs. The keyword OTHERS cannot appear in the list of exception names; it must appear by itself. You can have any number of exception handlers, and each handler can associate a list of exceptions with a sequence of statements. However, an exception name can appear only once in the exception-handling part of a PL/SQL block or subprogram.

The usual scoping rules for PL/SQL variables apply, so you can reference local and global variables in an exception handler. However, when an exception is raised inside a cursor FOR loop, the cursor is closed implicitly before the handler is invoked. Therefore, the values of explicit cursor attributes are *not* available in the handler.

Topics:

- Exceptions Raised in Declarations
- Exceptions Raised in Exception Handlers
- Transferring Control to or from Exception Handlers
- Error Code and Error Message Retrieval
- Continuing Execution After an Exception Is Raised
- Retrying a Transaction
- Identifying Exception Locations With Locator Variables

## Exceptions Raised in Declarations

In declarations, faulty initialization expressions can raise exceptions, but exception handlers cannot handle them, as Example 11–13 shows.

***Example 11–13 Raising an Exception in a Declaration***

```
DECLARE
  credit_limit CONSTANT NUMBER(3) := 5000;  -- Maximum value is 999
BEGIN
  NULL;
EXCEPTION
  WHEN OTHERS THEN
    -- Cannot catch exception. This handler is never invoked.
    DBMS_OUTPUT.PUT_LINE
      ('Can''t handle an exception in a declaration.');
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
```

```
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 2
```

Handlers in the current block cannot catch the raised exception because an exception raised in a declaration propagates immediately to the enclosing block.

## Exceptions Raised in Exception Handlers

When an exception occurs in an exception handler, that same handler cannot catch the exception. An exception raised inside a handler propagates immediately to the enclosing block, which is searched to find a handler for this exception. From there on, the exception propagates normally. For example:

```
EXCEPTION
  WHEN INVALID_NUMBER THEN
    INSERT INTO ... -- might raise DUP_VAL_ON_INDEX
  WHEN DUP_VAL_ON_INDEX THEN -- cannot catch exception
END;
```

## Transferring Control to or from Exception Handlers

A GOTO statement can transfer control from an exception handler into an enclosing block.

A GOTO statement cannot transfer control into an exception handler, or from an exception handler into the current block.

## Error Code and Error Message Retrieval

In an exception handler, you can retrieve the error code with the built-in function SQLCODE. To retrieve the associated error message, you can use either the packaged function DBMS_UTILTY.FORMAT_ERROR_STACK or the built-in function SQLERRM.

SQLERRM returns a maximum of 512 bytes, which is the maximum length of an Oracle Database error message (including the error code, nested messages, and message inserts, such as table and column names). DBMS_UTILTY.FORMAT_ERROR_STACK returns the full error stack, up to 2000 bytes. Therefore, DBMS_UTILTY.FORMAT_ERROR_STACK is recommended over SQLERRM, except when using the FORALL statement with its SAVE EXCEPTIONS clause. With SAVE EXCEPTIONS, use SQLERRM, as in Example 12–11 on page 12-18.

**See Also:**

- "SQLCODE Function" on page 13-124 for syntax and semantics of this function
- "SQLERRM Function" on page 13-125 for syntax and semantics of this function
- "Handling FORALL Exceptions" on page 12-17 for information about using the FORALL statement with its SAVE EXCEPTIONS clause
- *Oracle Database PL/SQL Packages and Types Reference* for information about DBMS_UTILTY.FORMAT_ERROR_STACK

A SQL statement cannot invoke SQLCODE or SQLERRM. To use their values in a SQL statement, assign them to local variables first, as in Example 11–14.

***Example 11–14 Displaying SQLCODE and SQLERRM***

```
DROP TABLE errors;
CREATE TABLE errors (
  code      NUMBER,
  message   VARCHAR2(64),
  happened  TIMESTAMP
);

DECLARE
  name    EMPLOYEES.LAST_NAME%TYPE;
  v_code  NUMBER;
  v_errm  VARCHAR2(64);
BEGIN
  SELECT last_name INTO name
  FROM EMPLOYEES
  WHERE EMPLOYEE_ID = -1;
EXCEPTION
  WHEN OTHERS THEN
    v_code := SQLCODE;
    v_errm := SUBSTR(SQLERRM, 1, 64);
    DBMS_OUTPUT.PUT_LINE
      ('Error code ' || v_code || ': ' || v_errm);

    /* Invoke another procedure,
       declared with PRAGMA AUTONOMOUS_TRANSACTION,
       to insert information about errors. */

    INSERT INTO errors (code, message, happened)
    VALUES (v_code, v_errm, SYSTIMESTAMP);
END;
/
```

Result:

```
Error code 100: ORA-01403: no data found
```

## Continuing Execution After an Exception Is Raised

An exception handler lets you recover from an otherwise irrecoverable error before exiting a block. When the handler completes, the block terminates—you cannot return to the block from the exception handler.

In Example 11–15, if the `SELECT INTO` statement raises `ZERO_DIVIDE`, execution cannot resume with the `INSERT` statement.

***Example 11–15 Exception Handler***

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS
  SELECT employee_id, salary, commission_pct
  FROM employees;

DECLARE
  sal_calc NUMBER(8,2);
BEGIN
  INSERT INTO employees_temp (employee_id, salary, commission_pct)
  VALUES (301, 2500, 0);

  SELECT (salary / commission_pct) INTO sal_calc
  FROM employees_temp
```

```
  WHERE employee_id = 301;

  INSERT INTO employees_temp VALUES (302, sal_calc/100, .1);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    NULL;
END;
/
```

You can still handle an exception for a statement, then continue with the next statement. Place the statement in its own sub-block with its own exception handlers. If an error occurs in the sub-block, a local handler can catch the exception. When the sub-block ends, the enclosing block continues to run at the point where the sub-block ends, as shown in Example 11–16.

***Example 11–16 Continuing After an Exception***

```
DECLARE
  sal_calc NUMBER(8,2);
BEGIN
  INSERT INTO employees_temp (employee_id, salary, commission_pct)
  VALUES (303, 2500, 0);

  BEGIN  -- sub-block begins
    SELECT (salary / commission_pct) INTO sal_calc
    FROM employees_temp
    WHERE employee_id = 301;
  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      sal_calc := 2500;
  END; -- sub-block ends

  INSERT INTO employees_temp (employee_id, salary, commission_pct)
  VALUES (304, sal_calc/100, .1);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    NULL;
END;
/
```

In Example 11–16, if the `SELECT INTO` statement raises a `ZERO_DIVIDE` exception, the local handler catches it and sets `sal_calc` to 2500. Execution of the handler is complete, so the sub-block terminates, and execution continues with the `INSERT` statement.

**See Also:** Example 5–38, "Collection Exceptions" on page 5-31

You can also perform a sequence of DML operations where some might fail, and process the exceptions only after the entire operation is complete, as described in "Handling FORALL Exceptions" on page 12-17.

## Retrying a Transaction

After an exception is raised, rather than abandon your transaction, you might want to retry it. The technique is:

1. Encase the transaction in a sub-block.
2. Place the sub-block inside a loop that repeats the transaction.

3. Before starting the transaction, mark a savepoint. If the transaction succeeds, commit, then exit from the loop. If the transaction fails, control transfers to the exception handler, where you roll back to the savepoint undoing any changes, then try to fix the problem.

In Example 11–17, the INSERT statement might raise an exception because of a duplicate value in a unique column. In that case, change the value that must be unique and continue with the next loop iteration. If the INSERT succeeds, exit from the loop immediately. With this technique, use a FOR or WHILE loop to limit the number of attempts.

***Example 11–17 Retrying a Transaction After an Exception***

```
DROP TABLE results;
CREATE TABLE results (
  res_name VARCHAR(20),
  res_answer VARCHAR2(3)
);

CREATE UNIQUE INDEX res_name_ix ON results (res_name);

INSERT INTO results (res_name, res_answer)
VALUES ('SMYTHE', 'YES');

INSERT INTO results (res_name, res_answer)
VALUES ('JONES', 'NO');

DECLARE
  name    VARCHAR2(20) := 'SMYTHE';
  answer  VARCHAR2(3) := 'NO';
  suffix  NUMBER := 1;
BEGIN
  FOR i IN 1..5 LOOP  -- try 5 times
    BEGIN  -- sub-block begins
       SAVEPOINT start_transaction;

       -- Remove rows from a table of survey results:
       DELETE FROM results WHERE res_answer = 'NO';

       -- Add a survey respondent's name and answers:
       INSERT INTO results  (res_name, res_answer)
       VALUES (name, answer);
       -- If two respondents have same name, raise DUP_VAL_ON_INDEX.

       COMMIT;
       EXIT;
    EXCEPTION
      WHEN DUP_VAL_ON_INDEX THEN
        ROLLBACK TO start_transaction;  -- undo changes
        suffix := suffix + 1;           -- try to fix problem
        name := name || TO_CHAR(suffix);
    END;  -- sub-block ends
  END LOOP;
END;
/
```

### Identifying Exception Locations With Locator Variables

Using one exception handler for a sequence of statements, such as INSERT, DELETE, or UPDATE statements, can mask the statement that caused an error. If you must know which statement failed, you can use a locator variable, as in Example 11–18.

***Example 11–18 Identifying Exception Locations with Locator Variables***

```
CREATE OR REPLACE PROCEDURE loc_var AS
  stmt_no  NUMBER;
  name_    VARCHAR2(100);
BEGIN
  stmt_no := 1;  -- designates 1st SELECT statement

  SELECT table_name INTO name_
  FROM user_tables
  WHERE table_name LIKE 'ABC%';

  stmt_no := 2;  -- designates 2nd SELECT statement

  SELECT table_name INTO name_
  FROM user_tables
  WHERE table_name LIKE 'XYZ%';
EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('Table name not found in query ' || stmt_no);
END;
/
CALL loc_var();
```

Result:

```
Table name not found in query 1

Call completed.
```

## Unhandled Exceptions

If it cannot find a handler for a raised exception, PL/SQL returns an unhandled exception error to the host environment, which determines the outcome. For example, in the Oracle Precompilers environment, any database changes made by a failed SQL statement or PL/SQL block are rolled back.

Unhandled exceptions can also affect subprograms. If you exit a subprogram successfully, PL/SQL assigns values to OUT parameters. However, if you exit with an unhandled exception, PL/SQL does not assign values to OUT parameters (unless they are NOCOPY parameters). Also, if a stored subprogram fails with an unhandled exception, PL/SQL does not roll back database work done by the subprogram.

You can avoid unhandled exceptions by coding an OTHERS handler at the topmost level of every PL/SQL program.

# 12

# PL/SQL Optimization and Tuning

This chapter explains how the PL/SQL compiler optimizes your code and how to write efficient PL/SQL code and speed up existing PL/SQL code.

Topics:


- PL/SQL Optimizer
- Subprogram Inlining
- PL/SQL Code to Consider Tuning
- Avoiding CPU Overhead in PL/SQL Code
- Avoiding Memory Overhead in PL/SQL Code
- Collecting Data About User-Defined Identifiers
- Profiling and Tracing PL/SQL Programs
- Reducing Loop Overhead with Bulk SQL
- Computation-Intensive PL/SQL Programs
- Tuning Dynamic SQL with EXECUTE IMMEDIATE Statement and Cursor Variables
- Tuning PL/SQL Subprogram Calls with NOCOPY Hint
- Compiling PL/SQL Units for Native Execution
- Performing Multiple Transformations with Pipelined Table Functions
- Updating Large Tables in Parallel


## PL/SQL Optimizer

Prior to Oracle Database Release 10*g*, the PL/SQL compiler translated your source code to system code without applying many changes to improve performance. Now, PL/SQL uses an optimizer that can rearrange code for better performance.

The optimizer is enabled by default. In rare cases, if the overhead of the optimizer makes compilation of very large applications too slow, you can lower the optimization by setting the compilation parameter `PLSQL_OPTIMIZE_LEVEL=1` instead of its default value 2. In even rarer cases, you might see a change in exception action, either an exception that is not raised at all, or one that is raised earlier than expected. Setting `PLSQL_OPTIMIZE_LEVEL=1` prevents the code from being rearranged.

**See Also:**

- *Oracle Database Advanced Application Developer's Guide* for information about the PL/SQL hierarchical profiler
- *Oracle Database Reference* for information about the PLSQL_OPTIMIZE_LEVEL compilation parameter
- *Oracle Database Reference* for information about the static dictionary view ALL_PLSQL_OBJECT_SETTINGS

## Subprogram Inlining

One optimization that the compiler can perform is **subprogram inlining**. Subprogram inlining replaces a subprogram call (to a subprogram in the same program unit) with a copy of the called subprogram.

To allow subprogram inlining, either accept the default value of the PLSQL_OPTIMIZE_LEVEL compilation parameter (which is 2) or set it to 3. With PLSQL_OPTIMIZE_LEVEL=2, you must specify each subprogram to be inlined. With PLSQL_OPTIMIZE_LEVEL=3, the PL/SQL compiler seeks opportunities to inline subprograms beyond those that you specify.

If a particular subprogram is inlined, performance almost always improves. However, because the compiler inlines subprograms early in the optimization process, it is possible for subprogram inlining to preclude later, more powerful optimizations.

If subprogram inlining slows the performance of a particular PL/SQL program, use the PL/SQL hierarchical profiler to identify subprograms for which you want to turn off inlining. To turn off inlining for a subprogram, use the INLINE pragma, described in "INLINE Pragma" on page 13-87.

In Example 12–1 and Example 12–2, assume that PLSQL_OPTIMIZE_LEVEL=2.

In Example 12–1, the INLINE pragma affects the procedure calls p1(1) and p1(2), but not the procedure calls p1(3) and p1(4).

***Example 12–1 Specifying that a Subprogram Is To Be Inlined***

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'YES');
x:= p1(1) + p1(2) + 17;    -- These 2 calls to p1 will be inlined
...
x:= p1(3) + p1(4) + 17;    -- These 2 calls to p1 will not be inlined
...
```

In Example 12–2 the INLINE pragma affects both functions named p2.

***Example 12–2 Specifying that an Overloaded Subprogram Is To Be Inlined***

```
FUNCTION p2 (p boolean) return PLS_INTEGER IS ...
FUNCTION p2 (x PLS_INTEGER) return PLS_INTEGER IS ...
...
PRAGMA INLINE(p2, 'YES');
x := p2(true) + p2(3);
...
```

In Example 12–3, assume that PLSQL_OPTIMIZE_LEVEL=3. The INLINE pragma affects the procedure calls p1(1) and p1(2), but not the procedure calls p1(3) and p1(4).

***Example 12–3 Specifying that a Subprogram Is Not To Be Inlined***

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'NO');
x:= p1(1) + p1(2) + 17;    -- These 2 calls to p1 will not be inlined
...
x:= p1(3) + p1(4) + 17;    -- These 2 calls to p1 might be inlined
...
```

PRAGMA INLINE ... 'NO' overrides PRAGMA INLINE ... 'YES' for the same subprogram, regardless of their order in the code. In Example 12–4, the second INLINE pragma overrides both the first and third INLINE pragmas.

***Example 12–4 Applying Two INLINE Pragmas to the Same Subprogram***

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'YES');
PRAGMA INLINE (p1, 'NO');
PRAGMA INLINE (p1, 'YES');
x:= p1(1) + p1(2) + 17;    -- These 2 calls to p1 will not be inlined
...
```

**See Also:** "INLINE Pragma" on page 13-87 for more information about subprogram inlining

## PL/SQL Code to Consider Tuning

Consider tuning the following kinds of PL/SQL code:

- Older code that does not take advantage of recent PL/SQL language features.

  Before tuning older code, benchmark the current system and measure how long particular subprograms take. With the many automatic optimizations, you might see performance improvements without doing any tuning.

- Code that does many mathematical calculations.

  Investigate the data types PLS_INTEGER, BINARY_FLOAT, and BINARY_DOUBLE.

- Functions that are called from queries, which might run millions of times.

  Make the function as efficient as possible. Consider using a function-based index to precompute the results for each row and save query time. See also "PL/SQL Function Result Cache" on page 8-29.

- Code that spends much time processing INSERT, UPDATE, or DELETE statements, or looping through query results.

  Investigate the FORALL statement for issuing DML and the BULK COLLECT clause and RETURNING BULK COLLECT clauses for queries.

- Code that spends much time processing PL/SQL statements, as opposed to issuing DDL statements like CREATE TABLE that are passed directly to SQL.

Investigate native compilation. Because many built-in database features use PL/SQL, you can apply this tuning feature to an entire database to improve performance in many areas, not just your own code.

# Avoiding CPU Overhead in PL/SQL Code

Topics:

- Make SQL Statements as Efficient as Possible
- Make Function Calls as Efficient as Possible
- Make Loops as Efficient as Possible
- Use SQL String Functions
- Put Least Expensive Conditional Tests First
- Minimize Implicit Data Type Conversion
- Avoid NUMBER Data Type and Constrained Subtypes

## Make SQL Statements as Efficient as Possible

PL/SQL programs look relatively simple because most of the work is done by SQL statements. Slow SQL statements are the main reason for slow execution.

If SQL statements are slowing down your program:

- Ensure that you have appropriate indexes. There are different kinds of indexes for different situations. Your index strategy might be different depending on the sizes of various tables in a query, the distribution of data in each query, and the columns used in the `WHERE` clauses.
- Ensure that you have up-to-date statistics on all the tables, using the subprograms in the `DBMS_STATS` package.
- Analyze the execution plans and performance of the SQL statements, using:
  - `EXPLAIN PLAN` statement
  - SQL Trace facility with `TKPROF` utility
- Rewrite the SQL statements if necessary. For example, query hints can avoid problems such as unnecessary full-table scans.

For more information about these methods, see *Oracle Database Performance Tuning Guide*.

Some PL/SQL features also help improve the performance of SQL statements:

- If you run SQL statements inside a PL/SQL loop, look at the `FORALL` statement as a way to replace loops of `INSERT`, `UPDATE`, and `DELETE` statements.
- If you are looping through the result set of a query, look at the `BULK COLLECT` clause of the `SELECT INTO` statement as a way to bring the entire result set into memory in a single operation.

## Make Function Calls as Efficient as Possible

Badly written subprograms (for example, a slow sort or search function) can harm performance. Avoid unnecessary calls to subprograms, and optimize their code:

- If a function is called in a SQL query, you can cache the function value for each row by creating a function-based index on the table in the query. The CREATE INDEX statement might take a while, but queries can be much faster.
- If a column is passed to a function in an SQL query, the query cannot use regular indexes on that column, and the function might be called for every row in a (potentially very large) table. Consider nesting the query so that the inner query filters the results to a small number of rows, and the outer query calls the function only a few times as shown in Example 12–5.

***Example 12–5 Nesting a Query to Improve Performance***

```
BEGIN
  -- Inefficient, calls function for every row:

  FOR item IN (
    SELECT DISTINCT(SQRT(department_id)) col_alias
    FROM employees
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE(item.col_alias);
  END LOOP;

  -- Efficient, calls function once for each distinct value:

  FOR item IN (
    SELECT SQRT(department_id) col_alias
    FROM (SELECT DISTINCT department_id FROM employees)
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE(item.col_alias);
  END LOOP;
END;
/
```

If you use OUT or IN OUT parameters, PL/SQL adds some performance overhead to ensure correct action in case of exceptions (assigning a value to the OUT parameter, then exiting the subprogram because of an unhandled exception, so that the OUT parameter keeps its original value).

If your program does not depend on OUT parameters keeping their values in such situations, you can add the NOCOPY keyword to the parameter declarations, so the parameters are declared OUT NOCOPY or IN OUT NOCOPY.

This technique can give significant speedup if you are passing back large amounts of data in OUT parameters, such as collections, big VARCHAR2 values, or LOBs.

This technique also applies to member methods of ADTs. If these methods modify attributes of the ADT, all the attributes are copied when the method ends. To avoid this overhead, you can explicitly declare the first parameter of the member method as SELF IN OUT NOCOPY, instead of relying on implicit declaration SELF IN OUT. For information about design considerations for object methods, see *Oracle Database Object-Relational Developer's Guide*.

## Make Loops as Efficient as Possible

Because PL/SQL applications are often built around loops, it is important to optimize both the loop itself and the code inside the loop:

- To issue a series of DML statements, replace loop constructs with FORALL statements.
- To loop through a result set and store the values, use the BULK COLLECT clause on the query to bring the query results into memory in one operation.
- If you must loop through a result set more than once, or issue other queries as you loop through a result set, you can probably enhance the original query to give you exactly the results you want. Some query operators to explore include UNION, INTERSECT, MINUS, and CONNECT BY.
- You can also nest one query inside another (known as a subquery) to do the filtering and sorting in multiple stages. For example, instead of calling a PL/SQL function in the inner WHERE clause (which might call the function once for each row of the table), you can filter the result set to a small set of rows in the inner query, and call the function in the outer query.

## Use SQL String Functions

SQL provides many highly optimized string functions, such as REPLACE, TRANSLATE, SUBSTR, INSTR, RPAD, and LTRIM. These functions use low-level code that is more efficient than PL/SQL code.

If you use PL/SQL string functions to search for regular expressions, consider using the SQL regular expression functions, such as REGEXP_SUBSTR.

- You can search for regular expressions using the SQL operator REGEXP_LIKE. See Example 6–7 on page 6-11.
- You can test or manipulate strings using the built-in functions REGEXP_INSTR, REGEXP_REPLACE, and REGEXP_SUBSTR.

Regular expression features use characters like '.', '*', '^', and '$' that you might be familiar with from Linux, UNIX, or PERL programming. For multilanguage programming, there are also extensions such as '[:lower:]' to match a lowercase letter, instead of '[a-z]' which does not match lowercase accented letters.

## Put Least Expensive Conditional Tests First

PL/SQL stops evaluating a logical expression as soon as the result can be determined. This functionality is known as short-circuit evaluation. See "Short-Circuit Evaluation" on page 2-33.

When evaluating multiple conditions separated by AND or OR, put the least expensive ones first. For example, check the values of PL/SQL variables before testing function return values, because PL/SQL might be able to skip calling the functions.

## Minimize Implicit Data Type Conversion

At run time, PL/SQL converts between different data types automatically. For example, assigning a PLS_INTEGER variable to a NUMBER variable results in a conversion because their internal representations are different.

Whenever possible, choose data types carefully to minimize implicit conversions. Use literals of the appropriate types, such as character literals in character expressions and decimal numbers in number expressions.

Minimizing conversions might mean changing the types of your variables, or even working backward and designing your tables with different data types. Or, you might convert data once, such as from an INTEGER column to a PLS_INTEGER variable, and

use the PL/SQL type consistently after that. The conversion from INTEGER to PLS_INTEGER data type might improve performance, because of the use of more efficient hardware arithmetic.

## Avoid NUMBER Data Type and Constrained Subtypes

The data type NUMBER and its subtypes are represented in a special internal format, designed for portability and arbitrary scale and precision, not for performance. Even the subtype INTEGER is treated as a floating-point number with nothing after the decimal point. Operations on NUMBER or INTEGER variables require calls to library subprograms.

Avoid constrained subtypes such as INTEGER, NATURAL, NATURALN, POSITIVE, POSITIVEN, and SIGNTYPE in performance-critical code. Variables of these types require extra checking at run time, each time they are used in a calculation.

Topics:

- Recommended Data Types for Integer Arithmetic
- Recommended Data Types for Floating-Point Arithmetic

### Recommended Data Types for Integer Arithmetic

When declaring a local integer variable:

- If the value of the variable might be NULL, or if the variable needs overflow checking, use the data type PLS_INTEGER.
- If the value of the variable can never be NULL, and the variable does not need overflow checking, use the data type SIMPLE_INTEGER.

PLS_INTEGER values use less storage space than INTEGER or NUMBER values, and PLS_INTEGER operations use hardware arithmetic. For more information, see "PLS_INTEGER and BINARY_INTEGER Data Types" on page 3-3.

SIMPLE_INTEGER is a predefined subtype of PLS_INTEGER. It has the same range as PLS_INTEGER and has a NOT NULL constraint. It differs significantly from PLS_INTEGER in its overflow semantics—for details, see "Overflow Semantics" on page 3-4.

### Recommended Data Types for Floating-Point Arithmetic

The BINARY_FLOAT and BINARY_DOUBLE types can use native hardware arithmetic instructions, and are more efficient for number-crunching applications such as scientific processing. They also require less space in the database.

If the value of the variable can never be NULL, use the subtype SIMPLE_FLOAT or SIMPLE_FLOAT instead of the base type BINARY_DOUBLE or BINARY_DOUBLE. Each subtype has the same range as its base type and has a NOT NULL constraint. Without the overhead of checking for nullness, SIMPLE_FLOAT and SIMPLE_DOUBLE provide significantly better performance than BINARY_FLOAT and BINARY_DOUBLE when PLSQL_CODE_TYPE='NATIVE', because arithmetic operations on SIMPLE_FLOAT and SIMPLE_DOUBLE values are done directly in the hardware. When PLSQL_CODE_TYPE='INTERPRETED', the performance improvement is smaller.

---

**Note:** These types are less suitable for financial code where accuracy is critical, because they do not always represent fractional values precisely, and handle rounding differently than the NUMBER types.

---

# Avoiding Memory Overhead in PL/SQL Code

Topics:

- Declare VARCHAR2 Variables of 4000 or More Characters
- Group Related Subprograms into Packages
- Pin Packages in the Shared Memory Pool
- Apply Advice of Compiler Warnings

## Declare VARCHAR2 Variables of 4000 or More Characters

You might need to allocate large `VARCHAR2` variables when you are not sure how big an expression result can be. You can conserve memory by declaring `VARCHAR2` variables with large sizes, such as 32000, rather than estimating just a little on the high side, such as by specifying 256 or 1000. PL/SQL has an optimization that makes it easy to avoid overflow problems and still conserve memory. Specify a size of more than 4000 characters for the `VARCHAR2` variable; PL/SQL waits until you assign the variable, then only allocates as much storage as needed.

## Group Related Subprograms into Packages

When you call a packaged subprogram for the first time, the whole package is loaded into the shared memory pool. Subsequent calls to related subprograms in the package require no disk I/O, and your code runs faster. If the package ages out of memory, and you reference it again, it is reloaded.

You can improve performance by sizing the shared memory pool correctly. Make it large enough to hold all frequently used packages, but not so large that memory is wasted.

## Pin Packages in the Shared Memory Pool

You can pin frequently accessed packages in the shared memory pool, using the supplied package `DBMS_SHARED_POOL`. When a package is pinned, it does not age out; it remains in memory no matter how full the pool gets or how frequently you access the package.

For more information about the `DBMS_SHARED_POOL` package, see *Oracle Database PL/SQL Packages and Types Reference*.

## Apply Advice of Compiler Warnings

The PL/SQL compiler issues warnings about things that do not make a program incorrect, but might lead to poor performance. If you receive such a warning, and the performance of this code is important, follow the suggestions in the warning and make the code more efficient.

# Collecting Data About User-Defined Identifiers

PL/Scope extracts, organizes, and stores data about user-defined identifiers from PL/SQL source code. You can retrieve source code identifier data with the static data dictionary views `*_IDENTIFIERS`. For more information, see *Oracle Database Advanced Application Developer's Guide*.

# Profiling and Tracing PL/SQL Programs

To help you isolate performance problems in large PL/SQL programs, PL/SQL provides these tools, implemented as PL/SQL packages:

| Tool | Package | Description |
|---|---|---|
| Profiler API | `DBMS_PROFILER` | Computes the time that your PL/SQL program spends at each line and in each subprogram.<br>You must have `CREATE` privileges on the units to be profiled.<br>Saves run-time statistics in database tables, which you can query. |
| Trace API | `DBMS_TRACE` | Traces the order in which subprograms run.<br>You can specify the subprograms to trace and the tracing level.<br>Saves run-time statistics in database tables, which you can query. |
| PL/SQL hierarchical profiler | `DBMS_HPROF` | Reports the dynamic execution program profile of your PL/SQL program, organized by subprogram calls. Accounts for SQL and PL/SQL execution times separately.<br>Requires no special source or compile-time preparation.<br>Generates reports in HTML. Provides the option of storing results in relational format in database tables for custom report generation (such as third-party tools offer). |

Topics:

- Profiler API: Package DBMS_PROFILER
- Trace API: Package DBMS_TRACE

For a detailed description of PL/SQL hierarchical profiler, see *Oracle Database Advanced Application Developer's Guide*.

## Profiler API: Package DBMS_PROFILER

The Profiler API ("Profiler") is implemented as PL/SQL package `DBMS_PROFILER`, whose services compute the time that your PL/SQL program spends at each line and in each subprogram and save these statistics in database tables, which you can query.

> **Note:** You can use Profiler only on units for which you have CREATE privilege. You do not need the CREATE privilege to use the PL/SQL hierarchical profiler (see *Oracle Database Advanced Application Developer's Guide*).

To use Profiler:

1. Start the profiling session.
2. Run your PL/SQL program long enough to get adequate code coverage.
3. Flush the collected data to the database.

4. Stop the profiling session.

After you have collected data with Profiler, you can:

1. Query the database tables that contain the performance data.
2. Identify the subprograms and packages that use the most execution time.
3. Determine why your program spent more time accessing certain data structures and running certain code segments.

   Inspect possible performance bottlenecks such as SQL statements, loops, and recursive functions.
4. Use the results of your analysis to replace inappropriate data structures and rework slow algorithms.

   For example, with an exponential growth in data, you might need to replace a linear search with a binary search.

For detailed information about the DBMS_PROFILER subprograms, see *Oracle Database PL/SQL Packages and Types Reference*.

### Trace API: Package DBMS_TRACE

The Trace API ("Trace") is implemented as PL/SQL package DBMS_TRACE, whose services trace execution by subprogram or exception and save these statistics in database tables, which you can query.

To use Trace:

1. (Optional) Limit tracing to specific subprograms and choose a tracing level.

   Tracing all subprograms and exceptions in a large program can produce huge amounts of data that are difficult to manage.
2. Start the tracing session.
3. Run your PL/SQL program.
4. Stop the tracing session.

After you have collected data with Trace, you can query the database tables that contain the performance data and analyze it in the same way that you analyze the performance data from Profiler (see "Profiler API: Package DBMS_PROFILER" on page 12-9).

For detailed information about the DBMS_TRACE subprograms, see *Oracle Database PL/SQL Packages and Types Reference*.

## Reducing Loop Overhead with Bulk SQL

PL/SQL sends SQL statements such as DML and queries to the SQL engine for execution, and SQL returns the results to PL/SQL. You can minimize the performance overhead of this communication between PL/SQL and SQL by using the PL/SQL features that are known collectively as **bulk SQL**.

The FORALL statement sends INSERT, UPDATE, or DELETE statements in batches, rather than one at a time. The BULK COLLECT clause brings back batches of results from SQL. If the DML statement affects four or more database rows, bulk SQL can improve performance considerably.

Assigning values to PL/SQL variables in SQL statements is called **binding**. PL/SQL binding operations fall into these categories:

| Binding Category | When This Binding Occurs |
|---|---|
| In-bind | When an INSERT or UPDATE statement stores a PL/SQL variable or host variable in the database |
| Out-bind | When the RETURNING clause of an INSERT, UPDATE, or DELETE statement assigns a database value to a PL/SQL variable or host variable |
| DEFINE | When a SELECT or FETCH statement assigns a database value to a PL/SQL variable or host variable |

Bulk SQL uses PL/SQL collections to pass large amounts of data back and forth in single operations. This process is called **bulk binding**. If the collection has *n* elements, bulk binding uses a single operation to perform the equivalent of *n* SELECT INTO, INSERT, UPDATE, or DELETE statements. A query that uses bulk binding can return any number of rows, without requiring a FETCH statement for each one.

**Note:** Parallel DML is disabled with bulk binds.

To speed up DML statements, issue them in a PL/SQL FORALL statement instead of a LOOP statement.

To speed up SELECT INTO statements, include the BULK COLLECT clause.

Topics:

- Running One DML Statement Multiple Times (FORALL Statement)
- Retrieving Query Results into Collections

## Running One DML Statement Multiple Times (FORALL Statement)

The FORALL statement lets you run multiple DML statements very efficiently. It can only repeat a single DML statement, unlike a general-purpose FOR loop. For full syntax and restrictions, see "FORALL Statement" on page 13-73.

The DML statement can reference multiple collections, but FORALL only improves performance where the index value is used as a subscript.

Usually, the bounds specify a range of consecutive index numbers. If the index numbers are not consecutive, such as after you delete collection elements, you can use the INDICES OF or VALUES OF clause to iterate over just those index values that really exist.

The INDICES OF clause iterates over all of the index values in the specified collection, or only those between a lower and upper bound.

The VALUES OF clause refers to a collection that is indexed by PLS_INTEGER and whose elements are of type PLS_INTEGER. The FORALL statement iterates over the index values specified by the elements of this collection.

The FORALL statement in Example 12–6 sends three DELETE statements to the SQL engine simultaneously.

***Example 12–6 Issuing DELETE Statements in a Loop***

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS SELECT * FROM employees;

DECLARE
```

```
  TYPE NumList IS VARRAY(20) OF NUMBER;
  depts NumList := NumList(10, 30, 70);  -- department numbers
BEGIN
  FORALL i IN depts.FIRST..depts.LAST
    DELETE FROM employees_temp
    WHERE department_id = depts(i);
  COMMIT;
END;
/
```

Example 12–7 loads some data into PL/SQL collections. Then it inserts the collection elements into a database table twice: first using a FOR loop, then using a FORALL statement. The FORALL version is much faster.

***Example 12–7 Issuing INSERT Statements in a Loop***

```
DROP TABLE parts1;
CREATE TABLE parts1 (
  pnum INTEGER,
  pname VARCHAR2(15)
);

DROP TABLE parts2;
CREATE TABLE parts2 (
  pnum INTEGER,
  pname VARCHAR2(15)
);

DECLARE
  TYPE NumTab IS TABLE OF parts1.pnum%TYPE INDEX BY PLS_INTEGER;
  TYPE NameTab IS TABLE OF parts1.pname%TYPE INDEX BY PLS_INTEGER;
  pnums   NumTab;
  pnames  NameTab;
  iterations  CONSTANT PLS_INTEGER := 500;
  t1  INTEGER;
  t2  INTEGER;
  t3  INTEGER;
BEGIN
  FOR j IN 1..iterations LOOP  -- load associative arrays
    pnums(j) := j;
    pnames(j) := 'Part No. ' || TO_CHAR(j);
  END LOOP;

  t1 := DBMS_UTILITY.get_time;

  FOR i IN 1..iterations LOOP
    INSERT INTO parts1 (pnum, pname)
    VALUES (pnums(i), pnames(i));
  END LOOP;

  t2 := DBMS_UTILITY.get_time;

  FORALL i IN 1..iterations
    INSERT INTO parts2 (pnum, pname)
    VALUES (pnums(i), pnames(i));

  t3 := DBMS_UTILITY.get_time;

  DBMS_OUTPUT.PUT_LINE('Execution Time (secs)');
  DBMS_OUTPUT.PUT_LINE('---------------------');
```

```
  DBMS_OUTPUT.PUT_LINE('FOR LOOP: ' || TO_CHAR((t2 - t1)/100));
  DBMS_OUTPUT.PUT_LINE('FORALL:   ' || TO_CHAR((t3 - t2)/100));
  COMMIT;
END;
/
```

Result:

```
Execution Time (secs)
---------------------
FOR LOOP: .03
FORALL:   .02

PL/SQL procedure successfully completed.
```

Running this block shows that the loop using FORALL is much faster.

The bounds of the FORALL loop can apply to part of a collection, not necessarily all the elements, as shown in Example 12–8.

#### *Example 12–8 FORALL Statement for Part of Collection*

```
DROP TABLE employees_temp;
CREATE TABLE employees_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS VARRAY(10) OF NUMBER;
  depts NumList := NumList(5,10,20,30,50,55,57,60,70,75);
BEGIN
  FORALL j IN 4..7  -- use only part of varray
    DELETE FROM employees_temp WHERE department_id = depts(j);
  COMMIT;
END;
/
```

You might need to delete some elements from a collection before using the collection in a FORALL statement. The INDICES OF clause processes sparse collections by iterating through only the remaining elements.

You might also want to leave the original collection alone, but process only some elements, process the elements in a different order, or process some elements more than once. Instead of copying the entire elements into collections, which might consume substantial amounts of memory, the VALUES OF clause lets you set up simple collections whose elements serve as pointers to elements in the original collection.

Example 12–9 creates a collection holding some arbitrary data, a set of table names. Deleting some elements makes it a sparse collection that does not work in a default FORALL statement. The program uses a FORALL statement with the INDICES OF clause to insert the data into a table. It then sets up two more collections, pointing to certain elements from the original collection. The program stores each set of names in a different database table using FORALL statements with the VALUES OF clause.

#### *Example 12–9 FORALL Statement for Nonconsecutive Index Values*

```
-- Create empty tables to hold order details:

DROP TABLE valid_orders;
CREATE TABLE valid_orders (
  cust_name  VARCHAR2(32),
  amount     NUMBER(10,2)
);
```

```
DROP TABLE big_orders;
CREATE TABLE big_orders AS
  SELECT * FROM valid_orders
  WHERE 1 = 0;

DROP TABLE rejected_orders;
CREATE TABLE rejected_orders AS
  SELECT * FROM valid_orders
  WHERE 1 = 0;

DECLARE
  -- Collections for set of customer names and order amounts:

  SUBTYPE cust_name IS valid_orders.cust_name%TYPE;
  TYPE cust_typ IS TABLE OF cust_name;
  cust_tab  cust_typ;

  SUBTYPE order_amount IS valid_orders.amount%TYPE;
  TYPE amount_typ IS TABLE OF NUMBER;
  amount_tab  amount_typ;

  -- Collections to point into CUST_TAB collection:

  TYPE index_pointer_t IS TABLE OF PLS_INTEGER;
  big_order_tab index_pointer_t := index_pointer_t();
  rejected_order_tab index_pointer_t := index_pointer_t();

  PROCEDURE setup_data IS
  BEGIN
    /* Set up sample order data,
       including some invalid orders and some 'big' orders. */

    cust_tab := cust_typ(
      'Company1','Company2','Company3','Company4','Company5'
    );
    amount_tab := amount_typ(5000.01, 0, 150.25, 4000.00, NULL);
  END setup_data;

BEGIN
  setup_data();
  DBMS_OUTPUT.PUT_LINE ('--- Original order data ---');

  FOR i IN 1..cust_tab.LAST LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Customer #' || i || ', ' || cust_tab(i) || ': $' || amount_tab(i)
    );
  END LOOP;

  -- Delete invalid orders (where amount is null or 0):

  FOR i IN 1..cust_tab.LAST LOOP
    IF amount_tab(i) is null or amount_tab(i) = 0 THEN
      cust_tab.delete(i);
      amount_tab.delete(i);
    END IF;
  END LOOP;
  DBMS_OUTPUT.PUT_LINE ('--- Data with invalid orders deleted ---');

  FOR i IN 1..cust_tab.LAST LOOP
```

```
    IF cust_tab.EXISTS(i) THEN
      DBMS_OUTPUT.PUT_LINE (
        'Customer #' || i || ', ' || cust_tab(i) || ': $' || amount_tab(i)
      );
    END IF;
  END LOOP;

  /* Subscripts of collections are not consecutive,
     so use FORALL...INDICES OF to iterate through actual subscripts,
     rather than using 1..COUNT. */

  FORALL i IN INDICES OF cust_tab
    INSERT INTO valid_orders (cust_name, amount)
    VALUES (cust_tab(i), amount_tab(i));

  /* Process the order data differently:
     Extract 2 subsets and store each subset in a different table.
     Reinitialize the CUST_TAB and AMOUNT_TAB collections. */

  setup_data();

  FOR i IN cust_tab.FIRST .. cust_tab.LAST LOOP
    IF amount_tab(i) IS NULL OR amount_tab(i) = 0 THEN
      rejected_order_tab.EXTEND;
      rejected_order_tab(rejected_order_tab.LAST) := i;
    END IF;
    IF amount_tab(i) > 2000 THEN
      big_order_tab.EXTEND;
      big_order_tab(big_order_tab.LAST) := i;
    END IF;
  END LOOP;

  /* Run one DML statement n one subset of elements
     and another DML statement on another subset. */

  FORALL i IN VALUES OF rejected_order_tab
    INSERT INTO rejected_orders (cust_name, amount)
    VALUES (cust_tab(i), amount_tab(i));
  FORALL i IN VALUES OF big_order_tab
    INSERT INTO big_orders (cust_name, amount)
    VALUES (cust_tab(i), amount_tab(i));
  COMMIT;
END;
/
```

Result:

```
--- Original order data ---
Customer #1, Company1: $5000.01
Customer #2, Company2: $0
Customer #3, Company3: $150.25
Customer #4, Company4: $4000
Customer #5, Company5: $
--- Data with invalid orders deleted ---
Customer #1, Company1: $5000.01
Customer #3, Company3: $150.25
Customer #4, Company4: $4000
```

Verify that correct order details were stored:

```
SELECT cust_name "Customer", amount "Valid order amount"
```

```
FROM valid_orders;
```

Result:

```
Customer                         Valid order amount
-------------------------------- ------------------
Company1                                    5000.01
Company3                                     150.25
Company4                                       4000

3 rows selected.
```

Query:

```
SELECT cust_name "Customer", amount "Big order amount"
FROM big_orders;
```

Result:

```
Customer                         Big order amount
-------------------------------- ----------------
Company1                                  5000.01
Company4                                     4000

2 rows selected.
```

Query:

```
SELECT cust_name "Customer", amount "Rejected order amount"
FROM rejected_orders;
```

Result:

```
Customer                         Rejected order amount
-------------------------------- ---------------------
Company2                                             0
Company5

2 rows selected.
```

Topics:

- Effect of FORALL Exceptions on Rollbacks
- Handling FORALL Exceptions
- Counting Rows Affected by FORALL

### Effect of FORALL Exceptions on Rollbacks

In a `FORALL` statement, if any execution of the SQL statement raises an unhandled exception, all database changes made during previous executions are rolled back. However, if a raised exception is caught and handled, changes are rolled back to an implicit savepoint marked before each execution of the SQL statement. Changes made during previous executions are not rolled back. For example, suppose you create a database table that stores department numbers and job titles, as shown in Example 12–10. Then, you change the job titles so that they are longer. The second `UPDATE` fails because the new value is too long for the column. Because you handle the exception, the first `UPDATE` is not rolled back and you can commit that change.

***Example 12–10 Rollbacks with FORALL Statement***

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp (
  deptno NUMBER(2),
  job VARCHAR2(18)
);

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts NumList := NumList(10, 20, 30);
BEGIN
  INSERT INTO emp_temp (deptno, job)
  VALUES(10, 'Clerk');

  -- Lengthening this job title raises an exception.

  INSERT INTO emp_temp (deptno, job)
  VALUES(20, 'Bookkeeper');

  INSERT INTO emp_temp  (deptno, job)
  VALUES(30, 'Analyst');

  COMMIT;

  FORALL j IN depts.FIRST..depts.LAST
    UPDATE emp_temp SET job = job || ' (Senior)'
    WHERE deptno = depts(j);
    -- raises a "value too large" exception
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ('Problem in the FORALL statement.');
    COMMIT; -- Commit results of successful updates.
END;
/
```

Result:

```
Problem in the FORALL statement.
```

### Handling FORALL Exceptions

PL/SQL provides a mechanism to handle exceptions raised during the execution of a FORALL statement. This mechanism enables a bulk-bind operation to save information about exceptions and continue processing.

To have a bulk bind complete despite errors, add the keywords SAVE EXCEPTIONS to your FORALL statement after the bounds, before the DML statement. Provide an exception handler to track the exceptions that occurred during the bulk operation.

Example 12–11 shows how you can perform several DML operations, without stopping if some operations encounter errors. In the example, EXCEPTION_INIT is used to associate the DML_ERRORS exception with the predefined error ORA-24381. ORA-24381 is raised if any exceptions are caught and saved after a bulk operation.

All exceptions raised during the execution are saved in the cursor attribute SQL%BULK_EXCEPTIONS, which stores a collection of records. Each record has two fields:

- SQL%BULK_EXCEPTIONS(i).ERROR_INDEX holds the iteration of the FORALL statement during which the exception was raised.

- SQL%BULK_EXCEPTIONS(i).ERROR_CODE holds the corresponding Oracle Database error code.

The values stored by SQL%BULK_EXCEPTIONS always refer to the most recently run FORALL statement. The number of exceptions is saved in SQL%BULK_EXCEPTIONS.COUNT. Its subscripts range from 1 to COUNT.

The individual error messages, or any substitution arguments, are not saved, but the error message text can looked up using ERROR_CODE with SQLERRM as shown in Example 12–11.

You might need to work backward to determine which collection element was used in the iteration that caused an exception. For example, if you use the INDICES OF clause to process a sparse collection, you must step through the elements one by one to find the one corresponding to SQL%BULK_EXCEPTIONS(i).ERROR_INDEX. If you use the VALUES OF clause to process a subset of elements, you must find the element in the index collection whose subscript matches SQL%BULK_EXCEPTIONS(i).ERROR_INDEX, and then use that element's value as the subscript to find the erroneous element in the original collection.

If you omit the keywords SAVE EXCEPTIONS, execution of the FORALL statement stops when an exception is raised. In that case, SQL%BULK_EXCEPTIONS.COUNT returns 1, and SQL%BULK_EXCEPTIONS contains just one record. If no exception is raised during execution, SQL%BULK_EXCEPTIONS.COUNT returns 0.

In Example 12–11, the bulk operation continues despite exceptions.

***Example 12–11 FORALL Statement and SQL%BULK_EXCEPTIONS***

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS SELECT * FROM employees;

DECLARE
  TYPE empid_tab IS TABLE OF employees.employee_id%TYPE;
  emp_sr empid_tab;

  -- Exception handler for ORA-24381:
  errors      NUMBER;
  dml_errors  EXCEPTION;
  PRAGMA EXCEPTION_INIT(dml_errors, -24381);
BEGIN
  SELECT employee_id
  BULK COLLECT INTO emp_sr FROM emp_temp
  WHERE hire_date < '30-DEC-94';

  -- Add '_SR' to job_id of most senior employees:
  FORALL i IN emp_sr.FIRST..emp_sr.LAST SAVE EXCEPTIONS
    UPDATE emp_temp SET job_id = job_id || '_SR'
    WHERE emp_sr(i) = emp_temp.employee_id;
    /* If errors occurred during FORALL SAVE EXCEPTIONS,
       a single exception is raised when the statement completes. */
EXCEPTION
  -- Figure out what failed and why
  WHEN dml_errors THEN
    errors := SQL%BULK_EXCEPTIONS.COUNT;
    DBMS_OUTPUT.PUT_LINE ('Number of statements that failed: ' || errors);

    FOR i IN 1..errors LOOP
      DBMS_OUTPUT.PUT_LINE (
        'Error #' || i || ' occurred during '||
        'iteration #' || SQL%BULK_EXCEPTIONS(i).ERROR_INDEX
```

```
      );
      DBMS_OUTPUT.PUT_LINE (
        'Error message is ' ||
        SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE)
      );
    END LOOP;
END;
/
```

In Example 12–11, PL/SQL raises predefined exceptions because updated values were too large to insert into the job_id column. After the FORALL statement, SQL%BULK_EXCEPTIONS.COUNT returned 2, and the contents of SQL%BULK_EXCEPTIONS were (7,12899) and (13,12899).

To get the Oracle Database error message (which includes the code), the value of SQL%BULK_EXCEPTIONS(i).ERROR_CODE was negated and then passed to the error-reporting function SQLERRM, which expects a negative number.

## Counting Rows Affected by FORALL

The composite attribute SQL%BULK_ROWCOUNT, used with the FORALL statement, works like an associative array. SQL%BULK_ROWCOUNT(i) stores the number of rows processed by the *i*th execution of an INSERT, UPDATE or DELETE statement, as in Example 12–12.

***Example 12–12 FORALL Statement and SQL%BULK_ROWCOUNT***

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts NumList := NumList(30, 50, 60);
BEGIN
  FORALL j IN depts.FIRST..depts.LAST
    DELETE FROM emp_temp WHERE department_id = depts(j);

  -- How many rows were affected by each DELETE statement?
  FOR i IN depts.FIRST..depts.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Iteration #' || i || ' deleted ' ||
      SQL%BULK_ROWCOUNT(i) || ' rows.'
    );
  END LOOP;
END;
/
```

Result:

```
Iteration #1 deleted 6 rows.
Iteration #2 deleted 45 rows.
Iteration #3 deleted 5 rows.
```

The FORALL statement and SQL%BULK_ROWCOUNT attribute use the same subscripts. For example, if FORALL uses the range 5..10, so does SQL%BULK_ROWCOUNT. If the FORALL statement uses the INDICES OF clause to process a sparse collection, SQL%BULK_ROWCOUNT has corresponding sparse subscripts. If the FORALL statement uses the VALUES OF clause to process a subset of elements, SQL%BULK_ROWCOUNT has subscripts corresponding to the values of the elements in the index collection. If

the index collection contains duplicate elements, so that some DML statements are issued multiple times using the same subscript, then the corresponding elements of SQL%BULK_ROWCOUNT represent the sum of all rows affected by the DML statement using that subscript.

SQL%BULK_ROWCOUNT is usually equal to 1 for inserts, because a typical insert operation affects only a single row. For the INSERT SELECT construct, SQL%BULK_ROWCOUNT might be greater than 1. For example, the FORALL statement in Example 12–13 inserts an arbitrary number of rows for each iteration. After each iteration, SQL%BULK_ROWCOUNT returns the number of items inserted.

***Example 12–13 Counting Rows Affected by FORALL with SQL%BULK_ROWCOUNT***

```
DROP TABLE emp_by_dept;
CREATE TABLE emp_by_dept AS
  SELECT employee_id, department_id
  FROM employees
  WHERE 1 = 0;

DECLARE
  TYPE dept_tab IS TABLE OF departments.department_id%TYPE;
  deptnums  dept_tab;
BEGIN
  SELECT department_id BULK COLLECT INTO deptnums FROM departments;

  FORALL i IN 1..deptnums.COUNT
    INSERT INTO emp_by_dept (employee_id, department_id)
      SELECT employee_id, department_id
      FROM employees
      WHERE department_id = deptnums(i);

  FOR i IN 1..deptnums.COUNT LOOP
    -- Count how many rows were inserted for each department; that is,
    -- how many employees are in each department.
    DBMS_OUTPUT.PUT_LINE (
      'Dept '||deptnums(i)||': inserted '||
      SQL%BULK_ROWCOUNT(i)||' records'
    );
  END LOOP;
  DBMS_OUTPUT.PUT_LINE('Total records inserted: ' || SQL%ROWCOUNT);
END;
/
```

Result:

```
Dept 10: inserted 1 records
Dept 20: inserted 2 records
Dept 30: inserted 6 records
Dept 40: inserted 1 records
Dept 50: inserted 45 records
Dept 60: inserted 5 records
Dept 70: inserted 1 records
Dept 80: inserted 34 records
Dept 90: inserted 3 records
Dept 100: inserted 6 records
Dept 110: inserted 2 records
Dept 120: inserted 0 records
Dept 130: inserted 0 records
Dept 140: inserted 0 records
Dept 150: inserted 0 records
```

```
Dept 160: inserted 0 records
Dept 170: inserted 0 records
Dept 180: inserted 0 records
Dept 190: inserted 0 records
Dept 200: inserted 0 records
Dept 210: inserted 0 records
Dept 220: inserted 0 records
Dept 230: inserted 0 records
Dept 240: inserted 0 records
Dept 250: inserted 0 records
Dept 260: inserted 0 records
Dept 270: inserted 0 records
Dept 280: inserted 0 records
Total records inserted: 106
```

You can also use the implicit cursor attributes described in "Implicit Cursors" on page 6-6 after running a `FORALL` statement.

## Retrieving Query Results into Collections

Retrieving query results directly into one or more collections in a single operation is more efficient than using loops to store one row at a time. To retrieve query results into collections, use either the `BULK COLLECT` clause (with the "SELECT INTO Statement" on page 13-118) or the `BULK COLLECT INTO` clause (with the "FETCH Statement" on page 13-67 or the "RETURNING INTO Clause" on page 13-113).

Each variable in the `INTO` list of the `BULK COLLECT` clause, or in the `BULK COLLECT INTO` clause, must be a collection. The table columns can hold scalar or composite values, including ADTs.

Example 12–14 loads two entire database columns into nested tables.

***Example 12–14 Retrieving Query Results with BULK COLLECT***

```
DECLARE
  TYPE NumTab IS TABLE OF employees.employee_id%TYPE;
  TYPE NameTab IS TABLE OF employees.last_name%TYPE;
  enums NumTab;   -- No need to initialize collections
  names NameTab;  -- Values to be filled by SELECT INTO
  PROCEDURE print_results IS
BEGIN
  IF enums.COUNT = 0 THEN
    DBMS_OUTPUT.PUT_LINE ('No results!');
  ELSE
    DBMS_OUTPUT.PUT_LINE ('Result:');
    FOR i IN enums.FIRST .. enums.LAST
    LOOP
      DBMS_OUTPUT.PUT_LINE ('  Employee #' || enums(i) || ': ' || names(i));
    END LOOP;
  END IF;
END;

BEGIN
  -- Retrieve data for employees with Ids greater than 1000:

  SELECT employee_id, last_name
  BULK COLLECT INTO enums, names
  FROM employees
  WHERE employee_id > 1000;
```

```
  /* Data was brought into memory by BULK COLLECT.
     No need to FETCH each row from result set. */

  print_results();

  -- Retrieve approximately 20% of all rows:

  SELECT employee_id, last_name
  BULK COLLECT INTO enums, names
  FROM employees SAMPLE (20);

  print_results();
END;
/
```

Result:

```
No results!
Result:
Employee #107: Lorentz
Employee #108: Greenberg
Employee #115: Khoo
Employee #118: Himuro
Employee #122: Kaufling
Employee #125: Nayer
Employee #133: Mallin
Employee #136: Philtanker
Employee #139: Seo
Employee #141: Rajs
Employee #149: Zlotkey
Employee #157: Sully
Employee #171: Smith
Employee #174: Abel
Employee #176: Taylor
Employee #180: Taylor
Employee #184: Sarchand
Employee #185: Bull
Employee #203: Mavris
```

The collections are initialized automatically. Nested tables and associative arrays are extended to hold as many elements as needed. If you use varrays, all the return values must fit in the varray's declared size. Elements are inserted starting at index 1, overwriting any existing elements.

Because the processing of the `BULK COLLECT INTO` clause is similar to a `FETCH` loop, it does not raise a `NO_DATA_FOUND` exception if no rows match the query. You must check whether the resulting nested table or varray is null, or if the resulting associative array has no elements, as shown in Example 12–15.

To prevent the resulting collections from expanding without limit, you can use the `LIMIT` clause to or pseudocolumn `ROWNUM` to limit the number of rows processed. You can also use the `SAMPLE` clause to retrieve a random sample of rows.

***Example 12–15 Limiting Query Results with Pseudocolumn ROWNUM***

```
DECLARE
  TYPE SalList IS TABLE OF employees.salary%TYPE;
  sals SalList;
BEGIN
  -- Limit number of rows to 50
```

```
  SELECT salary BULK COLLECT INTO sals
  FROM employees
  WHERE ROWNUM <= 50;

  -- Retrieve ~10% rows from table

  SELECT salary BULK COLLECT INTO sals FROM employees SAMPLE (10);
END;
/
```

You can process very large result sets by fetching a specified number of rows at a time from a cursor.

Topics:

- Examples of Bulk Fetching from a Cursor
- Limiting Rows for a Bulk FETCH Operation (LIMIT Clause)
- Retrieving DML Results Into a Collection (RETURNING INTO Clause)
- Using FORALL and BULK COLLECT Together
- Host Arrays with Bulk Binds
- SELECT BULK COLLECT INTO Statements and Aliasing

### Examples of Bulk Fetching from a Cursor

You can fetch from a cursor into one or more collections as shown in Example 12–16.

***Example 12–16 Bulk-Fetching from a Cursor Into One or More Collections***

```
DECLARE
  TYPE NameList IS TABLE OF employees.last_name%TYPE;
  TYPE SalList IS TABLE OF employees.salary%TYPE;

  CURSOR c1 IS
    SELECT last_name, salary
    FROM employees
    WHERE salary > 10000;

  names  NameList;
  sals   SalList;

  TYPE RecList IS TABLE OF c1%ROWTYPE;
  recs RecList;

  v_limit PLS_INTEGER := 10;

  PROCEDURE print_results IS
  BEGIN
    -- Check if collections are empty:

    IF names IS NULL OR names.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE('No results!');
    ELSE
      DBMS_OUTPUT.PUT_LINE('Result: ');
      FOR i IN names.FIRST .. names.LAST
      LOOP
        DBMS_OUTPUT.PUT_LINE('  Employee ' || names(i) || ': $' || sals(i));
      END LOOP;
    END IF;
```

```
  END;

BEGIN
  DBMS_OUTPUT.PUT_LINE ('--- Processing all results simultaneously ---');
  OPEN c1;
  FETCH c1 BULK COLLECT INTO names, sals;
  CLOSE c1;
  print_results();
  DBMS_OUTPUT.PUT_LINE ('--- Processing ' || v_limit || ' rows at a time ---');
  OPEN c1;
  LOOP
    FETCH c1 BULK COLLECT INTO names, sals LIMIT v_limit;
    EXIT WHEN names.COUNT = 0;
    print_results();
  END LOOP;
  CLOSE c1;
  DBMS_OUTPUT.PUT_LINE ('--- Fetching records rather than columns ---');
  OPEN c1;
  FETCH c1 BULK COLLECT INTO recs;
  FOR i IN recs.FIRST .. recs.LAST
  LOOP
    -- Now all columns from result set come from one record
    DBMS_OUTPUT.PUT_LINE (
      '  Employee ' || recs(i).last_name || ': $' || recs(i).salary
    );
  END LOOP;
END;
/
```

Result:

```
--- Processing all results simultaneously ---
Result:
Employee King: $24582
Employee Kochhar: $17340
Employee De Haan: $17340
Employee Greenberg: $12248.16
Employee Raphaely: $11220
Employee Weiss: $10088.93
Employee Russell: $14280
Employee Partners: $13770
Employee Errazuriz: $12240
Employee Cambrault: $11220
Employee Zlotkey: $10710
Employee Tucker: $10200
Employee King: $10200
Employee Vishney: $10710
Employee Ozer: $11730
Employee Bloom: $10200
Employee Abel: $11220
Employee Hartstein: $13260
Employee Baer: $10200
Employee Higgins: $12258.16
--- Processing 10 rows at a time ---
Result:
Employee King: $24582
Employee Kochhar: $17340
Employee De Haan: $17340
Employee Greenberg: $12248.16
Employee Raphaely: $11220
Employee Weiss: $10088.93
```

```
Employee Russell: $14280
Employee Partners: $13770
Employee Errazuriz: $12240
Employee Cambrault: $11220
Result:
Employee Zlotkey: $10710
Employee Tucker: $10200
Employee King: $10200
Employee Vishney: $10710
Employee Ozer: $11730
Employee Bloom: $10200
Employee Abel: $11220
Employee Hartstein: $13260
Employee Baer: $10200
Employee Higgins: $12258.16
--- Fetching records rather than columns ---
Employee King: $24582
Employee Kochhar: $17340
Employee De Haan: $17340
Employee Greenberg: $12248.16
Employee Raphaely: $11220
Employee Weiss: $10088.93
Employee Russell: $14280
Employee Partners: $13770
Employee Errazuriz: $12240
Employee Cambrault: $11220
Employee Zlotkey: $10710
Employee Tucker: $10200
Employee King: $10200
Employee Vishney: $10710
Employee Ozer: $11730
Employee Bloom: $10200
Employee Abel: $11220
Employee Hartstein: $13260
Employee Baer: $10200
Employee Higgins: $12258.16
```

Example 12–17 shows how you can fetch from a cursor into a collection of records.

***Example 12–17 Bulk-Fetching from a Cursor Into a Collection of Records***

```
DECLARE
  TYPE DeptRecTab IS TABLE OF departments%ROWTYPE;
  dept_recs  DeptRecTab;

  CURSOR c1 IS
    SELECT department_id, department_name, manager_id, location_id
    FROM departments
    WHERE department_id > 70;
BEGIN
  OPEN c1;
  FETCH c1 BULK COLLECT INTO dept_recs;
END;
/
```

### Limiting Rows for a Bulk FETCH Operation (LIMIT Clause)

The optional `LIMIT` clause, allowed only in bulk `FETCH` statements, limits the number of rows fetched from the database. In Example 12–18, with each iteration of the loop, the `FETCH` statement fetches ten rows (or fewer) into associative array `empids`. The

previous values are overwritten. Note the use of `empids.COUNT` to determine when to exit the loop.

***Example 12–18 Controlling Number of BULK COLLECT Rows with LIMIT***

```
DECLARE
  TYPE numtab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;

  CURSOR c1 IS
    SELECT employee_id
    FROM employees
    WHERE department_id = 80;

  empids  numtab;
  rows    PLS_INTEGER := 10;
BEGIN
  OPEN c1;
  LOOP  -- Fetch 10 rows or fewer in each iteration
    FETCH c1 BULK COLLECT INTO empids LIMIT rows;
    EXIT WHEN empids.COUNT = 0;  -- Not: EXIT WHEN c1%NOTFOUND
    DBMS_OUTPUT.PUT_LINE ('------- Results from Each Bulk Fetch --------');
    FOR i IN 1..empids.COUNT LOOP
      DBMS_OUTPUT.PUT_LINE ('Employee Id: ' || empids(i));
    END LOOP;
  END LOOP;
  CLOSE c1;
END;
/
```

Result:

```
------- Results from Each Bulk Fetch --------
Employee Id: 145
Employee Id: 146
Employee Id: 147
Employee Id: 148
Employee Id: 149
Employee Id: 150
Employee Id: 151
Employee Id: 152
Employee Id: 153
Employee Id: 154
------- Results from Each Bulk Fetch --------
Employee Id: 155
Employee Id: 156
Employee Id: 157
Employee Id: 158
Employee Id: 159
Employee Id: 160
Employee Id: 161
Employee Id: 162
Employee Id: 163
Employee Id: 164
------- Results from Each Bulk Fetch --------
Employee Id: 165
Employee Id: 166
Employee Id: 167
Employee Id: 168
Employee Id: 169
Employee Id: 170
Employee Id: 171
```

```
Employee Id: 172
Employee Id: 173
Employee Id: 174
------- Results from Each Bulk Fetch --------
Employee Id: 175
Employee Id: 176
Employee Id: 177
Employee Id: 179
```

### Retrieving DML Results Into a Collection (RETURNING INTO Clause)

You can use the BULK COLLECT clause in the RETURNING INTO clause of an INSERT, UPDATE, or DELETE statement.

***Example 12–19 BULK COLLECT with RETURNING INTO Clause***

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS TABLE OF employees.employee_id%TYPE;
  enums  NumList;
  TYPE NameList IS TABLE OF employees.last_name%TYPE;
  names  NameList;
BEGIN
  DELETE FROM emp_temp
  WHERE department_id = 30
  RETURNING employee_id, last_name
  BULK COLLECT INTO enums, names;

  DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');
  FOR i IN enums.FIRST .. enums.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Employee #' || enums(i) || ': ' || names(i));
  END LOOP;
END;
/
```

Result:

```
Deleted 6 rows:
Employee #114: Raphaely
Employee #115: Khoo
Employee #116: Baida
Employee #117: Tobias
Employee #118: Himuro
Employee #119: Colmenares
```

### Using FORALL and BULK COLLECT Together

You can combine the BULK COLLECT clause with a FORALL statement. The output collections are built up as the FORALL statement iterates.

In Example 12–20, the employee_id value of each deleted row is stored in the collection e_ids. The collection depts has 3 elements, so the FORALL statement iterates 3 times. If each DELETE issued by the FORALL statement deletes 5 rows, then the collection e_ids, which stores values from the deleted rows, has 15 elements when the statement completes.

***Example 12–20 FORALL with BULK COLLECT***

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS SELECT * FROM employees;

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts  NumList := NumList(10,20,30);

  TYPE enum_t IS TABLE OF employees.employee_id%TYPE;
  e_ids  enum_t;

  TYPE dept_t IS TABLE OF employees.department_id%TYPE;
  d_ids  dept_t;

BEGIN
  FORALL j IN depts.FIRST..depts.LAST
    DELETE FROM emp_temp
    WHERE department_id = depts(j)
    RETURNING employee_id, department_id
    BULK COLLECT INTO e_ids, d_ids;
  DBMS_OUTPUT.PUT_LINE ('Deleted ' || SQL%ROWCOUNT || ' rows:');
  FOR i IN e_ids.FIRST .. e_ids.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Employee #' || e_ids(i) || ' from dept #' || d_ids(i)
    );
  END LOOP;
END;
/
```

Result:

```
Deleted 9 rows:
Employee #200 from dept #10
Employee #201 from dept #20
Employee #202 from dept #20
Employee #114 from dept #30
Employee #115 from dept #30
Employee #116 from dept #30
Employee #117 from dept #30
Employee #118 from dept #30
Employee #119 from dept #30
```

The column values returned by each execution are added to the values returned previously. If you use a FOR loop instead of the FORALL statement, the set of returned values is overwritten by each DELETE statement.

You cannot use the SELECT BULK COLLECT statement in a FORALL statement.

### Host Arrays with Bulk Binds

Client-side programs can use anonymous PL/SQL blocks to bulk-bind input and output host arrays. This is the most efficient way to pass collections to and from the database server.

Host arrays are declared in a host environment such as an OCI or a Pro*C program and must be prefixed with a colon to distinguish them from PL/SQL collections. In this example, an input host array is used in a DELETE statement. At run time, the anonymous PL/SQL block is sent to the database server for execution. (Assume that values were assigned to the host array and host variables in host environment.)

```
BEGIN
  FORALL i IN :lower..:upper
    DELETE FROM employees
    WHERE department_id = :depts(i);
END;
/
```

### SELECT BULK COLLECT INTO Statements and Aliasing

In a statement of the form

```
SELECT column BULK COLLECT INTO collection FROM table ...
```

*column* and *collection* are analogous to IN and OUT NOCOPY subprogram parameters, respectively, and PL/SQL passes them by reference. As with subprogram parameters that are passed by reference, aliasing can cause unexpected results.

> **See Also:** "Subprogram Parameter Aliasing" on page 8-27

In Example 12–21, the intention is to select specific values from a collection, numbers1, and then store them in the same collection. The unexpected result is that all elements of numbers1 are deleted. For workarounds, see Example 12–22 and Example 12–23.

***Example 12–21 SELECT BULK COLLECT INTO Statement with Unexpected Results***

```
CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) IS
  numbers1  numbers_type := numbers_type(1,2,3,4,5);
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;

  --Self-selecting BULK COLLECT INTO clause:

  SELECT a.COLUMN_VALUE
  BULK COLLECT INTO numbers1
  FROM TABLE(numbers1) a
  WHERE a.COLUMN_VALUE > p.i
  ORDER BY a.COLUMN_VALUE;

  DBMS_OUTPUT.PUT_LINE('After SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
END p;
/
```

Invoke p:

```
BEGIN
  p(2);
END;
/
```

Result:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After SELECT statement
numbers1.COUNT() = 0

PL/SQL procedure successfully completed.
```

Invoke p:

```
BEGIN
  p(10);
END;
/
```

Result:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After SELECT statement
numbers1.COUNT() = 0
```

Example 12–22 uses a cursor to achieve the result intended by Example 12–21.

***Example 12–22 Cursor Workaround for Example 12–21***

```
CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) IS
  numbers1  numbers_type := numbers_type(1,2,3,4,5);

  CURSOR c IS
    SELECT a.COLUMN_VALUE
    FROM TABLE(numbers1) a
    WHERE a.COLUMN_VALUE > p.i
    ORDER BY a.COLUMN_VALUE;
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Before FETCH statement');
    DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

    FOR j IN 1..numbers1.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;

  OPEN c;
  FETCH c BULK COLLECT INTO numbers1;
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('After FETCH statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());
```

```
  IF numbers1.COUNT() > 0 THEN
    FOR j IN 1..numbers1.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;
  END IF;
END p;
/
```

Invoke p:

```
BEGIN
  p(2);
END;
/
```

Result:

```
Before FETCH statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After FETCH statement
numbers1.COUNT() = 3
numbers1(1) = 3
numbers1(2) = 4
numbers1(3) = 5
```

Invoke p:

```
BEGIN
  p(10);
END;
/
```

Result:

```
Before FETCH statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
After FETCH statement
numbers1.COUNT() = 0
```

Example 12–23 selects specific values from a collection, numbers1, and then stores them in a different collection, numbers2. Example 12–23 performs faster than Example 12–22.

***Example 12–23 Second Collection Workaround for Example 12–21***

```
CREATE OR REPLACE TYPE numbers_type IS
  TABLE OF INTEGER
/
CREATE OR REPLACE PROCEDURE p (i IN INTEGER) IS
  numbers1  numbers_type := numbers_type(1,2,3,4,5);
 numbers2  numbers_type := numbers_type(0,0,0,0,0);
```

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before SELECT statement');

  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

  FOR j IN 1..numbers1.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('numbers2.COUNT() = ' || numbers2.COUNT());

  FOR j IN 1..numbers2.COUNT() LOOP
    DBMS_OUTPUT.PUT_LINE('numbers2(' || j || ') = ' || numbers2(j));
  END LOOP;

  SELECT a.COLUMN_VALUE
  BULK COLLECT INTO numbers2      -- numbers2 appears here
  FROM TABLE(numbers1) a        -- numbers1 appears here
  WHERE a.COLUMN_VALUE > p.i
  ORDER BY a.COLUMN_VALUE;

  DBMS_OUTPUT.PUT_LINE('After SELECT statement');
  DBMS_OUTPUT.PUT_LINE('numbers1.COUNT() = ' || numbers1.COUNT());

  IF numbers1.COUNT() > 0 THEN
    FOR j IN 1..numbers1.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers1(' || j || ') = ' || numbers1(j));
    END LOOP;
  END IF;

  DBMS_OUTPUT.PUT_LINE('numbers2.COUNT() = ' || numbers2.COUNT());

  IF numbers2.COUNT() > 0 THEN
    FOR j IN 1..numbers2.COUNT() LOOP
      DBMS_OUTPUT.PUT_LINE('numbers2(' || j || ') = ' || numbers2(j));
    END LOOP;
  END IF;
END p;
/
```

Invoke p:

```
BEGIN
  p(2);
 END;
/
```

Result:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 5
numbers2(1) = 0
numbers2(2) = 0
numbers2(3) = 0
numbers2(4) = 0
```

```
numbers2(5) = 0
After SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 3
numbers2(1) = 3
numbers2(2) = 4
numbers2(3) = 5

PL/SQL procedure successfully completed.
```

Invoke `p`:

```
BEGIN
  p(10);
END;
/
```

Result:

```
Before SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 5
numbers2(1) = 0
numbers2(2) = 0
numbers2(3) = 0
numbers2(4) = 0
numbers2(5) = 0
After SELECT statement
numbers1.COUNT() = 5
numbers1(1) = 1
numbers1(2) = 2
numbers1(3) = 3
numbers1(4) = 4
numbers1(5) = 5
numbers2.COUNT() = 0
```

## Computation-Intensive PL/SQL Programs

The `BINARY_FLOAT` and `BINARY_DOUBLE` data types make it practical to write PL/SQL programs to do number-crunching, for scientific applications involving floating-point calculations. These data types act much like the native floating-point types on many hardware systems, with semantics derived from the IEEE-754 floating-point standard.

The way these data types represent decimal data make them less suitable for financial applications, where precise representation of fractional amounts is more important than pure performance.

The `PLS_INTEGER` data type is a PL/SQL-only data type that is more efficient than the SQL data types `NUMBER` or `INTEGER` for integer arithmetic. You can use `PLS_`

INTEGER to write pure PL/SQL code for integer arithmetic, or convert NUMBER or INTEGER values to PLS_INTEGER for manipulation by PL/SQL.

In a package, you can write overloaded versions of subprograms that accept different numeric parameters. The math subprograms can be optimized for each kind of parameter (BINARY_FLOAT, BINARY_DOUBLE, NUMBER, PLS_INTEGER), avoiding unnecessary conversions.

The built-in math functions (such as SQRT, SIN, COS) have fast overloaded versions that accept BINARY_FLOAT and BINARY_DOUBLE parameters. You can speed up math-intensive code by passing variables of these types to such functions, and by calling the TO_BINARY_FLOAT or TO_BINARY_DOUBLE functions when passing expressions to such functions.

## Tuning Dynamic SQL with EXECUTE IMMEDIATE Statement and Cursor Variables

Some programs (a general-purpose report writer for example) must build and process a variety of SQL statements, where the exact text of the statement is unknown until run time. Such statements probably change from execution to execution. They are called dynamic SQL statements.

Formerly, to run dynamic SQL statements, you had to use the supplied package DBMS_SQL. Now, in PL/SQL, you can run any kind of dynamic SQL statement using an interface called native dynamic SQL. The main PL/SQL features involved are the EXECUTE IMMEDIATE statement and cursor variables.

Native dynamic SQL code is more compact and much faster than calling theDBMS_SQL package. Example 12–24 declares a cursor variable and associates it with a dynamic SELECT statement.

***Example 12–24 Associating a Cursor with a Dynamic SELECT Statement***

```
DECLARE
  TYPE EmpCurTyp IS REF CURSOR;
  emp_cv      EmpCurTyp;
  v_ename     VARCHAR2(15);
  v_sal       NUMBER := 1000;
  table_name  VARCHAR2(30) := 'employees';
BEGIN
  OPEN emp_cv FOR 'SELECT last_name, salary FROM ' || table_name ||
    ' WHERE salary > :s' USING v_sal;
  CLOSE emp_cv;
END;
/
```

For more information, see Chapter 7, "PL/SQL Dynamic SQL."

## Tuning PL/SQL Subprogram Calls with NOCOPY Hint

By default, OUT and IN OUT parameters are passed by value. The values of any IN OUT parameters are copied before the subprogram runs. During subprogram execution, temporary variables hold the output parameter values. If the subprogram exits normally, these values are copied to the actual parameters. If the subprogram exits with an unhandled exception, the original parameters are unchanged.

When the parameters represent large data structures such as collections, records, and instances of ADTs, this copying slows down execution and uses up memory. In

particular, this overhead applies to each call to an object method: temporary copies are made of all the attributes, so that any changes made by the method are only applied if the method exits normally.

To avoid this overhead, you can specify the NOCOPY hint, which allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference. If the subprogram exits normally, the action is the same as normal. If the subprogram exits early with an exception, the values of OUT and IN OUT parameters (or object attributes) might still change. To use this technique, ensure that the subprogram handles all exceptions.

This example asks the compiler to pass IN OUT parameter v_staff by reference, to avoid copying the varray on entry to and exit from the subprogram:

```
DECLARE
  TYPE Staff IS VARRAY(200) OF Employee;
  PROCEDURE reorganize (v_staff IN OUT NOCOPY Staff) IS ...
```

Example 12–25 loads 25,000 records into a local nested table, which is passed to two local procedures that do nothing. A call to the subprogram that uses NOCOPY takes much less time.

#### *Example 12–25 NOCOPY with Parameters*

```
DECLARE
  TYPE EmpTabTyp IS TABLE OF employees%ROWTYPE;
  emp_tab EmpTabTyp := EmpTabTyp(NULL);  -- initialize
  t1 NUMBER;
  t2 NUMBER;
  t3 NUMBER;

  PROCEDURE get_time (t OUT NUMBER) IS
  BEGIN
    t := DBMS_UTILITY.get_time;
  END;

  PROCEDURE do_nothing1 (tab IN OUT EmpTabTyp) IS
  BEGIN
    NULL;
  END;

  PROCEDURE do_nothing2 (tab IN OUT NOCOPY EmpTabTyp) IS
  BEGIN
    NULL;
  END;

BEGIN
  SELECT * INTO emp_tab(1)
  FROM employees
  WHERE employee_id = 100;

  emp_tab.EXTEND(49999, 1);  -- Copy element 1 into 2..50000
  get_time(t1);
  do_nothing1(emp_tab);  -- Pass IN OUT parameter
  get_time(t2);
  do_nothing2(emp_tab);  -- Pass IN OUT NOCOPY parameter
  get_time(t3);
  DBMS_OUTPUT.PUT_LINE ('Call Duration (secs)');
  DBMS_OUTPUT.PUT_LINE ('--------------------');
  DBMS_OUTPUT.PUT_LINE ('Just IN OUT: ' || TO_CHAR((t2 - t1)/100.0));
  DBMS_OUTPUT.PUT_LINE ('With NOCOPY: ' || TO_CHAR((t3 - t2))/100.0);
END;
```

```
/
```

Result:

```
Call Duration (secs)
--------------------
Just IN OUT: 0
With NOCOPY: 0
```

### Restrictions on NOCOPY Hint

The use of NOCOPY increases the likelihood of parameter aliasing. For more information, see "Subprogram Parameter Aliasing" on page 8-27.

Remember, NOCOPY is a hint, not a directive. In these cases, the PL/SQL compiler ignores the NOCOPY hint and uses the by-value parameter-passing method; no error is generated:

- The actual parameter is an element of an associative array. This restriction does not apply if the parameter is an entire associative array.
- The actual parameter is constrained, such as by scale or NOT NULL. This restriction does not apply to size-constrained character strings. This restriction does not extend to constrained elements or attributes of composite types.
- The actual and formal parameters are records, one or both records were declared 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 called through a database link or as an external subprogram.

## Compiling PL/SQL Units for Native Execution

You can usually speed up PL/SQL units by compiling them into native code (processor-dependent system code), which is stored in the SYSTEM tablespace.

You can natively compile any PL/SQL unit of any type, including those that Oracle Database supplies.

Natively compiled program units work in all server environments, including shared server configuration (formerly called "multithreaded server") and Oracle Real Application Clusters (Oracle RAC).

On most platforms, PL/SQL native compilation requires no special set-up or maintenance. On some platforms, the DBA might want to do some optional configuration.

**See Also:**

- *Oracle Database Administrator's Guide* for information about configuring a database
- Platform-specific configuration documentation for your platform

You can test to see how much performance gain you can get by enabling PL/SQL native compilation.

If you have determined that PL/SQL native compilation will provide significant performance gains in database operations, Oracle recommends compiling the entire database for native mode, which requires DBA privileges. This speeds up both your own code and calls to all of the built-in PL/SQL packages.

Topics:


- Determining Whether to Use PL/SQL Native Compilation
- How PL/SQL Native Compilation Works
- Dependencies, Invalidation, and Revalidation
- Setting Up a New Database for PL/SQL Native Compilation*
- Compiling the Entire Database for PL/SQL Native or Interpreted Compilation*


* Requires DBA privileges.

## Determining Whether to Use PL/SQL Native Compilation

Whether to compile a PL/SQL unit for native or interpreted mode depends on where you are in the development cycle and on what the program unit does.

While you are debugging program units and recompiling them frequently, interpreted mode has these advantages:

- You can use PL/SQL debugging tools on program units compiled for interpreted mode (but not for those compiled for native mode).
- Compiling for interpreted mode is faster than compiling for native mode.

After the debugging phase of development, in determining whether to compile a PL/SQL unit for native mode, consider:

- PL/SQL native compilation provides the greatest performance gains for computation-intensive procedural operations. Examples are data warehouse applications and applications with extensive server-side transformations of data for display.
- PL/SQL native compilation provides the least performance gains for PL/SQL subprograms that spend most of their time running SQL.
- When many program units (typically over 15,000) are compiled for native execution, and are simultaneously active, the large amount of shared memory required might affect system performance.

## How PL/SQL Native Compilation Works

Without native compilation, the PL/SQL statements in a PL/SQL unit are compiled into an intermediate form, system code, which is stored in the catalog and interpreted at run time.

With PL/SQL native compilation, the PL/SQL statements in a PL/SQL unit are compiled into native code and stored in the catalog. The native code need not be interpreted at run time, so it runs faster.

Because native compilation applies only to PL/SQL statements, a PL/SQL unit that only calls SQL statements might not run faster when natively compiled, but it does run at least as fast as the corresponding interpreted code. The compiled code and the interpreted code make the same library calls, so their action is the same.

The first time a natively compiled PL/SQL unit runs, it is fetched from the SYSTEM tablespace into shared memory. Regardless of how many sessions call the program

unit, shared memory has only one copy it. If a program unit is not being used, the shared memory it is using might be freed, to reduce memory load.

Natively compiled subprograms and interpreted subprograms can call each other.

PL/SQL native compilation works transparently in a Oracle Real Application Clusters (Oracle RAC) environment.

The `PLSQL_CODE_TYPE` compilation parameter determines whether PL/SQL code is natively compiled or interpreted. For information about this compilation parameters, see "PL/SQL Units and Compilation Parameters" on page 1-11.

## Dependencies, Invalidation, and Revalidation

Recompilation is automatic with invalidated PL/SQL modules. For example, if an object on which a natively compiled PL/SQL subprogram depends changes, the subprogram is invalidated. The next time the same subprogram is called, the database recompiles the subprogram automatically. Because the `PLSQL_CODE_TYPE` setting is stored inside the library unit for each subprogram, the automatic recompilation uses this stored setting for code type.

Explicit recompilation does not necessarily use the stored `PLSQL_CODE_TYPE` setting. For the conditions under which explicit recompilation uses stored settings, see "PL/SQL Units and Compilation Parameters" on page 1-11.

## Setting Up a New Database for PL/SQL Native Compilation

If you have DBA privileges, you can set up a new database for PL/SQL native compilation by setting the compilation parameter `PLSQL_CODE_TYPE` to `NATIVE`. The performance benefits apply to all the built-in PL/SQL packages, which are used for many database operations.

> **Note:** If you compile the whole database as `NATIVE`, Oracle recommends that you set `PLSQL_CODE_TYPE` at the system level.

## Compiling the Entire Database for PL/SQL Native or Interpreted Compilation

If you have DBA privileges, you can recompile all PL/SQL modules in an existing database to `NATIVE` or `INTERPRETED`, using the `dbmsupgnv.sql` and `dbmsupgin.sql` scripts respectively during the process described in this section. Before making the conversion, review "Determining Whether to Use PL/SQL Native Compilation" on page 12-37.

> **Note:** If you compile the whole database as `NATIVE`, Oracle recommends that you set `PLSQL_CODE_TYPE` at the system level.

During the conversion to native compilation, `TYPE` specifications are not recompiled by `dbmsupgnv.sql` to `NATIVE` because these specifications do not contain executable code.

Package specifications seldom contain executable code so the run-time benefits of compiling to `NATIVE` are not measurable. You can use the `TRUE` command-line parameter with the `dbmsupgnv.sql` script to exclude package specs from recompilation to `NATIVE`, saving time in the conversion process.

When converting to interpreted compilation, the dbmsupgin.sql script does not accept any parameters and does not exclude any PL/SQL units.

---

**Note:** The following procedure describes the conversion to native compilation. If you must recompile all PL/SQL modules to interpreted compilation, make these changes in the steps.

- Skip the first step.
- Set the PLSQL_CODE_TYPE compilation parameter to INTERPRETED rather than NATIVE.
- Substitute dbmsupgin.sql for the dbmsupgnv.sql script.

---

1. Ensure that a test PL/SQL unit can be compiled. For example:

   ```
   ALTER PROCEDURE my_proc COMPILE PLSQL_CODE_TYPE=NATIVE REUSE SETTINGS;
   ```

2. Shut down application services, the listener, and the database.
   - Shut down all of the Application services including the Forms Processes, Web Servers, Reports Servers, and Concurrent Manager Servers. After shutting down all of the Application services, ensure that all of the connections to the database were terminated.
   - Shut down the TNS listener of the database to ensure that no new connections are made.
   - Shut down the database in normal or immediate mode as the user SYS. See the *Oracle Database Administrator's Guide*.
3. Set PLSQL_CODE_TYPE to NATIVE in the compilation parameter file. If the database is using a server parameter file, then set this after the database has started.

   The value of PLSQL_CODE_TYPE does not affect the conversion of the PL/SQL units in these steps. However, it does affect all subsequently compiled units, so explicitly set it to the desired compilation type.
4. Start up the database in upgrade mode, using the UPGRADE option. For information about SQL*Plus STARTUP, see the *SQL*Plus User's Guide and Reference*.
5. Run this code to list the invalid PL/SQL units. You can save the output of the query for future reference with the SQL SPOOL statement:

   ```
   -- To save the output of the query to a file:
     SPOOL pre_update_invalid.log
   SELECT o.OWNER, o.OBJECT_NAME, o.OBJECT_TYPE
   FROM DBA_OBJECTS o, DBA_PLSQL_OBJECT_SETTINGS s
   WHERE o.OBJECT_NAME = s.NAME AND o.STATUS='INVALID';
   -- To stop spooling the output: SPOOL OFF
   ```

   If any Oracle supplied units are invalid, try to validate them by recompiling them. For example:

   ```
   ALTER PACKAGE SYS.DBMS_OUTPUT COMPILE BODY REUSE SETTINGS;
   ```

   If the units cannot be validated, save the spooled log for future resolution and continue.

6. Run this query to determine how many objects are compiled NATIVE and INTERPRETED (to save the output, use the SQL SPOOL statement):

```
SELECT TYPE, PLSQL_CODE_TYPE, COUNT(*)
FROM DBA_PLSQL_OBJECT_SETTINGS
WHERE PLSQL_CODE_TYPE IS NOT NULL
GROUP BY TYPE, PLSQL_CODE_TYPE
ORDER BY TYPE, PLSQL_CODE_TYPE;
```

   Any objects with a NULL plsql_code_type are special internal objects and can be ignored.

7. Run the $ORACLE_HOME/rdbms/admin/dbmsupgnv.sql script as the user SYS to update the plsql_code_type setting to NATIVE in the dictionary tables for all PL/SQL units. This process also invalidates the units. Use TRUE with the script to exclude package specifications; FALSE to include the package specifications.

   This update must be done when the database is in UPGRADE mode. The script is guaranteed to complete successfully or rollback all the changes.

8. Shut down the database and restart in NORMAL mode.

9. Before you run the utlrp.sql script, Oracle recommends that no other sessions are connected to avoid possible problems. You can ensure this with this statement:

```
ALTER SYSTEM ENABLE RESTRICTED SESSION;
```

10. Run the $ORACLE_HOME/rdbms/admin/utlrp.sql script as the user SYS. This script recompiles all the PL/SQL modules using a default degree of parallelism. See the comments in the script for information about setting the degree explicitly.

    If for any reason the script is abnormally terminated, rerun the utlrp.sql script to recompile any remaining invalid PL/SQL modules.

11. After the compilation completes successfully, verify that there are no invalid PL/SQL units using the query in step 5. You can spool the output of the query to the post_upgrade_invalid.log file and compare the contents with the pre_upgrade_invalid.log file, if it was created previously.

12. Reexecute the query in step 6. If recompiling with dbmsupgnv.sql, confirm that all PL/SQL units, except TYPE specifications and package specifications if excluded, are NATIVE. If recompiling with dbmsupgin.sql, confirm that all PL/SQL units are INTERPRETED.

13. Disable the restricted session mode for the database, then start the services that you previously shut down. To disable restricted session mode, use this statement:

```
ALTER SYSTEM DISABLE RESTRICTED SESSION;
```

## Performing Multiple Transformations with Pipelined Table Functions

This section explains how to chain special kinds of functions known as pipelined table functions. These functions are used in situations such as data warehousing to apply multiple transformations to data.

---

**Note:** A pipelined table function cannot be run over a database link. The reason is that the return type of a pipelined table function is a SQL user-defined type, which can be used only in a single database (as explained in *Oracle Database Object-Relational Developer's Guide*). Although the return type of a pipelined table function might appear to be a PL/SQL type, the database actually converts that PL/SQL type to a corresponding SQL user-defined type.

---

Topics:

- Overview of Pipelined Table Functions
- Writing a Pipelined Table Function
- Pipelined Table Functions for Transformations
- Returning Results from Pipelined Table Functions
- Pipelining Data Between PL/SQL Table Functions
- Optimizing Multiple Calls to Pipelined Table Functions
- Fetching from Results of Pipelined Table Functions
- Passing Data with Cursor Variables
- Performing DML Statements Inside Pipelined Table Functions
- Performing DML Statements on Pipelined Table Functions
- Exception Handlers in Pipelined Table Functions

## Overview of Pipelined Table Functions

Pipelined table functions let you use PL/SQL to program a row source. You invoke the table function as the operand of the table operator in the `FROM` list of a SQL `SELECT` statement. It is also possible to invoke a table function as a `SELECT` list item; here, you do not use the table operator.

A table function can take a collection of rows as input. An input collection parameter can be either a collection type or a cursor variable.

Execution of a table function can be parallelized, and returned rows can be streamed directly to the next process without intermediate staging. Rows from a collection returned by a table function can also be pipelined, that is, iteratively returned as they are produced, instead of in a batch after all processing of the table function's input is completed.

---

**Note:** When rows from a collection returned by a table function are pipelined, the pipelined function always references the current state of the data. After opening the cursor on the collection, if the data in the collection is changed, then the change is reflected in the cursor. PL/SQL variables are private to a session and are not transactional. Therefore, the notion of read-consistency, well known for its applicability to table data, does not apply to PL/SQL collection variables.

---

Streaming, pipelining, and parallel execution of table functions can improve performance:

- By enabling multithreaded, concurrent execution of table functions
- By eliminating intermediate staging between processes
- By improving query response time: With non-pipelined table functions, the entire collection returned by a table function must be constructed and returned to the server before the query can return a single result row. Pipelining enables rows to be returned iteratively, as they are produced. This also reduces the memory that a table function requires, as the object cache need not materialize the entire collection.
- By iteratively providing result rows from the collection returned by a table function as the rows are produced instead of waiting until the entire collection is staged in tables or memory and then returning the entire collection.

## Writing a Pipelined Table Function

You declare a pipelined table function by specifying the `PIPELINED` keyword. Pipelined functions can be defined at the schema level with `CREATE FUNCTION` or in a package. The `PIPELINED` keyword indicates that the function returns rows iteratively. The return type of the pipelined table function must be a supported collection type, such as a nested table or a varray. This collection type can be declared at the schema level or inside a package. Inside the function, you return individual elements of the collection type. The elements of the collection type must be supported SQL data types, such as `NUMBER` and `VARCHAR2`. PL/SQL data types, such as `PLS_INTEGER` and `BOOLEAN`, are not supported as collection elements in a pipelined function.

Example 12–26 shows how to assign the result of a pipelined table function to a PL/SQL collection variable and use the function in a `SELECT` statement.

***Example 12–26 Assigning the Result of a Table Function***

```
CREATE OR REPLACE PACKAGE pkg1 AS
  TYPE numset_t IS TABLE OF NUMBER;
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED;
END pkg1;
/

CREATE PACKAGE BODY pkg1 AS
  -- FUNCTION f1 returns a collection of elements (1,2,3,... x)
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED IS
  BEGIN
    FOR i IN 1..x LOOP
      PIPE ROW(i);
    END LOOP;
    RETURN;
  END f1;
END pkg1;
/
```

Use pipelined function in `FROM` clause of `SELECT` statement:

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

Result:

```
COLUMN_VALUE
------------
           1
           2
           3
```

```
          4
          5

5 rows selected.
```

## Pipelined Table Functions for Transformations

A pipelined table function can accept any argument that regular functions accept. A table function that accepts a cursor variable as an argument can serve as a transformation function. That is, it can use the cursor variable to fetch the input rows, perform some transformation on them, and then pipeline the results out.

In Example 12–27, the `f_trans` function converts a row of the `employees` table into two rows.

***Example 12–27 Pipelined Table Function for Transformation***

```
CREATE OR REPLACE PACKAGE refcur_pkg IS
  TYPE refcur_t IS REF CURSOR RETURN employees%ROWTYPE;
  TYPE outrec_typ IS RECORD (
    var_num    NUMBER(6),
    var_char1  VARCHAR2(30),
    var_char2  VARCHAR2(30)
  );
  TYPE outrecset IS TABLE OF outrec_typ;
  FUNCTION f_trans (p refcur_t) RETURN outrecset PIPELINED;
END refcur_pkg;
/

CREATE OR REPLACE PACKAGE BODY refcur_pkg IS
  FUNCTION f_trans (p refcur_t) RETURN outrecset PIPELINED IS
    out_rec outrec_typ;
    in_rec  p%ROWTYPE;
  BEGIN
    LOOP
      FETCH p INTO in_rec;
      EXIT WHEN p%NOTFOUND;
      -- first row
      out_rec.var_num := in_rec.employee_id;
      out_rec.var_char1 := in_rec.first_name;
      out_rec.var_char2 := in_rec.last_name;
      PIPE ROW(out_rec);
      -- second row
      out_rec.var_char1 := in_rec.email;
      out_rec.var_char2 := in_rec.phone_number;
      PIPE ROW(out_rec);
    END LOOP;
    CLOSE p;
    RETURN;
  END f_trans;
END refcur_pkg;
/
```

Use f_transc table function in query:

```
SELECT * FROM TABLE (
   refcur_pkg.f_trans (
     CURSOR (SELECT * FROM employees WHERE department_id = 60)
   )
);
```

Result:

```
   VAR_NUM VAR_CHAR1                      VAR_CHAR2
---------- ------------------------------ ------------------------------
       103 Alexander                      Hunold
       103 AHUNOLD                        590.423.4567
       104 Bruce                          Ernst
       104 BERNST                         590.423.4568
       105 David                          Austin
       105 DAUSTIN                        590.423.4569
       106 Valli                          Pataballa
       106 VPATABAL                       590.423.4560
       107 Diana                          Lorentz
       107 DLORENTZ                       590.423.5567
```

In the preceding query, the pipelined table function `f_trans` fetches rows from the `CURSOR` subquery `SELECT * FROM employees` ..., performs the transformation, and pipelines the results back to the user as a table. The function produces two output rows (collection elements) for each input row.

When a `CURSOR` subquery is passed from SQL to a cursor variable function argument as in Example 12–27, the referenced cursor is open when the function begins running.

## Returning Results from Pipelined Table Functions

In PL/SQL, the `PIPE ROW` statement causes a pipelined table function to pipe a row and continue processing. The statement enables a PL/SQL table function to return rows as soon as they are produced. For performance, the PL/SQL run-time system provides the rows to the consumer in batches.

In Example 12–27, the `PIPE ROW(out_rec)` statement pipelines data out of the PL/SQL table function. `out_rec` is a record, and its type matches the type of an element of the output collection.

The `PIPE ROW` statement may be used only in the body of pipelined table functions; an exception is raised if it is used anywhere else. The `PIPE ROW` statement can be omitted for a pipelined table function that returns no rows.

A pipelined table function may have a `RETURN` statement that does not return a value. The `RETURN` statement transfers the control back to the consumer and ensures that the next fetch gets a `NO_DATA_FOUND` exception.

Because table functions pass control back and forth to an invoking subprogram as rows are produced, there is a restriction on combining table functions and `PRAGMA AUTONOMOUS_TRANSACTION`. If a table function is part of an autonomous transaction, it must `COMMIT` or `ROLLBACK` before each `PIPE ROW` statement, to avoid an error in the calling subprogram.

The database has three special SQL data types that enable you to dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type, including object and collection types. You can also use these three special types to create unnamed types, including anonymous collection types. The types are `SYS.ANYTYPE`, `SYS.ANYDATA`, and `SYS.ANYDATASET`. The `SYS.ANYDATA` type can be useful in some situations as a return value from table functions.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the interfaces to the `ANYTYPE`, `ANYDATA`, and `ANYDATASET` types and about the `DBMS_TYPES` package for use with these types

## Pipelining Data Between PL/SQL Table Functions

With serial execution, results are pipelined from one PL/SQL table function to another using an approach similar to co-subprogram execution. For example, this statement pipelines results from function `g` to function `f`:

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM TABLE(g()))));
```

Parallel execution works similarly except that each function runs in a different process (or set of processes).

## Optimizing Multiple Calls to Pipelined Table Functions

Multiple calls to a pipelined table function, either in the same query or in separate queries result in multiple executions of the underlying implementation. By default, there is no buffering or reuse of rows. For example:

```
SELECT * FROM TABLE(f(...)) t1, TABLE(f(...)) t2
  WHERE t1.id = t2.id;
SELECT * FROM TABLE(f());
SELECT * FROM TABLE(f());
```

If the function always produces the same result value for each combination of values passed in, you can declare the function `DETERMINISTIC`, and the database automatically buffers rows for it. If the function is not really deterministic, results are unpredictable.

## Fetching from Results of Pipelined Table Functions

PL/SQL cursors and cursor variables can be defined for queries over table functions. For example:

```
OPEN c FOR SELECT * FROM TABLE(f(...));
```

Cursors over table functions have the same fetch semantics as ordinary cursors. Cursor variable assignments based on table functions do not have any special semantics.

However, the SQL optimizer does not optimize across PL/SQL statements. For example:

```
DECLARE
  r SYS_REFCURSOR;
BEGIN
  OPEN r FOR
    SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab)));
  SELECT * BULK COLLECT INTO rec_tab FROM TABLE(g(r));
END;
/
```

does not run as well as:

```
SELECT * FROM TABLE(g(CURSOR(SELECT * FROM
  TABLE(f(CURSOR(SELECT * FROM tab))))));
```

This is so even ignoring the overhead associated with running two SQL statements and if the results can be pipelined between the two statements.

## Passing Data with Cursor Variables

You can pass a set of rows to a PL/SQL function in a cursor variable parameter. For example, this function is declared to accept an argument of the predefined weakly typed REF CURSOR type SYS_REFCURSOR:

```
FUNCTION f(p1 IN SYS_REFCURSOR) RETURN ... ;
```

Results of a subquery can be passed to a function directly:

```
SELECT * FROM TABLE(f(CURSOR(SELECT empid FROM tab)));
```

In the preceding example, the CURSOR keyword causes the results of a subquery to be passed as a cursor variable parameter.

A predefined weak REF CURSOR type SYS_REFCURSOR is also supported. With SYS_REFCURSOR, you need not first create a REF CURSOR type in a package before you can use it.

To use a strong REF CURSOR type, you still must create a PL/SQL package and declare a strong REF CURSOR type in it. Also, if you are using a strong REF CURSOR type as an argument to a table function, then the actual type of the cursor variable argument must match the column type, or an error is generated. Weakly typed cursor variable arguments to table functions can only be partitioned using the PARTITION BY ANY clause. You cannot use range or hash partitioning for weakly typed cursor variable arguments.

PL/SQL functions can accept multiple IN cursor variables, as in Example 12–28.

For more information about cursor variables, see "Cursor Variable Creation" on page 6-26.

***Example 12–28 Function with Two Cursor Variable Parameters***

```
CREATE OR REPLACE PACKAGE refcur_pkg IS
  TYPE refcur_t1 IS REF CURSOR RETURN employees%ROWTYPE;
  TYPE refcur_t2 IS REF CURSOR RETURN departments%ROWTYPE;
  TYPE outrec_typ IS RECORD (
    var_num    NUMBER(6),
    var_char1  VARCHAR2(30),
    var_char2  VARCHAR2(30)
  );
  TYPE outrecset IS TABLE OF outrec_typ;
  FUNCTION g_trans (p1 refcur_t1, p2 refcur_t2) RETURN outrecset PIPELINED;
END refcur_pkg;
/

CREATE PACKAGE BODY refcur_pkg IS
  FUNCTION g_trans (
    p1 refcur_t1,
    p2 refcur_t2
  ) RETURN outrecset PIPELINED
  IS
    out_rec outrec_typ;
    in_rec1 p1%ROWTYPE;
    in_rec2 p2%ROWTYPE;
  BEGIN
    LOOP
      FETCH p2 INTO in_rec2;
      EXIT WHEN p2%NOTFOUND;
    END LOOP;
    CLOSE p2;
```

```
    LOOP
      FETCH p1 INTO in_rec1;
      EXIT WHEN p1%NOTFOUND;
      -- first row
      out_rec.var_num := in_rec1.employee_id;
      out_rec.var_char1 := in_rec1.first_name;
      out_rec.var_char2 := in_rec1.last_name;
      PIPE ROW(out_rec);
      -- second row
      out_rec.var_num := in_rec2.department_id;
      out_rec.var_char1 := in_rec2.department_name;
      out_rec.var_char2 := TO_CHAR(in_rec2.location_id);
      PIPE ROW(out_rec);
    END LOOP;
    CLOSE p1;
    RETURN;
  END g_trans;
END refcur_pkg;
/
```

Use `g_trans` table function in query:

```
SELECT * FROM TABLE (
  refcur_pkg.g_trans (
    CURSOR (SELECT * FROM employees WHERE department_id = 60),
    CURSOR (SELECT * FROM departments WHERE department_id = 60)
  )
);
```

Result:

```
   VAR_NUM VAR_CHAR1                      VAR_CHAR2
---------- ------------------------------ ------------------------------
       103 Alexander                      Hunold
        60 IT                             1400
       104 Bruce                          Ernst
        60 IT                             1400
       105 David                          Austin
        60 IT                             1400
       106 Valli                          Pataballa
        60 IT                             1400
       107 Diana                          Lorentz
        60 IT                             1400

10 rows selected.
```

You can pass table function return values to other table functions by creating a cursor variable that iterates over the returned data:

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM TABLE(g(...)))));
```

You can explicitly open a cursor variable for a query and pass it as a parameter to a table function:

```
DECLARE
  r SYS_REFCURSOR;
  rec ...;
BEGIN
  OPEN r FOR SELECT * FROM TABLE(f(...));
  -- Must return a single row result set.
  SELECT * INTO rec FROM TABLE(g(r));
END;
```

```
/
```

In this case, the table function closes the cursor when it completes, so your program must not explicitly try to close the cursor.

A table function can compute aggregate results using the cursor variable parameter, as in Example 12–29, which computes a weighted average by iterating over a set of input rows.

***Example 12–29 Pipelined Table Function as Aggregate Function***

```
DROP TABLE gradereport;
CREATE TABLE gradereport (
  student VARCHAR2(30),
  subject VARCHAR2(30),
  weight NUMBER,
  grade NUMBER
);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark', 'Physics', 4, 4);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark','Chemistry', 4, 3);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark','Maths', 3, 3);

INSERT INTO gradereport (student, subject, weight, grade)
VALUES ('Mark','Economics', 3, 4);

CREATE PACKAGE pkg_gpa IS
  TYPE gpa IS TABLE OF NUMBER;
  FUNCTION weighted_average(input_values SYS_REFCURSOR)
    RETURN gpa PIPELINED;
END pkg_gpa;
/

CREATE PACKAGE BODY pkg_gpa IS
  FUNCTION weighted_average (input_values SYS_REFCURSOR)
    RETURN gpa PIPELINED
  IS
    grade         NUMBER;
    total         NUMBER := 0;
    total_weight  NUMBER := 0;
    weight        NUMBER := 0;
  BEGIN
    -- Function accepts cursor variable and loops through all input rows
    LOOP
      FETCH input_values INTO weight, grade;
      EXIT WHEN input_values%NOTFOUND;
      -- Accumulate the weighted average
      total_weight := total_weight + weight;
      total := total + grade*weight;
    END LOOP;
    PIPE ROW (total / total_weight);
    RETURN; -- the function returns a single result
  END weighted_average;
END pkg_gpa;
/
```

Use function in query (`column_value` is a keyword that returns the contents of nested table):

```
SELECT w.column_value "weighted result" FROM TABLE (
  pkg_gpa.weighted_average (
    CURSOR (SELECT weight, grade FROM gradereport)
  )
) w;
```

Result is a nested table with single row:

```
weighted result
---------------
            3.5

1 row selected.
```

## Performing DML Statements Inside Pipelined Table Functions

To run DML statements, declare a pipelined table function with the `AUTONOMOUS_TRANSACTION` pragma, which causes the function to run in a transaction not shared by other processes. For example (where `CollType` is a previously declared collection type):

```
CREATE OR REPLACE FUNCTION f (p SYS_REFCURSOR)
  RETURN CollType PIPELINED
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  NULL;
END;
/
```

During parallel execution, each instance of the table function creates an independent transaction.

## Performing DML Statements on Pipelined Table Functions

A pipelined table function cannot be the target table in `UPDATE`, `INSERT`, or `DELETE` statement. For example, these statements raise an exception:

```
UPDATE F(CURSOR(SELECT * FROM tab)) SET col = value;
  INSERT INTO f(...) VALUES ('any', 'thing');
```

However, you can create a view over a table function and use `INSTEAD OF` triggers to update it. For example:

```
CREATE VIEW BookTable AS SELECT x.Name, x.Author
  FROM TABLE(GetBooks('data.txt')) x;
```

This `INSTEAD OF` trigger fires when the user inserts a row into the `BookTable` view:

```
CREATE TRIGGER BookTable_insert
INSTEAD OF INSERT ON BookTable
REFERENCING NEW AS n
FOR EACH ROW
BEGIN
  ...
END
/
```

```
INSERT INTO BookTable (...) VALUES (...);
```

INSTEAD OF triggers can be defined for all DML operations on a view built on a table function.

### Exception Handlers in Pipelined Table Functions

Exception handling in pipelined table functions works just as it does with regular functions.

Some languages, such as C and Java, provide a mechanism for user-supplied exception handling. If an exception raised in a table function is handled, the table function runs the exception handler and continues processing. Exiting the exception handler takes control to the enclosing scope. If the exception is cleared, execution proceeds normally.

An unhandled exception in a table function causes the parent transaction to roll back.

## Updating Large Tables in Parallel

The DBMS_PARALLEL_EXECUTE package enables you to incrementally update the data in a large table in parallel, in two high-level steps:

1. Group sets of rows in the table into smaller chunks.
2. Apply the desired UPDATE statement to the chunks in parallel, committing each time you have finished processing a chunk.

This technique is recommended whenever you are updating a lot of data. Its advantages are:

- You lock only one set of rows at a time, for a relatively short time, instead of locking the entire table.
- You do not lose work that has been done if something fails before the entire operation finishes.
- You reduce rollback space consumption.
- You improve performance.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for more information about the DBMS_PARALLEL_EXECUTE package

# 13

# PL/SQL Language Elements

This chapter summarizes the syntax and semantics of PL/SQL language elements and provides links to examples and related topics.

For instructions for reading the syntax diagrams in this chapter, see *Oracle Database SQL Language Reference*.

Topics:


- Assignment Statement
- AUTONOMOUS_TRANSACTION Pragma
- Block
- Basic LOOP Statement
- CASE Statement
- CLOSE Statement
- Collection
- Collection Method Call
- Comment
- Constant
- CONTINUE Statement
- Cursor FOR LOOP Statement
- Cursor Variable
- DELETE Statement Extension
- EXCEPTION_INIT Pragma
- Exception
- Exception Handler
- EXECUTE IMMEDIATE Statement
- EXIT Statement
- Explicit Cursor
- Expression
- FETCH Statement
- FOR LOOP Statement
- FORALL Statement

- Function
- GOTO Statement
- IF Statement
- Implicit Cursor Attribute
- INLINE Pragma
- INSERT Statement Extension
- Literal
- Named Cursor Attribute
- NULL Statement
- OPEN Statement
- OPEN FOR Statement
- Parameter
- Procedure
- RAISE Statement
- Record
- RESTRICT_REFERENCES Pragma (deprecated)
- RETURN Statement
- RETURNING INTO Clause
- %ROWTYPE Attribute
- SELECT INTO Statement
- SERIALLY_REUSABLE Pragma
- SQLCODE Function
- SQLERRM Function
- %TYPE Attribute
- UPDATE Statement Extensions
- Variable
- WHILE LOOP Statement

> **See Also:** Chapter 2, "PL/SQL Language Fundamentals"

# Assignment Statement

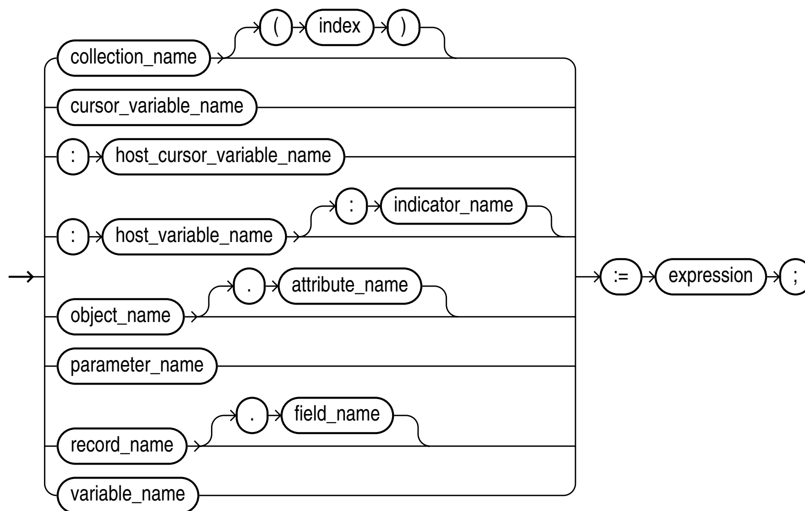
The assignment statement sets the value of a variable, field, parameter, or element.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

### *assignment_statement* ::=




See *expression ::=* on page 13-58.

## Semantics

#### *attribute_name*

The name of an attribute of *object_name*.

#### *collection_name*

The name of a collection. For information about collections, see "Collection" on page 13-23.

#### *cursor_variable_name*

The name of a cursor variable. For information about cursor variables, see "Cursor Variable" on page 13-39.

#### *expression*

The expression whose value is to be assigned to the target (the item to the left of the assignment operator) when the assignment statement runs.

The value of *expression* must have a data type that is compatible with the data type of the target (see "PL/SQL Data Type Conversion" on page 3-26).

If the target is a variable defined as NOT NULL, the value of *expression* cannot be NULL. If the target is a cursor variable, the value of *expression* must also be a cursor variable.

For general information about expressions, see "Expression" on page 13-58.

### *field_name*

The name of a field in *record_name*.

Specify *field_name* to assign the value of *expression* to a specific field of *record_name*.

Omit *field_name* to assign the value of *expression* to all fields of *record_name* simultaneously; that is, to assign one record to another. You can assign one record to another only if their declarations refer to the same table or cursor.

### *host_cursor_variable_name*

The name of a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. Do not put space between the colon (:) and *host_cursor_variable_name*.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

### *host_variable_name*

The name of a variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. Do not put space between the colon (:) and *host_variable_name*.

### *index*

A numeric expression whose value has the data type PLS_INTEGER or a data type that can be implicitly converted to PLS_INTEGER (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions" on page 3-29).

Specify *index* to assign the value of *expression* to a specific element of *collection_name*.

Omit *index* to assign the value of *expression* to all elements of *collection_name* simultaneously; that is, to assign one collection to another. You can assign one collection to another only if the collections have the same data type (not the same element type).

### *indicator_name*

The name of an indicator variable for *host_variable_name*.

An indicator variable indicates the value or condition of its host variable. For example, in the Oracle Precompiler environment, indicator variables let you detect NULL or truncated values in output host variables.

### *object_name*

The name of an instance of an ADT. For general information about ADTs, see "Abstract Data Types" on page 1-8.

***parameter_name***

The name of a formal OUT or IN OUT parameter of the subprogram in which the assignment statement appears. For information about formal parameters, see "Parameter" on page 13-101.

***record_name***

The name of a record. For information about records, see "Record" on page 13-106.

***variable_name***

The name of a PL/SQL variable. For information about variables, see "Variable" on page 13-131.

## Examples

- Example 2–25, "Assigning Values to Variables with Assignment Statement" on page 2-25
- Example 2–28, "Assigning BOOLEAN Values" on page 2-25
- Example 3–4, "Assigning a Literal Value to a TIMESTAMP Variable" on page 3-17
- Example 5–17, "Data Type Compatibility for Collection Assignment" on page 5-14

## Related Topics

**In this chapter:**

- "FETCH Statement" on page 13-67
- "SELECT INTO Statement" on page 13-118

**In other chapters:**

- "Scope and Visibility of Identifiers" on page 2-18
- "Assigning Values to Variables" on page 2-22
- "Assignments to Collections" on page 5-13
- "Assignments to Records" on page 5-37

# AUTONOMOUS_TRANSACTION Pragma

The AUTONOMOUS_TRANSACTION pragma marks a routine as **autonomous**; that is, independent of the main transaction.

In this context, a **routine** is one of these:

- Schema-level (not nested) anonymous PL/SQL block
- Standalone, packaged, or nested subprogram
- Method of an ADT
- Database trigger

When an autonomous routine is invoked, the main transaction is suspended. The autonomous transaction is fully independent of the main transaction: they share no locks, resources, or commit dependencies. The autonomous transaction does not affect the main transaction.

Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. They become visible to the main transaction when it resumes only if its isolation level is READ COMMITTED (the default).

Topics:

- Syntax
- Examples
- Related Topics

## Syntax

***autonomous_transaction_pragma*** **::=**

```
→ PRAGMA → AUTONOMOUS_TRANSACTION → ;
```

## Usage

You cannot apply this pragma to an entire package, but you can apply it to each subprogram in a package.

You cannot apply this pragma to an entire ADT, but you can apply it to each method of an ADT.

Unlike an ordinary trigger, an autonomous trigger can contain transaction control statements, such as COMMIT and ROLLBACK, and can issue DDL statements (such as CREATE and DROP) through the EXECUTE IMMEDIATE statement.

In the main transaction, rolling back to a savepoint located before the call to the autonomous subprogram does *not* roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

If an autonomous transaction attempts to access a resource held by the main transaction (which cannot resume until the autonomous routine exits), a deadlock can occur. The database raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.

If you try to exit an active autonomous transaction without committing or rolling back, the database raises an exception. If the exception goes unhandled, or if the transaction ends because of some other unhandled exception, the transaction is rolled back.

You cannot run a `PIPE ROW` statement in your autonomous routine while your autonomous transaction is open. You must close the autonomous transaction before running the `PIPE ROW` statement. This is normally accomplished by committing or rolling back the autonomous transaction before running the `PIPE ROW` statement.

## Examples


- Example 6–48, "Declaring an Autonomous Function in a Package" on page 6-49
- Example 6–49, "Declaring an Autonomous Standalone Procedure" on page 6-49
- Example 6–50, "Declaring an Autonomous PL/SQL Block" on page 6-50
- Example 6–51, "Autonomous Trigger the Logs INSERT Statements" on page 6-51
- Example 6–52, "Autonomous Trigger Using Native Dynamic SQL for DDL" on page 6-53
- Example 6–53, "Invoking an Autonomous Function" on page 6-54


## Related Topics

**In this chapter:**


- "EXCEPTION_INIT Pragma" on page 13-43
- "INLINE Pragma" on page 13-87
- "RESTRICT_REFERENCES Pragma" on page 13-109
- "SERIALLY_REUSABLE Pragma" on page 13-122


**In other chapters:**


- "Autonomous Transactions" on page 6-47

# Block

The block, which groups related declarations and statements, is the basic unit of a PL/SQL source program. It has an optional declarative part, a required executable part, and an optional exception-handling part. Declarations are local to the block and cease to exist when the block completes execution. Blocks can be nested.

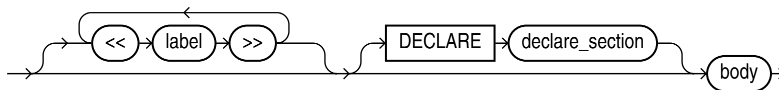
An anonymous block is an executable statement.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

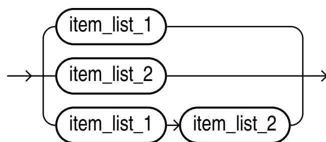
## Syntax

### *plsql_block* ::=



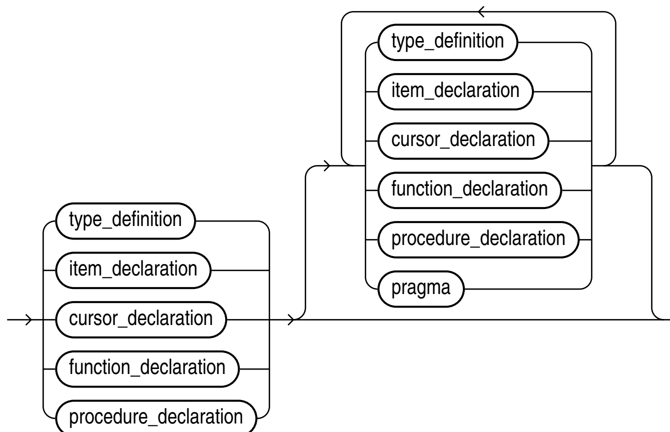

See *body ::=* on page 13-11.

### *declare_section* ::=




See *item_list_2 ::=* on page 13-9.

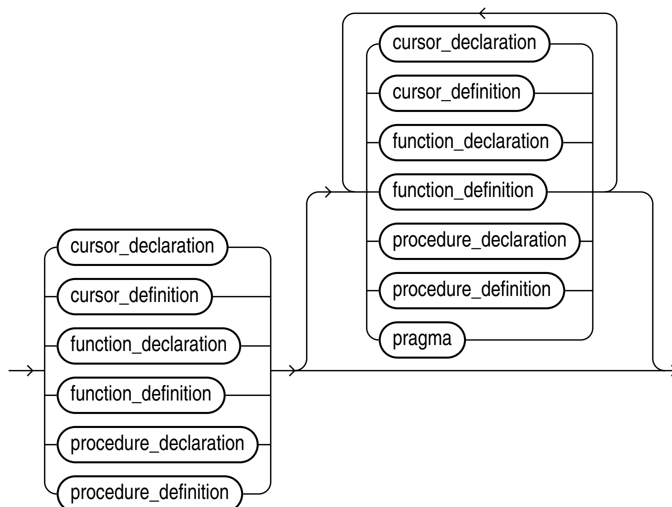
### *item_list_1* ::=




See:

- *function_declaration ::=* on page 13-76
- *item_declaration ::=* on page 13-10
- *pragma ::=* on page 13-10
- *procedure_declaration ::=* on page 13-103
- *type_definition ::=* on page 13-9

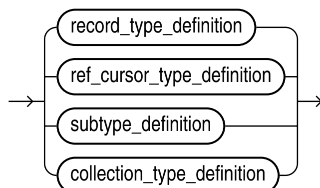
***item_list_2* ::=**




See:

- *function_declaration ::=* on page 13-76
- *function_definition ::=* on page 13-77
- *item_declaration ::=* on page 13-10
- *pragma ::=* on page 13-10
- *procedure_declaration ::=* on page 13-103
- *procedure_definition ::=* on page 13-103
- *type_definition ::=* on page 13-9

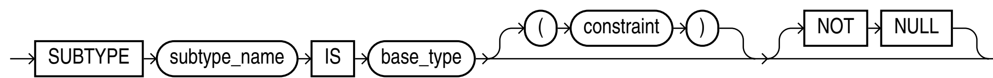
***type_definition* ::=**



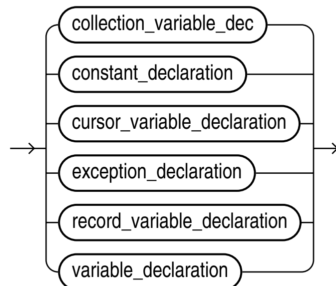

See:

- *collection_type_definition ::=* on page 13-23
- *record_type_definition ::=* on page 13-106
- *ref_cursor_type_definition ::=* on page 13-39

***subtype_definition* ::=**



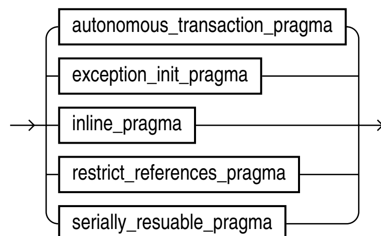

***item_declaration* ::=**




See:

- *collection_variable_dec ::=* on page 13-24
- *constant_declaration ::=* on page 13-32
- *cursor_definition ::=* on page 13-54
- *cursor_variable_declaration ::=* on page 13-39
- *exception_declaration ::=* on page 13-45
- *record_variable_declaration ::=* on page 13-106
- *variable_declaration ::=* on page 13-131

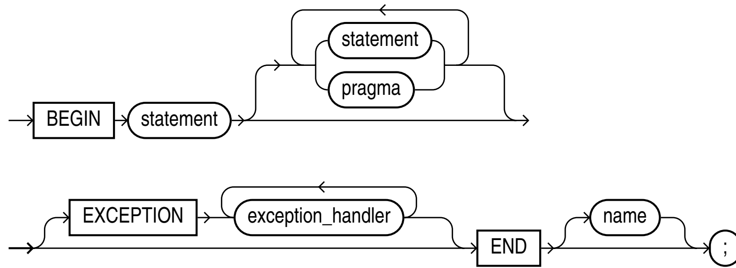
***pragma* ::=**




See:

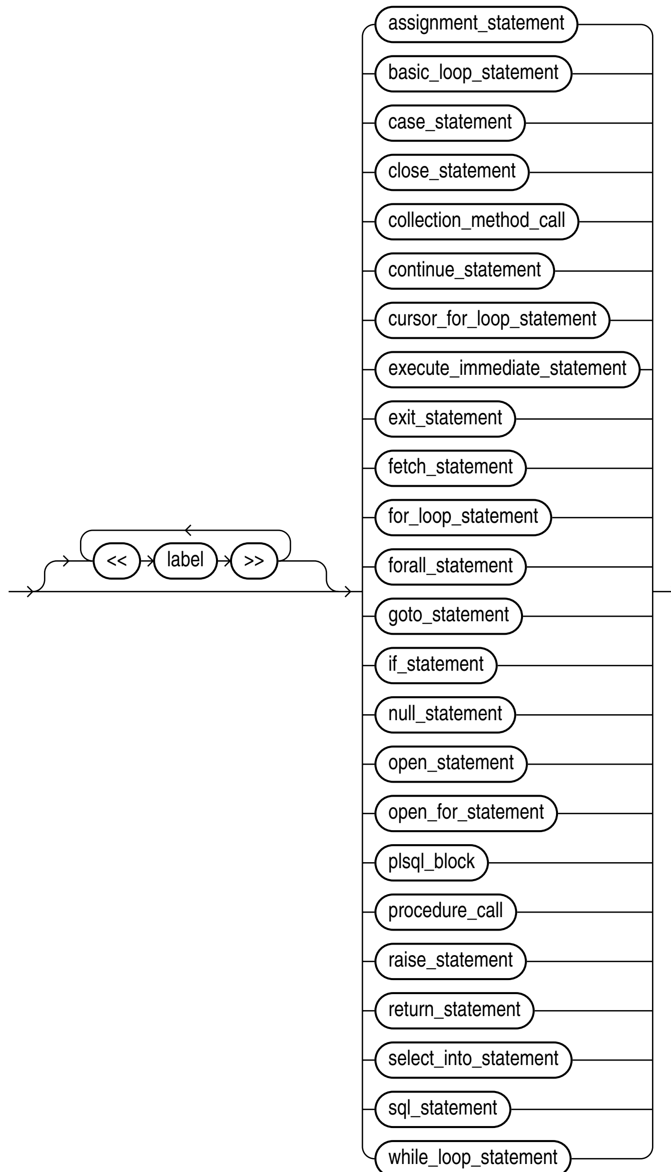
- *autonomous_transaction_pragma ::=* on page 13-6
- *exception_init_pragma ::=* on page 13-43
- *inline_pragma ::=* on page 13-87
- *restrict_references_pragma ::=* on page 13-109
- *serially_resuable_pragma ::=* on page 13-122

***body* ::=**




See *exception_handler ::=* on page 13-47.

***statement* ::=**

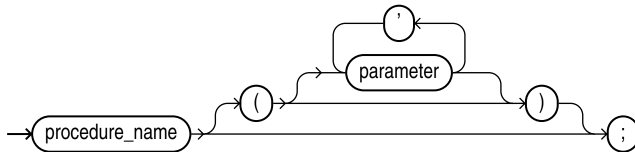



See:

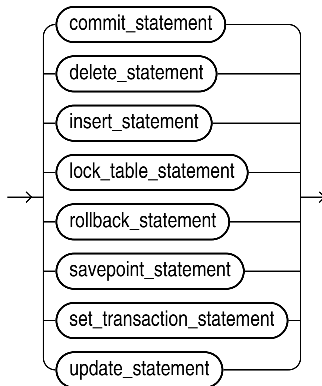
- *plsql_block ::=* on page 13-8

- *procedure_call ::=* on page 13-12
- *sql_statement ::=* on page 13-12

***procedure_call*** **::=**




***sql_statement*** **::=**




## Semantics

***assignment_statement***

See "Assignment Statement" on page 13-3.

***base_type***

The base type of the subtype that you are defining. The base type can be either a predefined PL/SQL scalar data type (or subtype) or an existing user-defined subtype. For information about predefined scalar data types, see "Predefined PL/SQL Scalar Data Types and Subtypes" on page 3-2. For information about user-defined subtypes, see "User-Defined PL/SQL Subtypes" on page 3-23.

***basic_loop_statement***

See "Basic LOOP Statement" on page 13-16.

**BEGIN**

Starts the executable part of the block, which contains executable statements.

***case_statement***

See "CASE Statement" on page 13-18.

***collection_method_call***

A call to one of these collection methods, which are procedures:

- `DELETE`
- `EXTEND`

- TRIM

For syntax, see "Collection Method Call" on page 13-27.

#### *continue_statement*

See "CONTINUE Statement" on page 13-35.

#### *close_statement*

See "CLOSE Statement" on page 13-21.

#### *commit_statement*

SQL `COMMIT` statement. For syntax, see *Oracle Database SQL Language Reference*.

#### *constraint*

The constraint on the base type of the subtype that you are defining.

**Restriction on *constraint*** Do not specify *`constraint`* if the base type cannot specify either a maximum size or a precision and scale.

#### *cursor_for_loop_statement*

See "Cursor FOR LOOP Statement" on page 13-37.

#### DECLARE

Starts the declarative part of the block, which contains local declarations. Local declarations exist only in the current block and its sub-blocks and are not visible to enclosing blocks.

#### *delete_statement*

SQL `DELETE` statement. For syntax, see *Oracle Database SQL Language Reference*. See also "DELETE Statement Extension" on page 13-42.

#### END

Ends the block.

#### EXCEPTION

Starts the exception-handling part of the block. When an exception is raised, normal execution of the block stops and control transfers to the appropriate *`exception_handler`*. After the exception handler completes, execution resumes with the statement following the block. For more information about exception-handling, see Chapter 11, "PL/SQL Error Handling."

#### *exception_handler*

See "Exception Handler" on page 13-47.

#### *execute_immediate_statement*

See "EXECUTE IMMEDIATE Statement" on page 13-49.

#### *exit_statement*

See "EXIT Statement" on page 13-52.

***fetch_statement***

See "FETCH Statement" on page 13-67.

***for_loop_statement***

See "FOR LOOP Statement" on page 13-70.

***forall_statement***

See "FORALL Statement" on page 13-73.

***goto_statement***

See "GOTO Statement" on page 13-80.

***if_statement***

See "IF Statement" on page 13-82.

***insert_statement***

SQL `INSERT` statement. For syntax, see *Oracle Database SQL Language Reference*. See also "INSERT Statement Extension" on page 13-89.

***label***

An undeclared identifier. If a block or statement has multiple labels, they must be unique for that block or statement.

***lock_table_statement***

SQL `LOCK TABLE` statement. For syntax, see *Oracle Database SQL Language Reference*.

***name***

The name of the block to which `END` applies—a *`label`*, *`function_name`*, *`procedure_name`*, or *`package_name`*.

***null_statement***

See "NULL Statement" on page 13-96.

***open_statement***

See "OPEN Statement" on page 13-97.

***open_for_statement***

See "OPEN FOR Statement" on page 13-98.

**( *parameter* [, *parameter* ]... )**

List of actual parameters for the procedure being called. The data type of each actual parameter must be compatible with the data type of the corresponding formal parameter (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions"). If the procedure specifies an initial value for a parameter, you can omit that parameter from the parameter list.

If the procedure has no parameters, or specifies an initial value for every parameter, you can either omit the parameter list or specify an empty parameter list.

> **See Also:** "Positional, Named, and Mixed Notation for Actual Parameters" on page 8-12

***raise_statement***

See "RAISE Statement" on page 13-105.

***return_statement***

See "RETURN Statement" on page 13-111.

***rollback_statement***

SQL `ROLLBACK` statement. For syntax, see *Oracle Database SQL Language Reference*.

***savepoint_statement***

SQL `SAVEPOINT` statement. For syntax, see *Oracle Database SQL Language Reference*.

***select_into_statement***

PL/SQL `SELECT INTO` statement. For syntax, see "SELECT INTO Statement" on page 13-118.

***set_transaction_statement***

SQL `SET TRANSACTION` statement. For syntax, see *Oracle Database SQL Language Reference*.

***subtype_name***

The name of the user-defined subtype that you are defining. For information about user-defined subtypes, see "User-Defined PL/SQL Subtypes" on page 3-23.

***update_statement***

SQL `UPDATE` statement. For syntax, see *Oracle Database SQL Language Reference*. See also "UPDATE Statement Extensions" on page 13-129.

***while_loop_statement***

See "WHILE LOOP Statement" on page 13-133.

## Examples


- Example 1–1, "PL/SQL Block Structure" on page 1-5
- Example 2–24, "Block with Multiple and Duplicate Labels" on page 2-22
- Example 4–29, "Incorrect Label Placement" on page 4-21


## Related Topics

**In this chapter:**


- "Comment" on page 13-30
- See links in "Semantics" on page 13-12.


**In other chapters:**


- "Blocks" on page 1-5
- "Identifiers" on page 2-5
- "Subtypes" on page 3-1

# Basic LOOP Statement

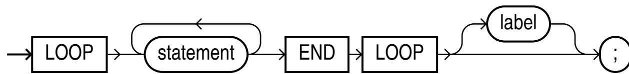
With each iteration of the basic LOOP statement, its statements run and control returns to the top of the loop. The LOOP statement ends when a statement inside the loop transfers control outside the loop or when an exception is raised.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***basic_loop_statement*** **::=**




See *statement ::=* on page 13-11.

## Semantics

***label***

A label that identifies `basic_LOOP_statement` (see ***label*** on page 13-14). CONTINUE, EXIT, and GOTO statements can reference this label.

Labels improve readability, especially when LOOP statements are nested, but only if you ensure that the label at the end of the LOOP statement matches one of the labels at the beginning of the same statement (the compiler does not check).

***statement***

To prevent an infinite loop, at least one statement must transfer control outside the loop. The statements that can transfer control outside the loop are:

- "CONTINUE Statement" on page 13-35 (when it transfers control to the next iteration of an enclosing labeled loop)
- "EXIT Statement" on page 13-52
- "GOTO Statement" on page 13-80
- "RAISE Statement" on page 13-105

## Examples

- Example 1–2, "Processing Query Result Rows One at a Time" on page 1-10
- Example 4–9, "Basic LOOP Statement with EXIT Statement" on page 4-10
- Example 4–10, "Basic LOOP Statement with EXIT WHEN Statement" on page 4-10
- Example 4–11, "Nested, Labeled Basic LOOP Statements with EXIT WHEN Statements" on page 4-11

- Example 4–12, "CONTINUE Statement in Basic LOOP Statement" on page 4-12
- Example 4–13, "CONTINUE WHEN Statement in Basic LOOP Statement" on page 4-12

## Related Topics

**In this chapter:**

- "Cursor FOR LOOP Statement" on page 13-37
- "FOR LOOP Statement" on page 13-70
- "WHILE LOOP Statement" on page 13-133

**In other chapters:**

- "Basic LOOP Statement" on page 4-9

# CASE Statement

The CASE statement chooses from a sequence of conditions and runs a corresponding statement.

The simple CASE statement evaluates a single expression and compares it to several potential values.

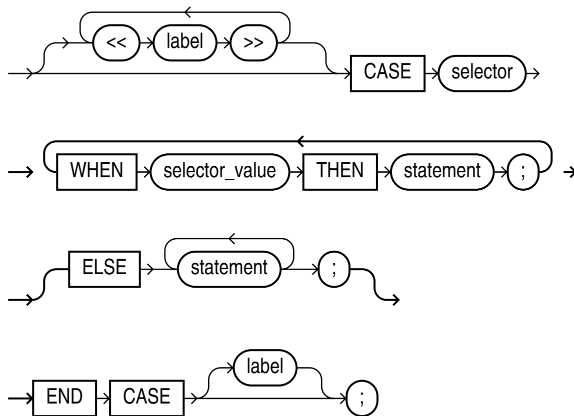
The searched CASE statement evaluates multiple Boolean expressions and chooses the first one whose value is TRUE.

Topics:

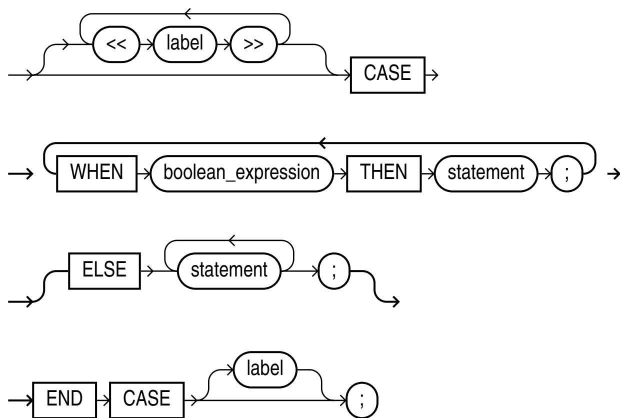
- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***simple_case_statement*** **::=**




***searched_case_statement*** **::=**




See:

- *boolean_expression ::=* on page 13-59
- *statement ::=* on page 13-11

## Semantics

#### *label*

An undeclared identifier. If the `CASE` statement has multiple labels, they must be unique for that statement.

#### *selector*

An expression whose value is evaluated once and used to select one of several alternatives. The value of *selector* can be of any PL/SQL type except `BLOB`, `BFILE`, or a user-defined type.

#### WHEN { *selector_value* | *boolean_expression* } THEN *statement*

The *selector_value*s or *boolean_expression*s are evaluated sequentially. If the value of a *selector_value* equals the value of *selector*, or if the value of a *boolean_expression* is `TRUE`, the *statement* associated with that *selector_value* or *boolean_expression* runs, and the `CASE` statement ends. Subsequent *selector_value*s or *boolean_expression*s are not evaluated.

A *selector_value* can be of any PL/SQL type except `BLOB`, `BFILE`, or a user-defined type.

> **Caution:** A *statement* can modify the database and invoke nondeterministic functions. There is no fall-through mechanism, as there is in the C `switch` statement.

#### ELSE *statement* [*statement* ]...

In the simple `CASE` statement, the statements after `ELSE` run if and only if no *selector_value* has the same value as *selector*.

In the searched `CASE` statement, the statements after `ELSE` run if and only if no *boolean_expression* has the value `TRUE`.

Without the `ELSE` clause, if no *selector_value* has the same value as *selector* or no *boolean_expression* has the value `TRUE`, the system raises a `CASE_NOT_FOUND` exception.

## Examples

- Example 4–6, "Simple CASE Statement" on page 4-7
- Example 4–7, "Searched CASE Statement" on page 4-8

## Related Topics

#### In this chapter:

- "IF Statement" on page 13-82

#### In other chapters:

- "CASE Expressions" on page 2-39
- "Conditional Selection Statements" on page 4-1

- "Simple CASE Statement" on page 4-7
- "Searched CASE Statement" on page 4-7

**See Also:**

- *Oracle Database SQL Language Reference* for information about the `NULLIF` function
- *Oracle Database SQL Language Reference* for information about the `COALESCE` function

# CLOSE Statement

The CLOSE statement closes a named cursor, thereby allowing its resources to be reused.

After closing a cursor, you can reopen it with the OPEN statement. You must close a cursor before reopening it.

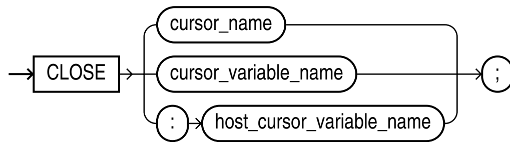
After closing a cursor variable, you can reopen it with the OPEN FOR statement. You need not close a cursor variable before reopening it.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***close_statement* ::=**




## Semantics

***cursor_name***

The name of an open explicit cursor.

***cursor_variable_name***

The name of an open cursor variable.

***host_cursor_variable_name***

The name of a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. Do not put space between the colon (:) and *host_cursor_variable_name*.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

## Examples

- Example 4–26, "EXIT in FOR LOOP" on page 4-20
- Example 6–7, "Fetching with a Cursor" on page 6-11
- Example 6–10, "Fetching Bulk Data with a Cursor" on page 6-13

## Related Topics

**In this chapter:**


- "FETCH Statement" on page 13-67
- "OPEN Statement" on page 13-97
- "OPEN FOR Statement" on page 13-98


**In other chapters:**


- "Closing Explicit Cursors" on page 6-13

# Collection

A collection groups elements of the same type in a specified order. Each element has a unique subscript that determines its position in the collection.

PL/SQL has three kinds of collections:

- Associative arrays (formerly called "PL/SQL tables" or "index-by tables")
- Nested tables
- Variable-size arrays (varrays)

Associative arrays can be indexed by either integers or strings. Nested tables and varrays are indexed by integers.

To create a collection, you first define a collection type, and then declare a variable of that type.

---

**Note:** This topic applies to collection types that you define inside a PL/SQL block or package, which are different from standalone stored collection types that you create with the "CREATE TYPE Statement" on page 14-69.

In a PL/SQL block or package, you can define all three collection types. With the `CREATE TYPE` statement, you can create nested table types and varray types, but not associative array types.

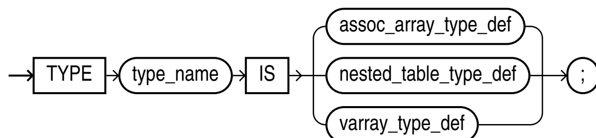
---

Topics:

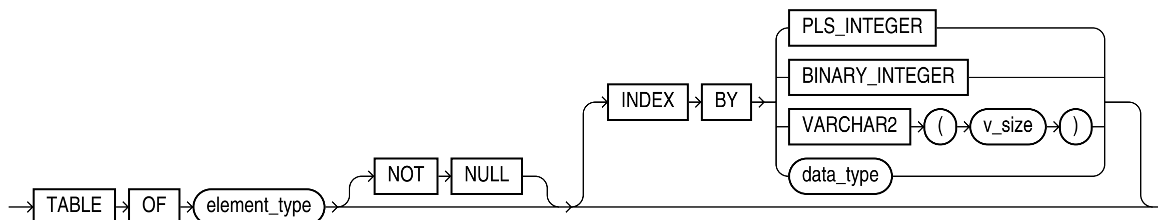
- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***collection_type_definition*** **::=**

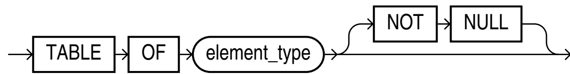



***assoc_array_type_def*** **::=**



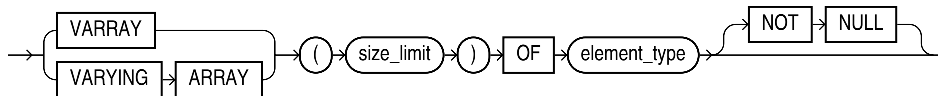
See *element_type ::=* on page 13-24.

***nested_table_type_def*** **::=**



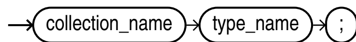

See *element_type ::=* on page 13-24.

***varray_type_def*** **::=**

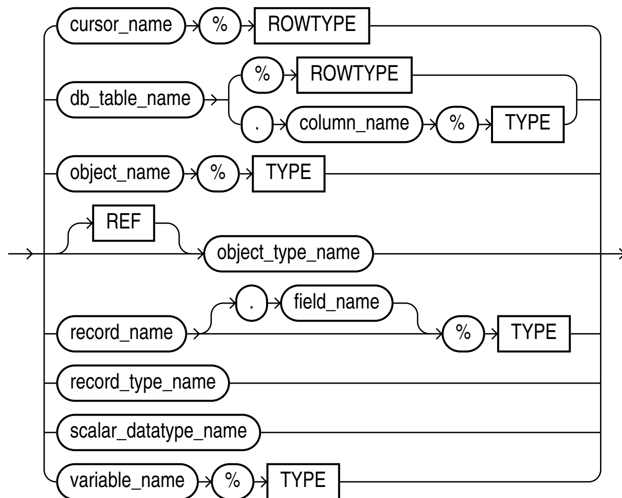



See *element_type ::=* on page 13-24.

***collection_variable_dec*** **::=**




***element_type*** **::=**




## Semantics

### *assoc_array_type_def*

The type definition for an associative array.

**Restriction on *assoc_array_type_def*** The type definition of an associative array can appear only in the declarative part of a block, subprogram, package specification, or package body.

### *collection_name*

The name of the collection variable that you are declaring.

#### *data_type*

A data type specified with either %ROWTYPE or %TYPE that represents either PLS_INTEGER, BINARY_INTEGER, or VARCHAR2(*v_size*). See *datatype ::=* on page 13-32.

#### *element_type*

The data type of the collection element (any PL/SQL data type except REF CURSOR).

In *nested_table_type_def*:

- *element_type* cannot be NCLOB.
- If *element_type* is an ADT, then *nested_table_type_def* describes a table whose columns match the name and attributes of the ADT.
- If *element_type* is a scalar type, then *nested_table_type_def* describes a table with a single, scalar type column called COLUMN_VALUE.

#### INDEX BY

Specifies the data type of the indexes of the associative array—PLS_INTEGER, BINARY_INTEGER, VARCHAR2(*v_size*) or *data_type*.

#### *nested_table_type_def*

The type definition for a nested table.

#### NOT NULL

Specifies that no element of the collection can have the value NULL.

#### *size_limit*

A positive integer literal that specifies the maximum number of elements that the varray can have. The maximum value of *size_limit* is 2147483647.

#### *type_name*

The name of the collection type that you are defining.

#### *v_size*

The length of the VARCHAR2 key by which the associative array is indexed.

#### *varray_type_def*

The type definition for a variable-size array.

### Examples

- Example 5–1, "Declaring and Populating Associative Array Indexed by String" on page 5-3
- Example 5–3, "Declaring Nested Tables, Varrays, and Associative Arrays" on page 5-8
- Example 5–4, "Declaring Collections with %TYPE" on page 5-8
- Example 5–5, "Declaring a Procedure Parameter as a Nested Table" on page 5-9
- Example 5–42, "Declaring and Initializing Record Types" on page 5-34

## Related Topics

**In this chapter:**


- "Collection Method Call" on page 13-27
- "FORALL Statement" on page 13-73
- "Record" on page 13-106


**In other chapters:**


- "Collection Topics" on page 5-1
- "Retrieving Query Results into Collections" on page 12-21
- "CREATE TYPE Statement" on page 14-69

# Collection Method Call

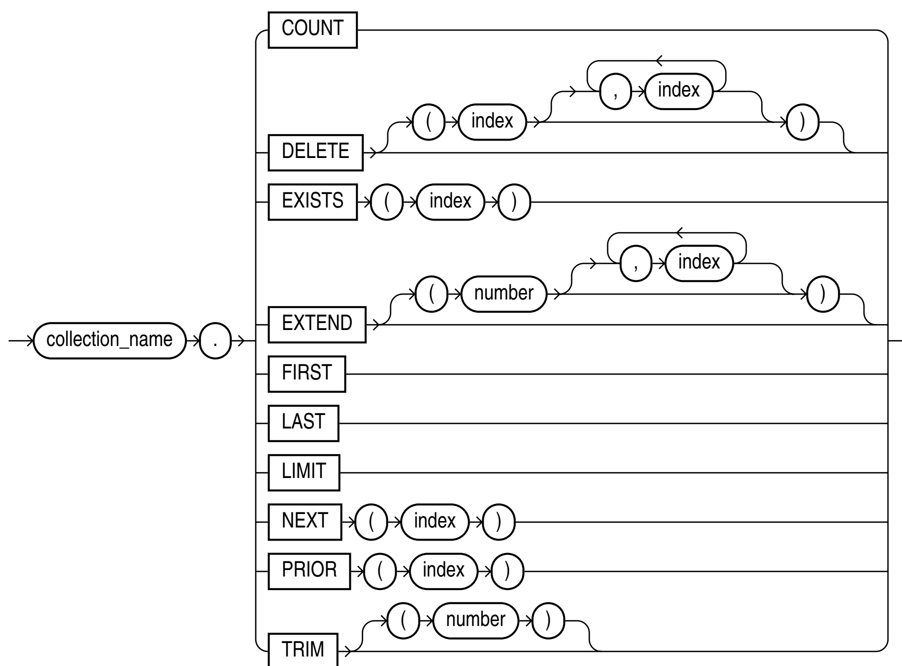
A collection method is a built-in PL/SQL subprogram that returns information about a collection or operates on a collection.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***collection_method_call*** **::=**




## Semantics

***collection_name***

The name of a collection.

**COUNT**

A function that returns the current number of elements in the collection, described in "COUNT Method" on page 5-23.

**DELETE**

A procedure that deletes elements from the collection, described in "DELETE Method" on page 5-30.

**Restriction on DELETE** If *collection_name* identifies a varray, you cannot specify indexes with DELETE.

### EXISTS

A function that returns TRUE if the *index*th element of the collection exists and FALSE otherwise. For more information, see "EXISTS Method" on page 5-23.

**Restriction on EXISTS** You cannot use EXISTS if *collection_name* identifies an associative array.

### EXTEND

A procedure that increases the size of the collection, described in "EXTEND Method" on page 5-27.

**Restriction on EXTEND** You cannot use EXTEND if *collection_name* identifies an associative array.

### FIRST

A function that returns the first subscript or key value in the collection, described in "FIRST and LAST Methods" on page 5-24.

### *index*

A numeric expression whose value has data type PLS_INTEGER or a data type that can be implicitly converted to PLS_INTEGER (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions" on page 3-29).

### LAST

A function that returns the last subscript or key value in the collection, described in "FIRST and LAST Methods" on page 5-24.

### LIMIT

A function that returns the maximum number of elements that the collection can have. If the collection has no maximum size, LIMIT returns NULL. (For an example, see "LIMIT Method" on page 5-24.

### NEXT

A function that returns the subscript that succeeds index *n*. If *n* has no successor, NEXT(*n*) returns NULL. For more information, see "PRIOR and NEXT Methods" on page 5-25.

### PRIOR

A function that returns the subscript that precedes index *n* in a collection. If *n* has no predecessor, PRIOR(*n*) returns NULL. For more information, see "PRIOR and NEXT Methods" on page 5-25.

### TRIM

A procedure that deletes elements from the end of a collection, described in "TRIM Method" on page 5-28.

**Restriction on TRIM** You cannot use TRIM if *collection_name* identifies an associative array.

## Examples


- Example 5–28, "Checking Whether a Collection Element EXISTS" on page 5-23
- Example 5–29, "Counting Collection Elements with COUNT" on page 5-23
- Example 5–30, "Checking the Maximum Size of a Collection with LIMIT" on page 5-24
- Example 5–31, "FIRST and LAST Methods" on page 5-25
- Example 5–32, "PRIOR and NEXT Methods" on page 5-26
- Example 5–34, "EXTEND Method" on page 5-27
- Example 5–35, "TRIM Method" on page 5-29
- Example 5–37, "DELETE Method" on page 5-30


## Related Topics

**In this chapter:**


- "Collection" on page 13-23


**In other chapters:**


- "Collection Methods" on page 5-22

# Comment

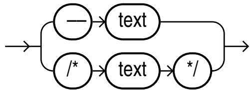
A comment is text that the PL/SQL compiler ignores. Its primary purpose is to document code, but you can also use it to disable obsolete or unfinished pieces of code (that is, you can turn the code into comments). PL/SQL has both single-line and multiline comments.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***comment*** **::=**




## Semantics

**--**

Turns the rest of the line into a single-line comment. Any text that wraps to the next line is not part of the comment.

> **Caution:** Do not put a single-line comment in a PL/SQL block to be processed dynamically by an Oracle Precompiler program. The Oracle Precompiler program ignores end-of-line characters, which means that a single-line comment ends when the block ends.

**/***

Begins a comment, which can span multiple lines.

***/**

Ends a comment.

***text***

Any text.

**Restriction on *text*** In a multiline comment, `text` cannot include the multiline comment delimiter `/*` or `*/`. Therefore, one multiline comment cannot contain another multiline comment. However, a multiline comment can contain a single-line comment.

## Examples

- Example 2–5, "Single-Line Comments" on page 2-11

- Example 2–6, "Multiline Comments" on page 2-11

### Related Topics

- "Comments" on page 2-10

# Constant

A constant holds a value that does not change.

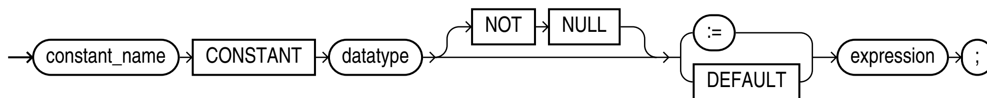
A constant declaration specifies the name, data type, and value of the constant and allocates storage for it. The declaration can also impose the NOT NULL constraint.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

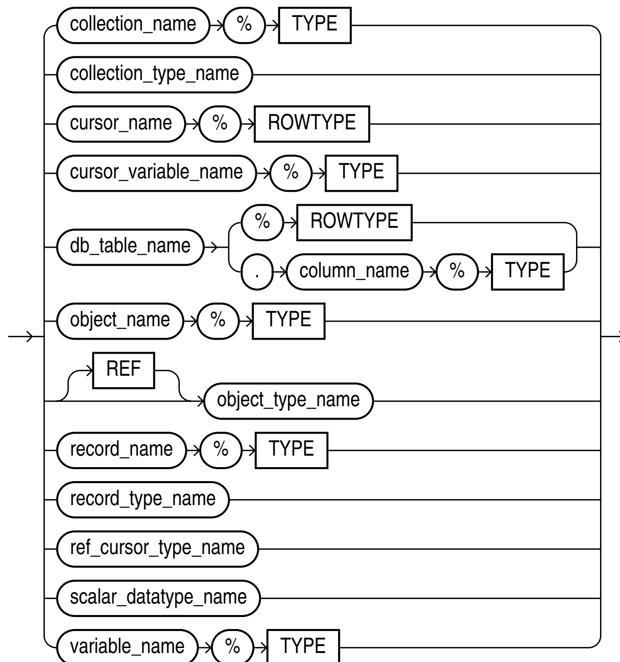
## Syntax

### *constant_declaration* ::=




See *expression ::=* on page 13-58.

### *datatype* ::=




## Semantics

### *collection_name*

The name of a collection.

***collection_type_name***
A user-defined collection type defined using the data type specifier TABLE or VARRAY.

***constant_name***
The name of the constant that you are declaring.

***cursor_name***
The name of an explicit cursor.

***cursor_variable_name***
The name of a PL/SQL cursor variable.

***db_table_name***
The name of a database table or view, which must be accessible when the declaration is elaborated.

***column_name***
The name of a column in *db_table_name*.

***expression***
An expression whose value has a data type compatible with *datatype* (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions"). When *constant_declaration* is elaborated, the value of *expression* is assigned to *constant_name*. This value is the initial value of the constant.

**NOT NULL**
Prevents the program from assigning the value NULL to the constant.

PL/SQL subtypes NATURALN, POSITIVEN, and SIMPLE_INTEGER are predefined as NOT NULL. When declaring a constant of one of these subtypes, you can omit NOT NULL.

If you declare a NOT NULL constant, either by specifying NOT NULL or by specifying a subtype that is predefined as NOT NULL, you must also specify an initial value for the constant (see ***expression***).

***object_name***
The name of an instance of a type.

***record_name***
The name of a record.

***field_name***
The name of a field of *record_name*.

***record_type_name***
The name of a user-defined type that was defined with the data type specifier RECORD.

***ref_cursor_type_name***
The name of a user-defined type that was defined with the data type specifier REF CURSOR.

### %ROWTYPE

See "%ROWTYPE Attribute" on page 13-116.

### *scalar_datatype_name*

The name of a predefined scalar data type, including any qualifiers for size, precision, and character or byte semantics. For information about predefined scalar data types, see "Predefined PL/SQL Scalar Data Types and Subtypes" on page 3-2.

### %TYPE

See "%TYPE Attribute" on page 13-127.

## Examples

- Example 2–8, "Constant Declarations" on page 2-13
- Example 2–9, "Variable and Constant Declarations with Initial Values" on page 2-14

## Related Topics

**In this chapter:**

- "%ROWTYPE Attribute" on page 13-116
- "%TYPE Attribute" on page 13-127
- "Variable" on page 13-131

**In other chapters:**

- "Comments" on page 2-10
- "Constant Declarations" on page 2-13

# CONTINUE Statement

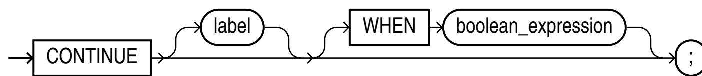
The CONTINUE statement exits the current iteration of a loop, either conditionally or unconditionally, and transfers control to the next iteration of either the current loop or an enclosing labeled loop.

Topics:

- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

***continue_statement* ::=**




See *boolean_expression ::=* on page 13-59.

## Semantics

***label***

A name that identifies either the current loop or an enclosing loop (see "Basic LOOP Statement" on page 13-16).

Without *label*, the CONTINUE statement transfers control to the next iteration of the current loop. With *label*, the CONTINUE statement transfers control to the next iteration of the loop that *label* identifies.

**WHEN *boolean_expression***

Without this clause, the CONTINUE statement exits the current iteration of the loop unconditionally. With this clause, the CONTINUE statement exits the current iteration of the loop if and only if the value of *boolean_expression* is TRUE.

## Usage

A CONTINUE statement can appear anywhere inside a loop, but not outside a loop.

A CONTINUE statement cannot cross a subprogram or method boundary.

If a CONTINUE statement exits a cursor FOR loop prematurely (for example, to exit an inner loop and transfer control to the next iteration of an outer loop), the cursor closes (in this context, CONTINUE works like GOTO).

> **Note:** As of Release 11.1, CONTINUE is a PL/SQL keyword. If your program invokes a subprogram named CONTINUE, you get a warning.

## Examples

- Example 4–12, "CONTINUE Statement in Basic LOOP Statement" on page 4-12
- Example 4–13, "CONTINUE WHEN Statement in Basic LOOP Statement" on page 4-12

## Related Topics

**In this chapter:**

- "EXIT Statement" on page 13-52
- "Expression" on page 13-58
- "Basic LOOP Statement" on page 13-16

**In other chapters:**

- "CONTINUE Statement" on page 4-11
- "CONTINUE WHEN Statement" on page 4-12

# Cursor FOR LOOP Statement

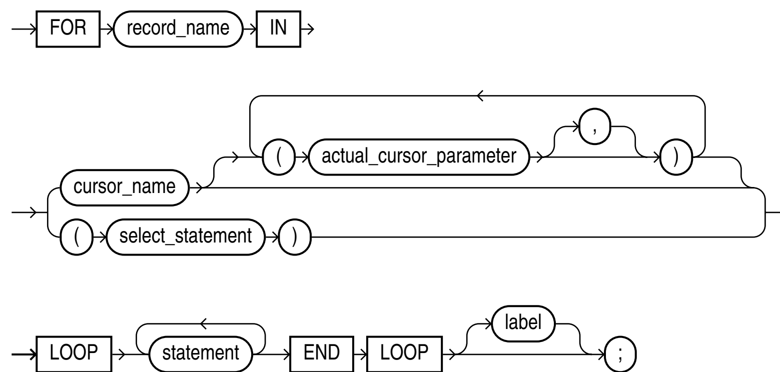
The cursor FOR LOOP statement implicitly declares its index as a record variable of the row type that a specified cursor returns and opens a cursor. With each iteration, the cursor FOR LOOP statement fetches a row from the result set into the record. When there are no more rows to fetch, the cursor FOR LOOP statement closes the cursor. The cursor also closes if a statement inside the loop transfers control outside the loop or if an exception is raised.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

### *cursor_for_loop_statement* ::=




See *statement ::=* on page 13-11.

## Semantics

#### *actual_cursor_parameter*

An actual parameter that corresponds to a formal parameter of the explicit cursor `cursor_name`. For more information, see "Explicit Cursors that Accept Parameters" on page 6-18.

#### *cursor_name*

The name of an explicit cursor that is not already open when the cursor FOR LOOP is entered.

#### *label*

A label that identifies *`cursor_for_loop_statement`* (see ***label*** on page 13-14). CONTINUE, EXIT, and GOTO statements can reference this label.

Labels improve readability, especially when LOOP statements are nested, but only if you ensure that the label at the end of the LOOP statement matches one of the labels at the beginning of the same statement (the compiler does not check).

#### *record_name*

An identifier for the index. The cursor FOR LOOP statement implicitly declares *record_name* as a record variable whose field names match the column names of the rows that *cursor_name* or *select_statement* returns.

The variable *record_name* is local to the cursor FOR LOOP statement. Statements inside the loop can reference *record_name* and its fields. Any calculated column must have an alias if you want to reference it inside the loop—for details, see "When Explicit Cursors Need Column Aliases" on page 6-17. Statements outside the loop cannot reference *record_name*. After the cursor FOR LOOP statement runs, *record_name* is undefined.

#### *select_statement*

A SQL SELECT statement (not a PL/SQL SELECT INTO statement). For this SELECT statement, PL/SQL declares, opens, fetches from, and closes an implicit cursor. However, because *select_statement* is not an independent statement, the implicit cursor is internal—you cannot reference it with the name SQL.

> **See Also:** *Oracle Database SQL Language Reference* for SELECT statement syntax

## Examples


- Example 6–7, "Fetching with a Cursor" on page 6-11
- Example 6–10, "Fetching Bulk Data with a Cursor" on page 6-13


## Related Topics

**In this chapter:**


- "Basic LOOP Statement" on page 13-16
- "CONTINUE Statement" on page 13-35
- "EXIT Statement" on page 13-52
- "Explicit Cursor" on page 13-54
- "FETCH Statement" on page 13-67
- "FOR LOOP Statement" on page 13-70
- "FORALL Statement" on page 13-73
- "OPEN Statement" on page 13-97
- "WHILE LOOP Statement" on page 13-133


**In other chapters:**


- "Looping Through Multiple Rows (Cursor FOR LOOP Statement)" on page 6-21

# Cursor Variable

A cursor variable points to the unnamed work area in which the database stores processing information when it runs a multiple-row query. With this pointer to the work area, you can access its information, and process the rows of the query individually.

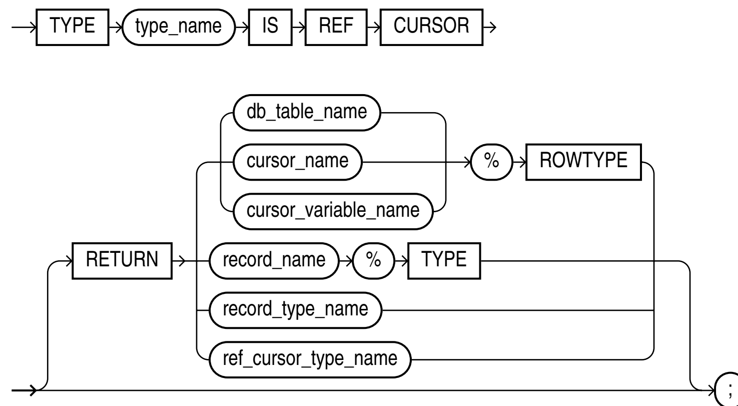
To create a cursor variable, define a `REF CURSOR` type, and then declare the cursor variable to be of that type. Declaring a cursor variable creates a pointer, not an item. A cursor variable holds the address of an item instead of the item itself (like a C or Pascal pointer).

Topics:

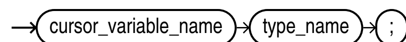
- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

***ref_cursor_type_definition*** **::=**




***cursor_variable_declaration*** **::=**




## Semantics

***cursor_name***

The name of an explicit cursor.

***cursor_variable_name***

The name of a cursor variable.

***db_table_name***

The name of a database table or view, which must be accessible when the declaration is elaborated.

***record_name***

The name of a user-defined record.

***record_type_name***

The name of a user-defined type that was defined with the data type specifier RECORD.

**RETURN**

Specifies the data type of the value that the cursor variable returns.

Specify RETURN to define a strong REF CURSOR type. Omit RETURN to define a weak REF CURSOR type. For information about strong and weak REF CURSOR types, see "Cursor Variable Creation" on page 6-26.

**%ROWTYPE**

See "%ROWTYPE Attribute" on page 13-116.

**%TYPE**

See "%TYPE Attribute" on page 13-127.

***type_name***

The name of the REF CURSOR type that you are defining and the type of the cursor variable that you are declaring.

## Usage

A REF CURSOR type definition can appear either in the declarative part of a block, subprogram, package specification, or package body (in which case it is local to the block, subprogram, or package) or in the "CREATE TYPE Statement" on page 14-69 (in which case it is a standalone stored type).

A cursor variable declaration can appear only in the declarative part of a block, subprogram, or package body (not in a package specification).

Cursor variables are subject to the restrictions in "Cursor Variable Restrictions" on page 6-34.

## Examples


- Example 6–5, "Explicit Cursor Declaration" on page 6-9
- Example 6–7, "Fetching with a Cursor" on page 6-11
- Example 6–10, "Fetching Bulk Data with a Cursor" on page 6-13
- Example 6–28, "Cursor Variable as Parameter" on page 6-28
- Example 6–30, "Stored Procedure to Open a Cursor Variable" on page 6-30
- Example 6–31, "Stored Procedure to Open Cursor Variables with Different Queries" on page 6-30
- Example 6–32, "Cursor Variable with Different Return Types" on page 6-31
- Example 6–38, "Cursor Expression" on page 6-35

## Related Topics

**In this chapter:**


- "CLOSE Statement" on page 13-21
- "Named Cursor Attribute" on page 13-93
- "Explicit Cursor" on page 13-54
- "FETCH Statement" on page 13-67
- "OPEN FOR Statement" on page 13-98
- "%ROWTYPE Attribute" on page 13-116
- "%TYPE Attribute" on page 13-127


**In other chapters:**


- "Cursor Variables" on page 6-25

# DELETE Statement Extension

The PL/SQL extension to the *where_clause* of the SQL DELETE statement lets you specify a CURRENT OF clause, which restricts the DELETE statement to the current row of the specified cursor. For information about the CURRENT OF clause, see "UPDATE Statement Extensions" on page 13-129.

**See Also:** *Oracle Database SQL Language Reference* for the syntax of the SQL DELETE statement

# EXCEPTION_INIT Pragma

The EXCEPTION_INIT pragma associates a user-defined exception name with an Oracle Database error number. You can intercept any Oracle Database error number and write an exception handler for it, instead of using the OTHERS handler.

Topics:

- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

***exception_init_pragma* ::=**

```
→ PRAGMA → EXCEPTION_INIT → ( → exception_name → , → error_number → ) → ;
```

## Semantics

***error_number***

A valid Oracle Database error number not already associated with a user-defined exception. For a list of Oracle Database error messages and information about them, including their numbers, see *Oracle Database Error Messages*.

---

**Note:** These are the same error numbers (always negative) returned by the"SQLCODE Function" on page 13-124.

---

***exception_name***

The name of a user-defined exception.

## Usage

A EXCEPTION_INIT pragma can appear only in the same declarative part as its associated exception, anywhere after the exception declaration.

## Examples

- Example 11–7, "PRAGMA EXCEPTION_INIT" on page 11-11
- Example 12–11, "FORALL Statement and SQL%BULK_EXCEPTIONS" on page 12-18

## Related Topics

**In this chapter:**

- "Exception" on page 13-45
- "Exception Handler" on page 13-47

- "SQLCODE Function" on page 13-124
- "SQLERRM Function" on page 13-125


**In other chapters:**


- "Associating Exceptions with Numbers" on page 11-10

# Exception

An exception declaration declares a user-defined exception.

You must raise a user-defined exception explicitly with either a RAISE statement or the DBMS_STANDARD.RAISE_APPLICATION_ERROR procedure. The latter lets you associate an error message with the user-defined exception.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***exception_declaration*** **::=**

```
→ exception_name → EXCEPTION → ;
```

## Semantics

***exception_name***

The name of the exception that you are declaring.

---

**Caution:** Using the name of a predefined exception for *exception_name* is not recommended. For details, see "Redeclaring Predefined Exceptions" on page 11-12. For the names of a predefined exceptions, see Table 11–2.

---

## Examples

- Example 2–24, "Block with Multiple and Duplicate Labels" on page 2-22
- Example 5–35, "TRIM Method" on page 5-29
- Example 5–38, "Collection Exceptions" on page 5-31
- Example 6–40, "ROLLBACK Statement" on page 6-39
- Example 7–13, "Validation Checks Guarding Against SQL Injection" on page 7-16
- Example 8–1, "Declaring, Defining, and Invoking a Simple PL/SQL Procedure" on page 8-3
- Example 10–6, "Creating emp_admin Package" on page 10-11
- Example 11–3, "Anonymous Block with Exception Handlers" on page 11-5
- Example 11–6, "Scope of Exceptions" on page 11-10
- Example 11–12, "Reraising an Exception" on page 11-16
- Example 12–10, "Rollbacks with FORALL Statement" on page 12-17

- Example 12–11, "FORALL Statement and SQL%BULK_EXCEPTIONS" on page 12-18

## Related Topics

**In this chapter:**

- "Exception Handler" on page 13-47
- "RAISE Statement" on page 13-105

**In other chapters:**

- "User-Defined Exceptions" on page 11-9
- "Exception Declarations" on page 11-9

# Exception Handler

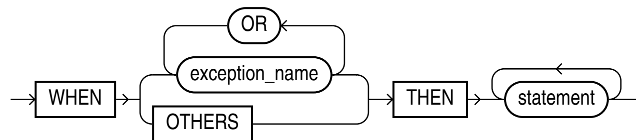
An exception handler processes a raised exception (run-time error or warning condition). The exception can be either predefined or user-defined. The run-time system raises predefined exceptions automatically, but you must raise user-defined exceptions explicitly with either a `RAISE` statement or the procedure `DBMS_STANDARD.RAISE_APPLICATION_ERROR`. The latter lets you associate an error message with the user-defined exception.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***exception_handler* ::=**




See *statement ::=* on page 13-11.

## Semantics

#### *exception_name*

The name of either a predefined exception (see Table 11–2) or a user-defined exception (see "Exception" on page 13-45).

If a specified exception is raised, the associated statement runs.

#### OTHERS

Specifies all exceptions not explicitly named in the exception-handling part of the block. `WHEN OTHERS` is optional. If used, it must be the last exception handler in the exception-handling part of the block.

If an exception that is not explicitly named is raised, the associated statement runs.

## Examples

- Example 2–24, "Block with Multiple and Duplicate Labels" on page 2-22
- Example 5–35, "TRIM Method" on page 5-29
- Example 5–38, "Collection Exceptions" on page 5-31
- Example 6–40, "ROLLBACK Statement" on page 6-39
- Example 7–13, "Validation Checks Guarding Against SQL Injection" on page 7-16

- Example 8–1, "Declaring, Defining, and Invoking a Simple PL/SQL Procedure" on page 8-3
- Example 10–6, "Creating emp_admin Package" on page 10-11
- Example 11–3, "Anonymous Block with Exception Handlers" on page 11-5
- Example 11–6, "Scope of Exceptions" on page 11-10
- Example 11–12, "Reraising an Exception" on page 11-16
- Example 12–10, "Rollbacks with FORALL Statement" on page 12-17
- Example 12–11, "FORALL Statement and SQL%BULK_EXCEPTIONS" on page 12-18

## Related Topics

**In this chapter:**


- "Block" on page 13-8
- "EXCEPTION_INIT Pragma" on page 13-43
- "Exception" on page 13-45
- "RAISE Statement" on page 13-105
- "SQLCODE Function" on page 13-124
- "SQLERRM Function" on page 13-125


**In other chapters:**


- "Exception Handlers" on page 11-16

# EXECUTE IMMEDIATE Statement

The EXECUTE IMMEDIATE statement builds and runs a dynamic SQL statement in a single operation. The EXECUTE IMMEDIATE statement is the means by which native dynamic SQL processes most dynamic SQL statements.

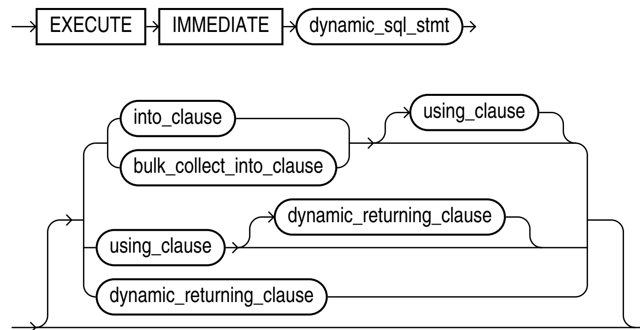
> **Caution:** When using dynamic SQL, be aware of SQL injection, a security risk. For more information about SQL injection, see "SQL Injection" on page 7-9.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

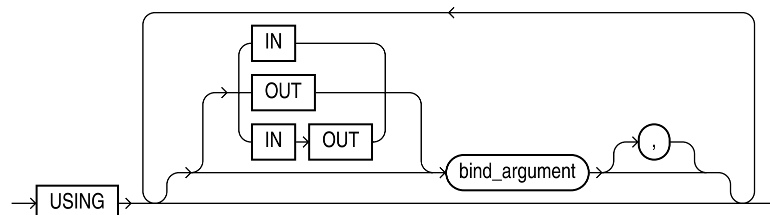
***execute_immediate_statement*** **::=**




See:

- *bulk_collect_into_clause ::=* on page 13-113
- *dynamic_returning_clause ::=* on page 13-113
- *into_clause ::=* on page 13-113

***using_clause*** **::=**



## Semantics

### *bind_argument*

An expression whose value replaces its corresponding placeholder in *dynamic_sql_stmt* at run time.

Every placeholder in *dynamic_sql_stmt* must be associated with a *bind_argument* in the USING clause or RETURNING INTO clause (or both) or with a define variable in the INTO clause.

You can run *dynamic_sql_stmt* repeatedly using different values for the bind arguments. You incur some overhead, because EXECUTE IMMEDIATE prepares the dynamic string before every execution.

**Restriction on *bind_argument*** The value of *bind_argument* cannot be TRUE, FALSE, or NULL. To pass the value NULL to the dynamic SQL statement, use an uninitialized variable where you want to use NULL, as in "Uninitialized Variable for NULL in USING Clause" on page 7-4.

### *bulk_collect_into_clause*

Specifies one or more collections in which to store the rows that the statement returns. For more information about this clause, see "RETURNING INTO Clause" on page 13-113.

**Restriction on *bulk_collect_into_clause*** Use if and only if *dynamic_sql_stmt* can return multiple rows.

### *dynamic_returning_clause*

Returns the column values of the rows affected by the dynamic SQL statement, in either individual variables or records. For more information about this clause, see "RETURNING INTO Clause" on page 13-113.

**Restriction on *dynamic_returning_clause*** Use if and only if *dynamic_sql_stmt* has a RETURNING INTO clause.

### *dynamic_sql_stmt*

A string literal, string variable, or string expression that represents a SQL statement. Its type must be either CHAR, VARCHAR2, or CLOB.

### IN, OUT, IN OUT

Parameter modes of bind arguments. An IN bind argument passes its value to *dynamic_sql_stmt*. An OUT bind argument stores a value that *dynamic_sql_stmt* returns. An IN OUT bind argument passes its initial value to *dynamic_sql_stmt* and stores a value that *dynamic_sql_stmt* returns. The default parameter mode for *bind_argument* is IN.

### *into_clause*

Specifies the variables or record in which to store the column values that the statement returns. For more information about this clause, see "RETURNING INTO Clause" on page 13-113.

**Restriction on *into_clause*** Use if and only if *dynamic_sql_stmt* returns a single row.

#### *using_clause*

Specifies bind arguments.

##### Restrictions on *using_clause*

- Use if and only if *dynamic_sql_stmt* includes placeholders for bind arguments.
- If *dynamic_sql_stmt* has a RETURNING INTO clause, *using_clause* can contain only IN bind arguments. The bind arguments in the RETURNING INTO clause are OUT bind arguments by definition.

## Examples

- Example 7–1, "Invoking a Subprogram from a Dynamic PL/SQL Block" on page 7-3
- Example 7–2, "Unsupported Data Type in Native Dynamic SQL" on page 7-4
- Example 7–3, "Uninitialized Variable for NULL in USING Clause" on page 7-4
- Example 7–5, "Repeated Placeholder Names in Dynamic PL/SQL Block" on page 7-6

## Related Topics

##### In this chapter:

- "RETURNING INTO Clause" on page 13-113

##### In other chapters:

- "EXECUTE IMMEDIATE Statement" on page 7-2
- "DBMS_SQL Package" on page 7-6

# EXIT Statement

The EXIT statement exits the current iteration of a loop, either conditionally or unconditionally, and transfers control to the end of either the current loop or an enclosing labeled loop.

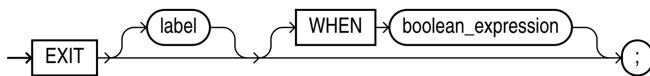
An EXIT statement can appear anywhere inside a loop, but not outside a loop.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***exit_statement* ::=**




See *boolean_expression ::=* on page 13-59.

## Semantics

***label***

A *label* that identifies either the current loop or an enclosing loop (see "Basic LOOP Statement" on page 13-16).

Without *label*, the EXIT statement transfers control to the end of the current loop. With *label*, the EXIT statement transfers control to the end of the loop identified by *label*.

**WHEN *boolean_expression***

Without this clause, the EXIT statement exits the current iteration of the loop unconditionally. With this clause, the EXIT statement exits the current iteration of the loop if and only if the value of *boolean_expression* is TRUE.

## Examples

- Example 4–9, "Basic LOOP Statement with EXIT Statement" on page 4-10
- Example 4–10, "Basic LOOP Statement with EXIT WHEN Statement" on page 4-10
- Example 4–11, "Nested, Labeled Basic LOOP Statements with EXIT WHEN Statements" on page 4-11
- Example 4–26, "EXIT in FOR LOOP" on page 4-20
- Example 4–27, "EXIT with Label in FOR LOOP" on page 4-20

## Related Topics

**In this chapter:**


- "CONTINUE Statement" on page 13-35
- "Basic LOOP Statement" on page 13-16
- "EXIT Statement" on page 4-10
- "EXIT WHEN Statement" on page 4-10

# Explicit Cursor

An **explicit cursor** is a named pointer to a private SQL area that stores information for processing a specific SQL DML statement—usually one that returns multiple rows. You can use an explicit cursor to retrieve the rows of a result set one at a time.

You must declare and define an explicit cursor before using it. You can either declare and define it at the same time, or you can declare it first and then define it later in the same block, subprogram, or package.

---

**Note:** An explicit cursor declared in a package specification is affected by the AUTHID clause of the package. For more information, see "CREATE PACKAGE Statement" on page 14-42.

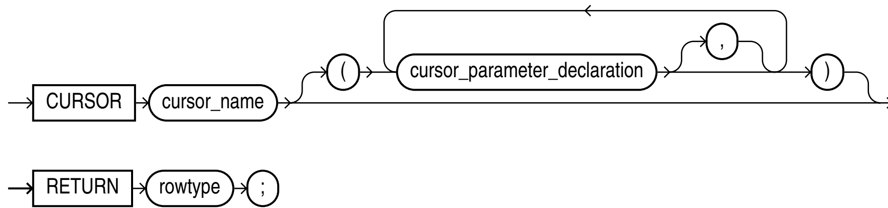
---

Topics:

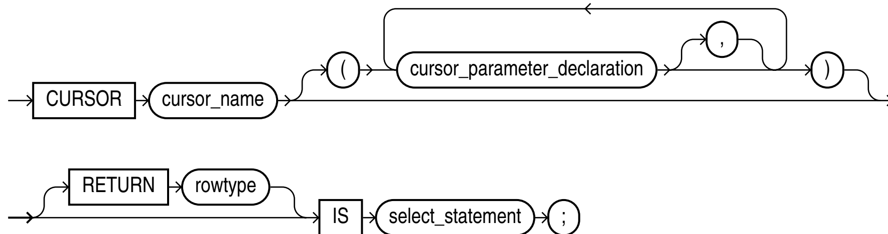
- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

***cursor_declaration*** **::=**

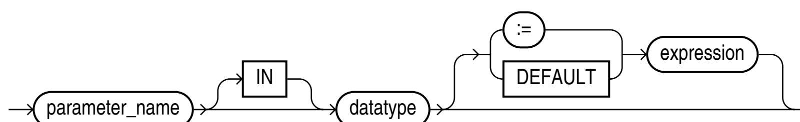



***cursor_definition*** **::=**




See *rowtype ::=* on page 13-55.

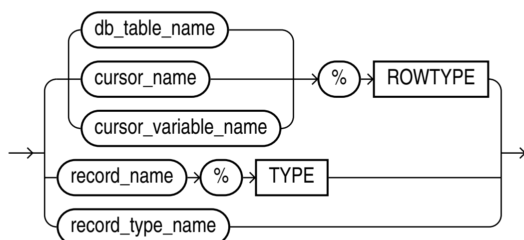
***cursor_parameter_declaration* ::=**




See:

- *datatype ::=* on page 13-32
- *expression ::=* on page 13-58

***rowtype* ::=**




## Semantics

### *cursor_declaration*

Declares an explicit cursor but does not define it. The definition must appear later in the same block, subprogram, or package as the declaration.

An explicit cursor declaration is also called a **cursor specification**, or **cursor spec**.

### *cursor_definition*

Either defines an explicit cursor that was declared earlier or both declares and defines an explicit cursor.

### *cursor_name*

In *`cursor_declaration`*, *`cursor_name`* is the name that you are giving to the explicit cursor that you are declaring. This name can be any identifier except the reserved word `SQL`.

In *`cursor_definition`* and *`rowtype`*, *`cursor_name`* is the name of an explicit cursor.

### *cursor_variable_name*

The name of a record variable.

### *datatype*

The data type of the formal cursor parameter that you are declaring.

**Restriction on *datatype*** This *`datatype`* cannot have constraints (that is, precision and scale for a number, or length for a string).

### *db_table_name*

The name of a database table or view that is accessible when the cursor declaration is elaborated.

#### *expression*

An *expression* as in "Expression" on page 13-58. When the cursor declaration is elaborated, the value of *expression* is assigned to the parameter. The value and the parameter must have compatible data types.

> **Note:** If you supply an actual parameter for *parameter_name* when you open the cursor, then *expression* is not evaluated.

#### *parameter_name*

The name of the formal cursor parameter that you are declaring. This `IN` parameter is local to the cursor. It must appear in *select_statement*, and can appear anywhere in *select_statement* that a constant can appear. When the cursor opens, *select_statement* uses the parameter value. For more information, see "Explicit Cursors that Accept Parameters" on page 6-18.

#### *record_name*

The name of a record.

#### *record_type_name*

The name of a type that was defined with the data type specifier `RECORD`.

#### RETURN *rowtype*

Specifies the data type of the row that the cursor returns. The columns of this row must match the columns of the row that *select_statement* returns.

#### %ROWTYPE

The data type of a row of *db_table_name*, *cursor_name*, or *cursor_variable_name* (whichever is specified). For more information, see "%ROWTYPE Attribute" on page 13-116.

#### *select_statement*

A SQL `SELECT` statement (not a PL/SQL `SELECT INTO` statement). If the cursor has formal parameters, each parameter must appear in *select_statement*. The *select_statement* can also reference other PL/SQL variables in its scope.

> **See:** *Oracle Database SQL Language Reference* for `SELECT` statement syntax

#### %TYPE

See "%TYPE Attribute" on page 13-127.

### Usage

After declaring an explicit cursor, you can open it (with the `OPEN` statement), fetch rows one at a time from the result set (with the `FETCH` statement), and then close it (with the `CLOSE` statement). After closing the cursor, you can neither fetch records from the result set nor see the cursor attribute values.

Cursors follow the same scoping rules as variables (see "Scope and Visibility of Identifiers" on page 2-18).

You cannot assign a value to an explicit cursor or use it in an expression.

## Examples

- Example 6–5, "Explicit Cursor Declaration" on page 6-9
- Example 6–17, "Passing Parameters to Explicit Cursors" on page 6-19
- Example 6–44, "FOR UPDATE Cursor in CURRENT OF Clause of UPDATE Statement" on page 6-45
- Example 6–46, "Trying to Fetch with FOR UPDATE Cursor After COMMIT Statement" on page 6-46

## Related Topics

**In this chapter:**

- "CLOSE Statement" on page 13-21
- "Cursor Variable" on page 13-39
- "FETCH Statement" on page 13-67
- "Named Cursor Attribute" on page 13-93
- "OPEN Statement" on page 13-97
- "%ROWTYPE Attribute" on page 13-116
- "%TYPE Attribute" on page 13-127

**In other chapters:**

- "Explicit Cursors" on page 6-9
- "Queries" on page 6-19
- "SELECT FOR UPDATE and FOR UPDATE Cursors" on page 6-44

# Expression

An expression is an arbitrarily complex combination of operands (variables, constants, literals, operators, function calls, and placeholders) and operators. The simplest expression is a single variable.

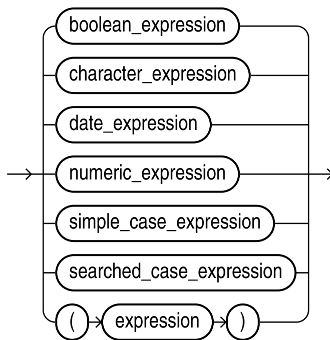
The PL/SQL compiler determines the data type of an expression from the types of the operands and operators that comprise the expression. Every time the expression is evaluated, a single value of that type results.

Topics:


- Syntax
- Semantics
- Examples
- Related Topics


## Syntax

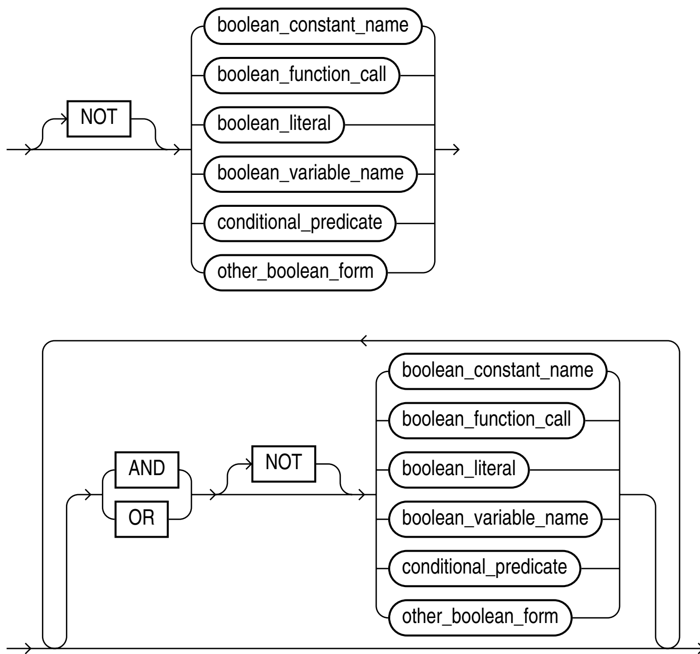
***expression* ::=**




See:

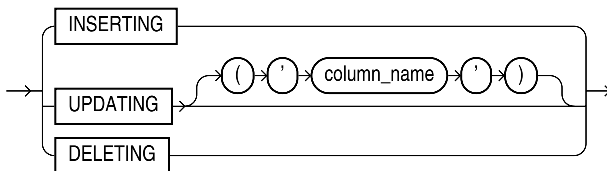

- *boolean_expression ::=* on page 13-59
- *character_expression ::=* on page 13-60
- *date_expression ::=* on page 13-60
- *numeric_expression ::=* on page 13-60
- *searched_case_expression ::=* on page 13-62
- *simple_case_expression ::=* on page 13-61

***boolean_expression* ::=**

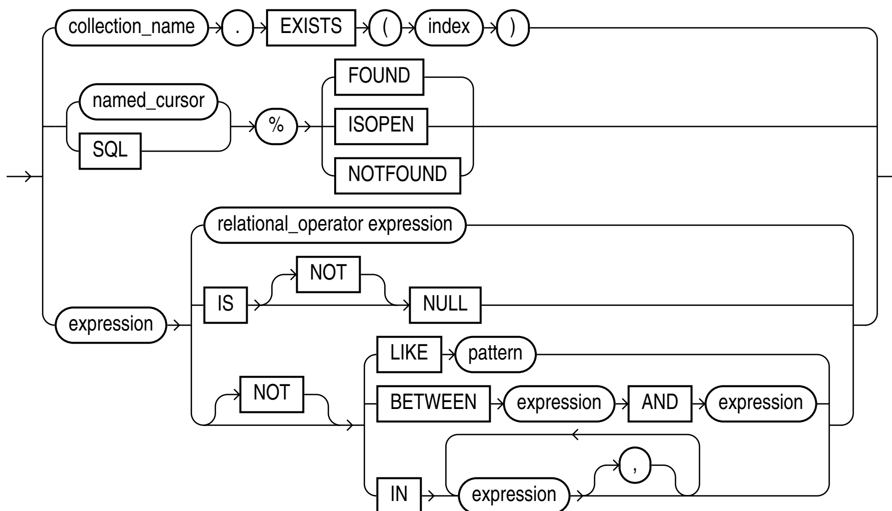



See *function_call ::=* on page 13-61.

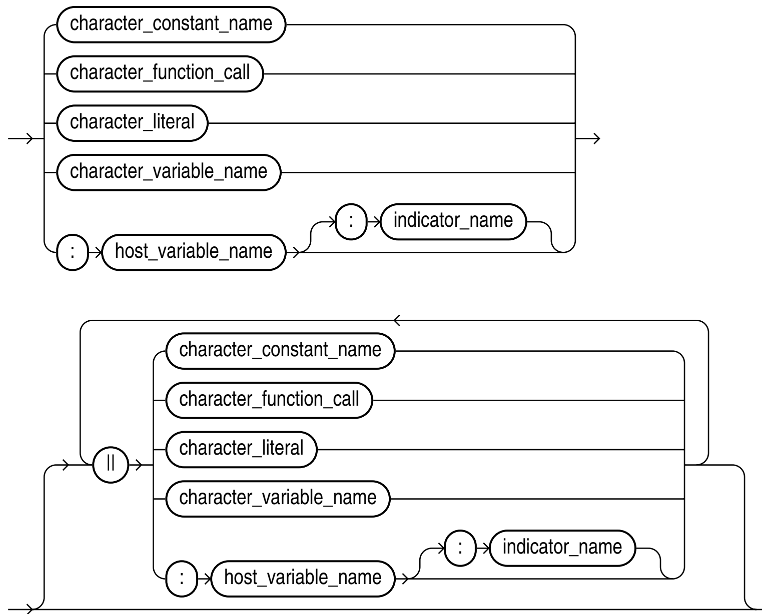
***conditional_predicate* ::=**




***other_boolean_form* ::=**

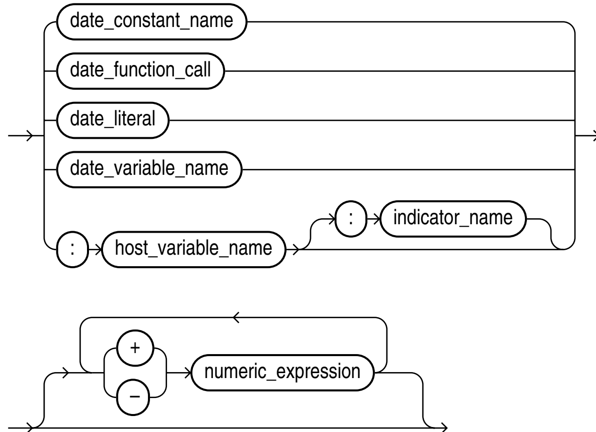


***character_expression*** **::=**



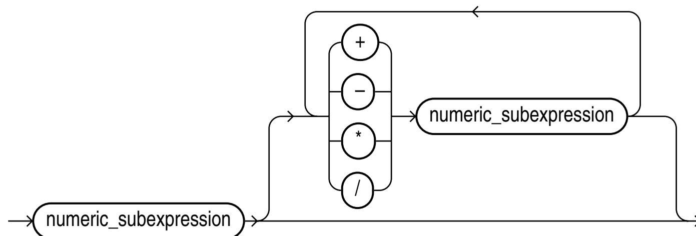

See *function_call ::=* on page 13-61.

***date_expression*** **::=**

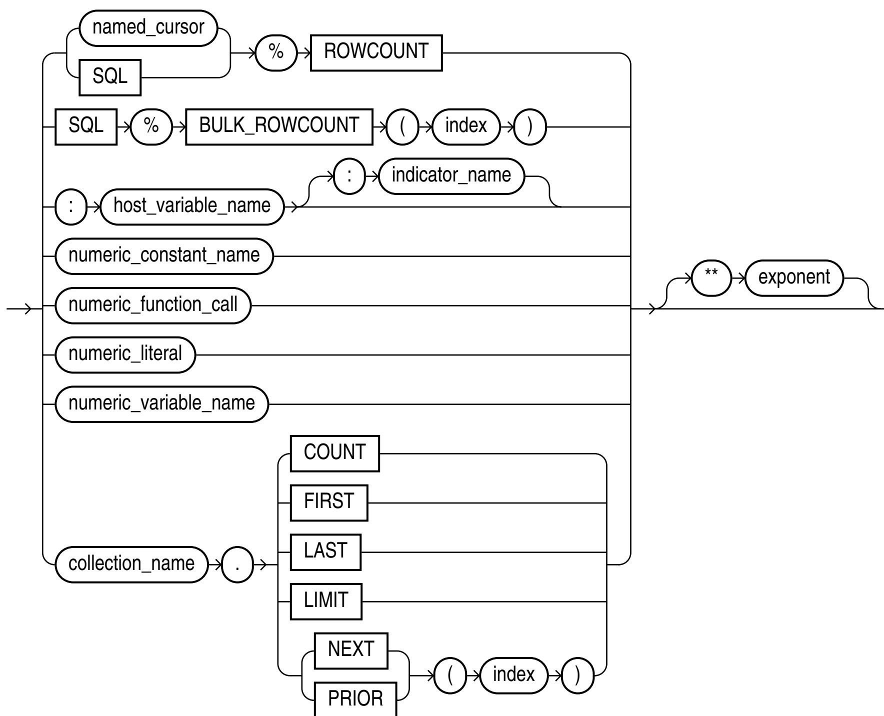



See *function_call ::=* on page 13-61.

***numeric_expression*** **::=**



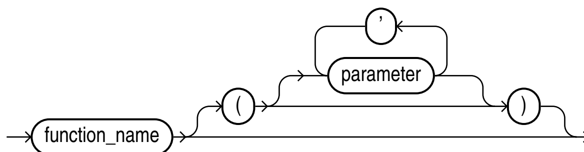
***numeric_subexpression* ::=**



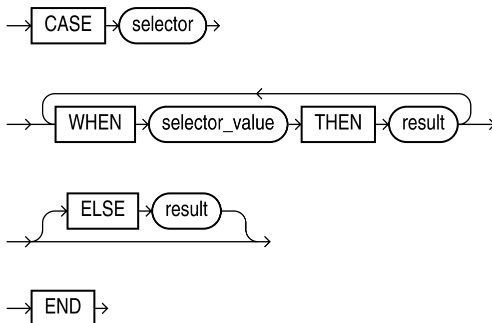

See:

- *function_call ::=* on page 13-61
- *named_cursor ::=* on page 13-93

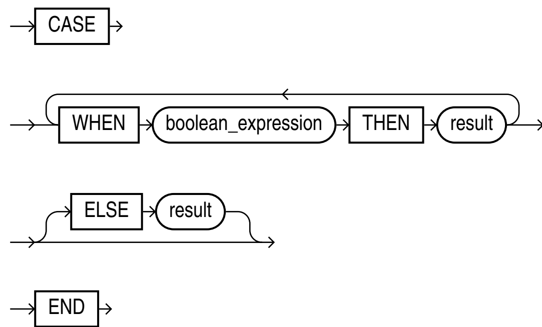
***function_call* ::=**




***simple_case_expression* ::=**



***searched_case_expression*** **::=**




See *boolean_expression ::=* on page 13-59.

## Semantics

### BETWEEN

See "BETWEEN Operator" on page 2-37.

### *boolean_constant_name*

The name of a constant of type `BOOLEAN`.

### *boolean_expression*

An expression whose value is `TRUE`, `FALSE`, or `NULL`. For more information, see "BOOLEAN Expressions" on page 2-39.

**Restriction on *boolean_expression*** Because SQL has no data type equivalent to `BOOLEAN`, you cannot:

- Assign a `BOOLEAN` expression to a database table column
- Select or fetch the value of a database table column into a `BOOLEAN` variable
- Use a `BOOLEAN` expression in a SQL statement, built-in SQL function, or PL/SQL function invoked from a SQL statement

### *boolean_function_call*

A call to a function that returns a `BOOLEAN` value.

### *boolean_literal*

The predefined value `TRUE`, `FALSE`, or `NULL`.

### *boolean_variable_name*

The name of a variable of type `BOOLEAN`.

### %BULK_ROWCOUNT

An attribute of the implicit cursor `SQL` for use with the `FORALL` statement. See **SQL%BULK_ROWCOUNT** on page 13-85.

### *character_constant_name*

The name of a constant that stores a character value.

***character_expression***

An expression whose value is a character or character string.

***character_function_call***

A call to a function that returns either a character value or a value that can be implicitly converted to a character value.

***character_literal***

A literal whose value is either a character value or a value that can be implicitly converted to a character value.

***character_variable_name***

The name of a variable that stores a character value.

***collection_name***

The name of a collection.

***date_constant_name***

The name of a constant that stores a date value.

***date_expression***

An expression that returns a datetime value.

***date_function_call***

A function call that returns either a date value or a value that can be implicitly converted to a date value.

***date_literal***

A literal whose value is either a date value or a value that can be implicitly converted to a date value.

***date_variable_name***

The name of a variable that stores a date value.

**EXISTS, COUNT, FIRST, LAST, LIMIT, NEXT, PRIOR**

Collection methods described in "Collection Method Call" on page 13-27.

***exponent***

An expression whose value is numeric.

**%FOUND, %ISOPEN, %NOTFOUND, %ROWCOUNT**

Cursor attributes described in "Implicit Cursor Attribute" on page 13-84 and "Named Cursor Attribute" on page 13-93.

***host_variable_name***

The name of a variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. Do not put space between the colon (:) and *host_variable_name*.

#### IN
See "IN Operator" on page 2-38.

#### *index*
A numeric expression whose value is of type `PLS_INTEGER` or a value that can be implicitly converted to `PLS_INTEGER` (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions" on page 3-29).

#### *indicator_name*
The name of an indicator variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. An indicator variable indicates the value or condition of its associated host variable (for example, in the Oracle Precompiler environment, an indicator variable can a detect null or truncated value in an output host variable). Do not put space between *`host_variable_name`* and the colon (:) or between the colon and *`indicator_name`*. This is correct:

```
:host_variable_name:indicator_name
```

#### INSERTING, UPDATING, DELETING
See "Conditional Predicates for Detecting the DML Operation that Fired a Trigger" on page 9-24.

#### IS [NOT] NULL
See "IS [NOT] NULL Operator" on page 2-36.

#### LIKE *pattern*
See "LIKE Operator" on page 2-36.

#### NOT, AND, OR
See "Logical Operators" on page 2-29.

#### *numeric_constant_name*
The name of a constant that stores a numeric value.

#### *numeric_expression*
An expression that returns a numeric value.

#### *numeric_function_call*
A call to a function that returns either a numeric value or a value that can be implicitly converted to a numeric value.

#### *numeric_literal*
A literal whose value is either a numeric value or a value that can be implicitly converted to a numeric value.

#### *numeric_variable_name*
The name of variable that stores a numeric value.

#### *parameter* [, *parameter* ]...
List of actual parameters for the function being called. The data type of each actual parameter must be compatible with the data type of the corresponding formal parameter (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions"). If the

function specifies an initial value for a parameter, you can omit that parameter from the parameter list.

If the function has no parameters, or specifies an initial value for every parameter, you can either omit the parameter list or specify an empty parameter list.

> **See Also:** "Positional, Named, and Mixed Notation for Actual Parameters" on page 8-12

#### *relational_operator*

See "Relational Operators" on page 2-34.

#### SQL

The implicit cursor associated with the most recently run SELECT or DML statement. For more information, see "Implicit Cursors" on page 6-6.

#### +, -, /, *, **

Addition, subtraction, division, multiplication, and exponentiation operators.

#### ||

The concatenation operator, which appends one string operand to another. For more information, see "Concatenation Operator" on page 2-26.

#### *selector*

An expression of any PL/SQL type except BLOB, BFILE, or a user-defined type. The *selector* is evaluated once.

#### WHEN { *selector_value* | *boolean_expression* } THEN *result*

The *selector_value*s or *boolean_expression*s are evaluated sequentially. If a *selector_value* is the value of *selector*, or if the value of a *boolean_expression* is TRUE, the *result* associated with that *selector_value* or *boolean_expression* is returned. Subsequent *selector_value*s or *boolean_expression*s are not evaluated.

A *selector_value* can be of any PL/SQL type except BLOB, BFILE, an ADT, a PL/SQL record, an associative array, a varray, or a nested table.

> **See Also:**
>
> - "Simple CASE Statement" on page 4-7
> - "Searched CASE Statement" on page 4-7

#### ELSE *result*

In the simple CASE expression, *result* is returned if and only if no *selector_value* has the same value as *selector*.

In the searched CASE statement, *result* is returned if and only if no *boolean_expression* has the value TRUE.

If you omit the ELSE clause, the case expression returns NULL.

## Examples

- Example 2–29, "Concatenation Operator" on page 2-26
- Example 2–30, "Concatenation Operator with NULL Operands" on page 2-26

- Example 2–31, "Controlling Evaluation Order with Parentheses" on page 2-27
- Example 2–32, "Expression with Nested Parentheses" on page 2-27
- Example 2–33, "Improving Readability with Parentheses" on page 2-28
- Example 2–34, "Operator Precedence" on page 2-28
- Example 2–43, "Relational Operators in Expressions" on page 2-35
- Example 2–44, "LIKE Operator in Expression" on page 2-36
- Example 2–46, "BETWEEN Operator in Expressions" on page 2-37
- Example 2–47, "IN Operator in Expressions" on page 2-38
- Example 2–50, "Simple CASE Expression" on page 2-40
- Example 2–52, "Searched CASE Expression" on page 2-41


## Related Topics

**In this chapter:**


- "Constant" on page 13-32
- "Literal" on page 13-91
- "Variable" on page 13-131


**In other chapters:**


- "Expressions" on page 2-25
- "Operator Precedence" on page 2-27
- "Predefined PL/SQL Scalar Data Types and Subtypes" on page 3-2
- "Implicit Conversion" on page 3-27

# FETCH Statement

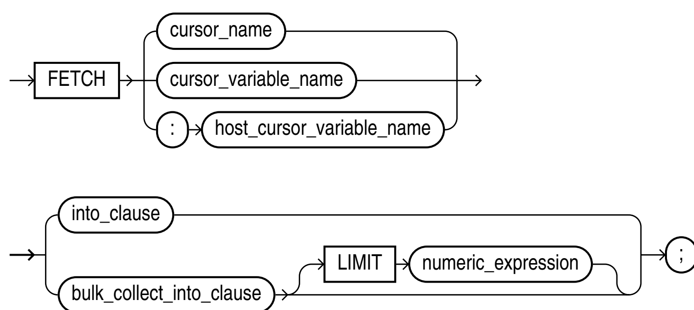
The FETCH statement retrieves rows of data from the result set of a multiple-row query—one at a time, several at a time, or all at once—and stores the data in variables, records, or collections.

Topics:

- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

### *fetch_statement* ::=




See:

- *bulk_collect_into_clause ::=* on page 13-113
- *into_clause ::=* on page 13-113
- *numeric_expression ::=* on page 13-60

## Semantics

### *bulk_collect_into_clause* [ LIMIT *numeric_expression* ]

To have the FETCH statement retrieve all rows at once, omit LIMIT *numeric_expression*.

To limit the number of rows that the FETCH statement retrieves at once, specify LIMIT *numeric_expression*.

Use *bulk_collect_into_clause* to specify one or more collections in which to store the rows that the FETCH statement returns. For more information about *bulk_collect_into_clause*, see "RETURNING INTO Clause" on page 13-113.

#### Restrictions on *bulk_collect_into_clause*

- You cannot use *bulk_collect_into_clause* in client-side programs.

- When the `FETCH` statement requires implicit data type conversions, *`bulk_collect_into_clause`* can have only one *`collection_name`* or *`host_array_name`*.

### *cursor_name*

The name of an open explicit cursor. To open an explicit cursor, use the "OPEN Statement" on page 13-97.

### *cursor_variable_name*

The name of an open cursor variable. To open a cursor variable, use the "OPEN FOR Statement" on page 13-98. The cursor variable can be a formal subprogram parameter (see "Cursor Variables as Subprogram Parameters" on page 6-28).

### *host_cursor_variable_name*

The name of a cursor variable declared in a PL/SQL host environment, passed to PL/SQL as a bind argument, and then opened. To open a cursor variable, use the "OPEN FOR Statement" on page 13-98. Do not put space between the colon (:) and *`host_cursor_variable_name`*.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

### *into_clause*

To have the `FETCH` statement retrieve one row at a time, use this clause to specify the variables or record in which to store the column values of a row that the cursor returns. For more information about this clause, see "RETURNING INTO Clause" on page 13-113.

## Usage

A sequence of `FETCH` statements always runs out of data to retrieve, and no exception is raised when a `FETCH` statement returns no data. To detect this condition, use the cursor attribute `%FOUND` or `%NOTFOUND`, described in "Named Cursor Attribute" on page 13-93.

## Examples


- Example 6–7, "Fetching with a Cursor" on page 6-11
- Example 6–9, "Fetching the Same Cursor Into Different Variables" on page 6-12
- Example 6–10, "Fetching Bulk Data with a Cursor" on page 6-13
- Example 6–33, "Fetching from Cursor Variable into Record" on page 6-31
- Example 6–34, "Fetching from Cursor Variable into Collections" on page 6-32
- Example 6–46, "Trying to Fetch with FOR UPDATE Cursor After COMMIT Statement" on page 6-46
- Example 7–4, "Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements" on page 7-5
- Example 12–16, "Bulk-Fetching from a Cursor Into One or More Collections" on page 12-23
- Example 12–17, "Bulk-Fetching from a Cursor Into a Collection of Records" on page 12-25

## Related Topics

**In this chapter:**


- "Assignment Statement" on page 13-3
- "CLOSE Statement" on page 13-21
- "Cursor Variable" on page 13-39
- "Explicit Cursor" on page 13-54
- "OPEN Statement" on page 13-97
- "OPEN FOR Statement" on page 13-98
- "RETURNING INTO Clause" on page 13-113
- "SELECT INTO Statement" on page 13-118


**In other chapters:**


- "Fetching Data with Explicit Cursors" on page 6-11
- "Fetching Bulk Data with Explicit Cursors" on page 6-13
- "Looping Through Multiple Rows (Cursor FOR LOOP Statement)" on page 6-21
- "Fetching from Cursor Variables" on page 6-31
- "OPEN FOR, FETCH, and CLOSE Statements" on page 7-4
- "Retrieving Query Results into Collections" on page 12-21

# FOR LOOP Statement

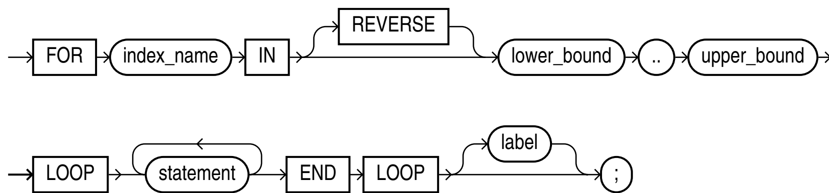
With each iteration of the FOR LOOP statement, its statements run, its index is either incremented or decremented, and control returns to the top of the loop. The FOR LOOP statement ends when its index reaches a specified value, when a statement inside the loop transfers control outside the loop, or when an exception is raised.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***for_loop_statement* ::=**




See *statement ::=* on page 13-11.

## Semantics

### *index_name*

An identifier for the implicitly declared integer variable that is local to the FOR LOOP statement. Statements outside the loop cannot reference *index_name*. Statements inside the loop can reference *index_name*, but cannot change its value. After the FOR LOOP statement runs, *index_name* is undefined.

**See Also:** "FOR LOOP Index" on page 4-15

### *label*

A label that identifies *for_loop_statement* (see ***label*** on page 13-14). CONTINUE, EXIT, and GOTO statements can reference this label.

Labels improve readability, especially when LOOP statements are nested, but only if you ensure that the label at the end of the LOOP statement matches one of the labels at the beginning of the same statement (the compiler does not check).

### [ REVERSE ] *lower_bound* .. *upper_bound*

The *lower_bound* and *upper_bound* must evaluate to numbers (see "Lower Bound and Upper Bound" on page 4-18). PL/SQL evaluates *lower_bound* and *upper_bound* once, when the FOR LOOP statement is entered, and stores them as temporary PLS_INTEGER values, rounding them to the nearest integer if necessary.

If *lower_bound* equals *upper_bound*, the *statements* run only once.

If *lower_bound* does not equal *upper_bound* when the FOR LOOP statement begins to run, then:

- If REVERSE is omitted:

  If *lower_bound* is greater than *upper_bound*, the *statements* do not run, and control transfers to the statement after the FOR LOOP statement.

  Otherwise, *lower_bound* is assigned to *index*, the *statements* run, and control returns to the top of the loop, where *index* is compared to *upper_bound*. If *index* is less than *upper_bound*, *index* is incremented by one, the *statements* run again, and control returns to the top of the loop. When *index* is greater than *upper_bound*, control transfers to the statement after the FOR LOOP statement.

- If REVERSE is specified:

  If *upper_bound* is less than *lower_bound*, the *statements* do not run, and control transfers to the statement after the FOR LOOP statement.

  Otherwise, *upper_bound* is assigned to *index*, the *statements* run, and control returns to the top of the loop, where *index* is compared to *lower_bound*. If *index* is greater than *lower_bound*, *index* is decremented by one, the *statements* run again, and control returns to the top of the loop. When *index* is less than *lower_bound*, control transfers to the statement after the FOR LOOP statement.

## Examples


- Example 4–15, "FOR LOOP Statements" on page 4-14
- Example 4–16, "Reverse FOR LOOP Statements" on page 4-15
- Example 4–22, "Several Types of FOR LOOP Bounds" on page 4-18
- Example 4–23, "Changing the Increment of the Counter in a FOR LOOP Statement" on page 4-18
- Example 4–24, "Specifying a LOOP Range at Run Time" on page 4-18
- Example 4–25, "FOR LOOP with Lower Bound > Upper Bound" on page 4-19
- Example 4–18, "Statement Outside FOR LOOP Tries to Reference Index" on page 4-16
- Example 4–19, "FOR LOOP Index with Same Name as Declared Variable" on page 4-16
- Example 4–20, "FOR LOOP References Declared Variable with Same Name as Index" on page 4-17
- Example 4–21, "Nested FOR LOOP Statements with Same Index Name" on page 4-17
- Example 4–26, "EXIT in FOR LOOP" on page 4-20
- Example 4–27, "EXIT with Label in FOR LOOP" on page 4-20


## Related Topics

**In this chapter:**


- "Basic LOOP Statement" on page 13-16
- "CONTINUE Statement" on page 13-35

- "Cursor FOR LOOP Statement" on page 13-37
- "EXIT Statement" on page 13-52
- "FETCH Statement" on page 13-67
- "FORALL Statement" on page 13-73
- "OPEN Statement" on page 13-97
- "WHILE LOOP Statement" on page 13-133


**In other chapters:**


- "FOR LOOP Statement" on page 4-14

# FORALL Statement

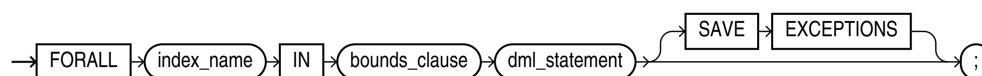
The FORALL statement runs one DML statement multiple times, with different values in the VALUES and WHERE clauses. The different values come from existing, populated collections. The FORALL statement is usually much faster than an equivalent FOR loop.

Topics:

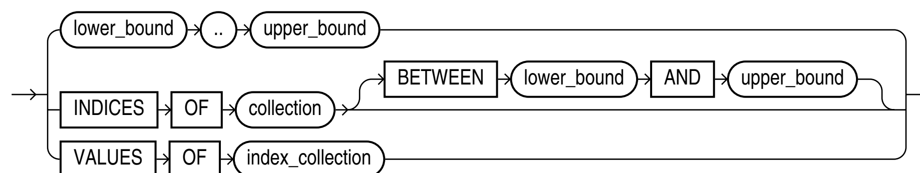
- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

***forall_statement*** **::=**




***bounds_clause*** **::=**




## Semantics

### *bounds_clause*

Specifies the collection element subscripts that provide values for the variable *index_name*. For each value, the SQL engine runs *dml_statement* once.

### *dml_statement*

A static or dynamic INSERT, UPDATE, or DELETE statement that references at least one collection in its VALUES or WHERE clause. Performance benefits apply only to collection references that use *index_name* as a subscript.

Every collection that *dml_statement* references must have subscripts that match the values of *index_name*. If you apply the DELETE, EXTEND, or TRIM method to one collection, apply it to the other collections also, so that all collections have the same set of subscripts. If any collection lacks a referenced element, an exception is raised.

#### Restrictions on *dml_statement*

- If *dml_statement* is an UPDATE statement, its SET and WHERE clauses cannot reference the same collection.

  The workaround is to make a copy of the collection, and reference the original collection in the SET clause and the copy in the WHERE clause.

- If *dml_statement* is a dynamic SQL statement, then values in the USING clause (bind arguments for the dynamic SQL statement) must be simple references to the collection, not expressions. For example, *collection*(*i*) is valid, but UPPER(*collection*(*i*)) is invalid.

#### *index_name*

An identifier for the implicitly declared integer variable that is local to the FORALL statement. Statements outside the FORALL statement cannot reference *index_name*. Statements inside the FORALL statement can reference *index_name* as an index variable, but cannot use it in expressions or change its value. After the FORALL statement runs, *index_name* is undefined.

#### INDICES OF *collection* [ BETWEEN *lower_bound* AND *upper_bound* ]

Specifies that the values of *index_name* correspond to the subscripts of the elements of the specified collection. The subscripts need not be consecutive.

Both *lower_bound* and *upper_bound* are numeric expressions that PL/SQL evaluates once, when the FORALL statement is entered, and rounds to the nearest integer if necessary. The resulting integers are the lower and upper bounds of a valid range of index numbers, which need not be consecutive.

**Restriction on *collection*** If *collection* is an associative array, it must be indexed by integer.

#### *lower_bound* .. *upper_bound*

Both *lower_bound* and *upper_bound* are numeric expressions that PL/SQL evaluates once, when the FORALL statement is entered, and rounds to the nearest integer if necessary. The resulting integers must be the lower and upper bounds of a valid range of consecutive index numbers. If an element in the range is missing or was deleted, an exception is raised.

#### SAVE EXCEPTIONS

This option enables the FORALL loop to continue even if some of its DML statements fail. Instead of raising an exception immediately, the program raises a single exception after the FORALL statement finishes. For information about the exceptions, use the implicit cursor attribute **SQL%BULK_EXCEPTIONS** on page 13-85.

#### VALUES OF *index_collection*

Specifies that the values of *index_name* are the elements of *index_collection*, a collection of PLS_INTEGER elements that is indexed by PLS_INTEGER. The subscripts of *index_collection* need not be consecutive. If *index_collection* is empty, an exception is raised and the FORALL statement does not run.

## Usage

You can use the FORALL statement only in server-side programs, not in client-side programs.

## Examples


- Example 12–6, "Issuing DELETE Statements in a Loop" on page 12-11
- Example 12–7, "Issuing INSERT Statements in a Loop" on page 12-12
- Example 12–8, "FORALL Statement for Part of Collection" on page 12-13

- Example 12–9, "FORALL Statement for Nonconsecutive Index Values" on page 12-13
- Example 12–11, "FORALL Statement and SQL%BULK_EXCEPTIONS" on page 12-18
- Example 12–20, "FORALL with BULK COLLECT" on page 12-28

## Related Topics

**In this chapter:**

- "Basic LOOP Statement" on page 13-16

**In other chapters:**

- "Running One DML Statement Multiple Times (FORALL Statement)" on page 12-11
- "Retrieving Query Results into Collections" on page 12-21

# Function

A **function** is a subprogram that returns a single value. You must declare and define a function before invoking it. You can either declare and define it at the same time, or you can declare it first and then define it later in the same block, subprogram, or package. A function invocation, or function call, is an expression.

---

**Note:**

- This topic applies to functions that you declare and define inside a PL/SQL block, subprogram, or package. These functions differ from standalone stored functions that you create with the "CREATE FUNCTION Statement" on page 14-32, as explained in "Overview of PL/SQL Subprograms" on page 8-1.
- A function declared and defined in a package is affected by the AUTHID clause of the package. For more information, see "CREATE PACKAGE Statement" on page 14-42.

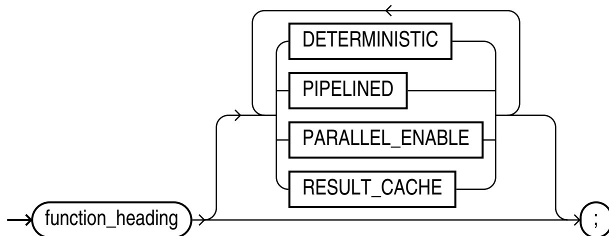
---

Topics:

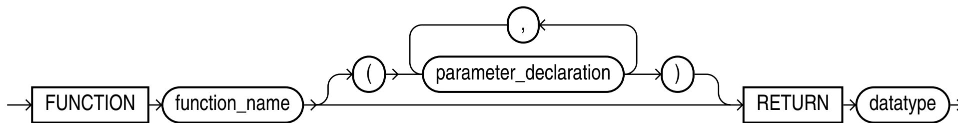
- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***function_declaration*** **::=**



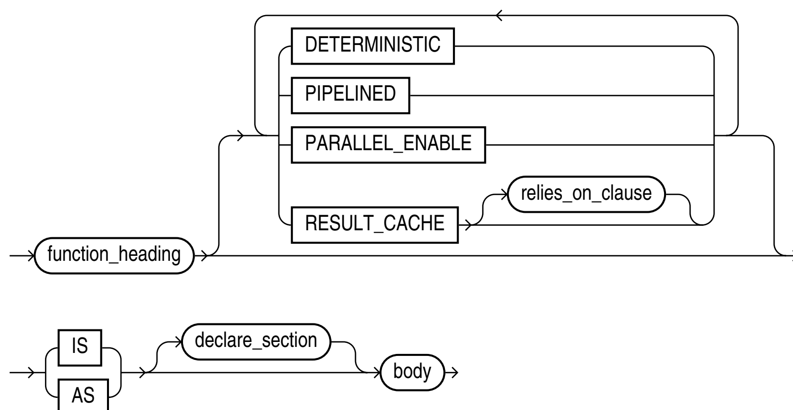

***function_heading*** **::=**




See:

- *datatype ::=* on page 13-32
- *parameter_declaration ::=* on page 13-101

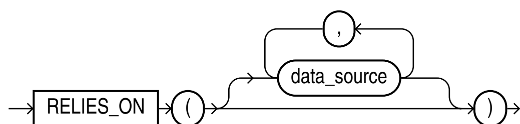
***function_definition* ::=**




See:

- *body ::=* on page 13-11
- *declare_section ::=* on page 13-8

***relies_on_clause* ::=**




## Semantics

#### *body*

The required executable part of the function and, optionally, the exception-handling part of the function.

At least one execution path must lead to a `RETURN` statement in the executable part of the function; otherwise, a run-time error occurs.

#### *declare_section*

The optional declarative part of the function. Declarations are local to the function, can be referenced in *body*, and cease to exist when the function completes execution.

#### DETERMINISTIC

Specify `DETERMINISTIC` to indicate that the function returns the same result value whenever it is invoked with the same values for its parameters. This helps the optimizer avoid redundant function calls: If a stored function was invoked 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, because results might vary across calls. Instead, consider making the function result-cached (see "Making Result-Cached Functions Handle Session-Specific Settings" on page 8-35 and "Making Result-Cached Functions Handle Session-Specific Application Contexts" on page 8-36).

Only `DETERMINISTIC` functions can be invoked from a function-based index or a materialized view that has query-rewrite enabled. For more information and possible

limitations of the DETERMINISTIC option, see "CREATE FUNCTION Statement" on page 14-32.

**See Also:**

- "Subprogram Side Effects" on page 8-27
- CREATE INDEX statement in *Oracle Database SQL Language Reference*

### *function_declaration*

Declares a function, but does not define it. The definition must appear later in the same block, subprogram, or package as the declaration.

A function declaration is also called a **function specification**, or **function spec**.

### *function_definition*

Either defines a function that was declared earlier or both declares and defines a function.

### *function_name*

The name that you give to the function that you are declaring or defining.

### IN, OUT, IN OUT

Parameter modes that define the action of formal parameters. For summary information about parameter modes, see Table 8–1 on page 8-10.

### NOCOPY

Specify NOCOPY to instruct the database to pass this argument as fast as possible. This clause can significantly enhance performance when passing a large value like a record, an associative array, or a varray to an OUT or IN OUT parameter. IN parameter values are always passed NOCOPY.

- When you specify NOCOPY, assignments made to a package variable may show immediately in this parameter, or assignments made to this parameter may show immediately in a package variable, if the package variable is passed as the actual assignment corresponding to this parameter.
- Similarly, changes made either to this parameter or to another parameter may be visible immediately through both names if the same variable is passed to both.
- If the function is exited with an unhandled exception, then any assignment made to this parameter may be visible in the caller's variable.

These effects might not occur on any particular call. You should use NOCOPY only when these effects would not matter.

### *parameter_name*

The name of the formal parameter that you are declaring, which you can reference in *body*.

### PARALLEL_ENABLE

Enables the function to be used safely in slave sessions of parallel DML evaluations.

#### PIPELINED

PIPELINED specifies to return the results of a table function iteratively. A table function returns a collection type (a nested table or varray) with elements that are SQL data types. You can query table functions using the TABLE keyword before the function name in the FROM clause of a SQL query. For more information, see "Performing Multiple Transformations with Pipelined Table Functions" on page 12-40.

#### *relies_on_clause*

Specifies the data sources on which the results of the function depend. Each *data_source* is the name of either a database table or view.

**Note:**

- This clause is deprecated. As of Release 11.2, the database detects all data sources that are queried while a result-cached function is running, and *relies_on_clause* does nothing.
- You cannot use *relies_on_clause* in a function declared in an anonymous block.

#### RESULT_CACHE

Causes the results of the function to be cached. For more information, see "PL/SQL Function Result Cache" on page 8-29.

#### RETURN *datatype*

For *datatype*, specify the data type of the return value of the function. The return value can have any data type supported by PL/SQL.

**Restriction on *datatype*** You cannot constrain this data type (with NOT NULL, for example).

### Examples

- Example 8–2, "Declaring, Defining, and Invoking a Simple PL/SQL Function" on page 8-5
- Example 5–44, "Returning a Record from a Function" on page 5-36

### Related Topics

#### In this chapter:

- "Parameter" on page 13-101
- "Procedure" on page 13-103

#### In other chapters:

- Chapter 8, "PL/SQL Subprograms"

**See Also:** *Oracle Database Advanced Application Developer's Guide* for information about restrictions on user-defined functions that are called from SQL statements and expressions

# GOTO Statement

The GOTO statement transfers control to a labeled block or statement.

Topics:

- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

***goto_statement*** **::=**

## Semantics

***label***

Identifies either a block or a statement (see the syntax diagrams *plsql_block ::=* on page 13-8 and *statement ::=* on page 13-11 and the description of ***label*** on page 13-14).

If the GOTO statement cannot find *label* in the current block, it transfers control to the first enclosing block in which *label* appears.

## Usage

If a GOTO statement exits a cursor FOR loop prematurely, the cursor closes.

### GOTO Statement Restrictions

- A GOTO statement cannot transfer control into an IF statement, CASE statement, LOOP statement, or sub-block.
- A GOTO statement cannot transfer control from one IF statement clause to another, or from one CASE statement WHEN clause to another.
- A GOTO statement cannot transfer control out of a subprogram.
- A GOTO statement cannot transfer control from an exception handler back into the current block. However, a GOTO statement can transfer control from an exception handler into an enclosing block.

## Examples

- Example 4–28, "GOTO Statement" on page 4-21
- Example 4–31, "GOTO Statement Transfers Control to Enclosing Block" on page 4-22
- Example 4–32, "GOTO Statement Cannot Transfer Control into IF Statement" on page 4-22

## Related Topics

**In this chapter:**


- "Block" on page 13-8


**In other chapters:**


- "GOTO Statement" on page 4-21

# IF Statement

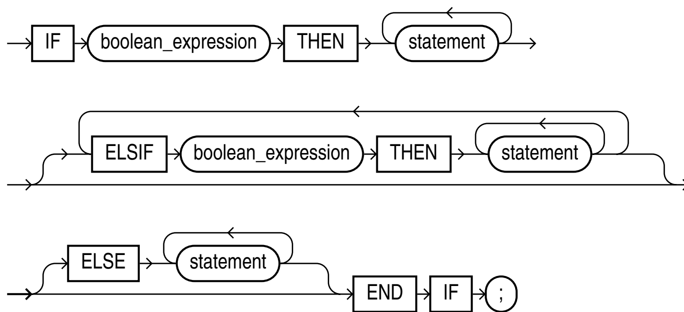
The IF statement runs or skips a sequence of one or more statements, depending on the value of a BOOLEAN expression.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***if_statement*** **::=**




See *boolean_expression ::=* on page 13-59.

## Semantics

### *boolean_expression*

The first *boolean_expression* is always evaluated. Except for the first one, a *boolean_expression* is evaluated only if the values of the preceding expressions are FALSE.

If a *boolean_expression* is evaluated and its value is TRUE, the statements after the corresponding THEN run. The succeeding expressions are not evaluated, and the statements associated with them do not run.

### ELSE

If no *boolean_expression* has the value TRUE, the statements after ELSE run.

## Examples

- Example 4–1, "IF THEN Statement" on page 4-2
- Example 4–2, "IF THEN ELSE Statement" on page 4-3
- Example 4–3, "Nested IF THEN ELSE Statements" on page 4-4
- Example 4–4, "IF THEN ELSIF Statement" on page 4-5

## Related Topics

**In this chapter:**

- "CASE Statement" on page 13-18
- "Expression" on page 13-58

**In other chapters:**

- "Conditional Selection Statements" on page 4-1

# Implicit Cursor Attribute

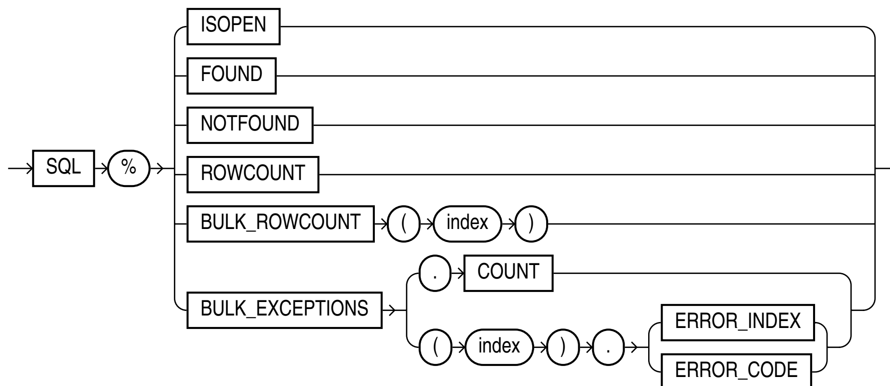
An implicit cursor has attributes that return information about the most recently run SELECT or DML statement that is not associated with a named cursor.

Topics:

- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

***implicit_cursor_attribute* ::=**




## Semantics

### %ISOPEN

SQL%ISOPEN always has the value FALSE.

### %FOUND

SQL%FOUND has one of these values:

- If no SELECT or DML statement has run, NULL.
- If the most recent SELECT or DML statement returned a row, TRUE.
- If the most recent SELECT or DML statement did not return a row, FALSE.

### %NOTFOUND

SQL%NOTFOUND has one of these values:

- If no SELECT or DML statement has run, NULL.
- If the most recent SELECT or DML statement returned a row, FALSE.
- If the most recent SELECT or DML statement did not return a row, TRUE.

#### %ROWCOUNT

`SQL%ROWCOUNT` has one of these values:

- If no `SELECT` or DML statement has run, `NULL`.
- If a `SELECT` or DML statement has run, the number of rows fetched so far.

#### SQL%BULK_ROWCOUNT

A composite attribute for use with the "FORALL Statement" on page 13-73. This attribute acts like an associative array. 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, `SQL%BULK_ROWCOUNT(i)` has the value zero. For more information, see "Counting Rows Affected by FORALL" on page 12-19.

**Restriction on SQL%BULK_ROWCOUNT** You cannot assign the value of `SQL%BULK_ROWCOUNT(index)` to another collection or pass it as a parameter to a subprogram.

#### SQL%BULK_EXCEPTIONS

A composite attribute for use with a `FORALL` statement that has a `SAVE EXCEPTIONS` clause (see "FORALL Statement" on page 13-73). This attribute acts like an associative array that stores information about any exceptions raised while the `FORALL` statement ran. For each index value *i* between 1 and `SQL%BULK_EXCEPTIONS.COUNT`:

- `SQL%BULK_EXCEPTIONS(i).ERROR_INDEX` specifies which iteration of the `FORALL` loop caused an exception.
- `SQL%BULK_EXCEPTIONS(i).ERROR_CODE` specifies the Oracle Database error code that corresponds to the exception.

For more information, see "Effect of FORALL Exceptions on Rollbacks" on page 12-16 and "Handling FORALL Exceptions" on page 12-17.

## Usage

You can use cursor attributes in procedural statements, but not in SQL statements.

## Examples

- Example 6–3, "SQL%FOUND Attribute" on page 6-7
- Example 6–4, "SQL%ROWCOUNT Attribute" on page 6-8
- Example 6–7, "Fetching with a Cursor" on page 6-11
- Example 6–12, "%FOUND Attribute" on page 6-14
- Example 6–11, "%ISOPEN Attribute" on page 6-14
- Example 6–13, "%NOTFOUND Attribute" on page 6-15
- Example 6–14, "%ROWCOUNT Attribute" on page 6-16
- Example 12–11, "FORALL Statement and SQL%BULK_EXCEPTIONS" on page 12-18
- Example 12–12, "FORALL Statement and SQL%BULK_ROWCOUNT" on page 12-19
- Example 12–13, "Counting Rows Affected by FORALL with SQL%BULK_ROWCOUNT" on page 12-20

## Related Topics

**In this chapter:**

- "Named Cursor Attribute" on page 13-93
- "FORALL Statement" on page 13-73

**In other chapters:**

- "Implicit Cursors" on page 6-6
- "Queries" on page 6-19

# INLINE Pragma

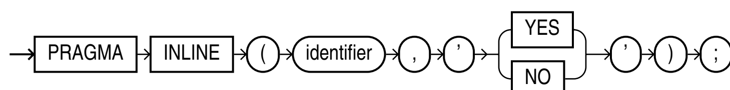
The INLINE pragma specifies that a subprogram call is, or is not, to be inlined. Inlining replaces a subprogram call (to a subprogram in the same program unit) with a copy of the called subprogram.

Topics:

- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

***inline_pragma*** **::=**




## Semantics

***identifier***

The name of a subprogram.

**YES**

If PLSQL_OPTIMIZE_LEVEL=2, 'YES' specifies that the subprogram call is to be inlined.

If PLSQL_OPTIMIZE_LEVEL=3, 'YES' specifies that the subprogram call has a high priority for inlining.

**NO**

Specifies that the subprogram call is not to be inlined.

## Usage

The INLINE pragma affects only the immediately following declaration or statement, and only some kinds of statements.

When the INLINE pragma immediately precedes one of these statements, the pragma affects every call to the specified subprogram in that statement (see Example 12–1):

- Assignment
- Call
- Conditional
- CASE
- CONTINUE WHEN

- `EXECUTE IMMEDIATE`
- `EXIT WHEN`
- `LOOP`
- `RETURN`

The `INLINE` pragma does not affect statements that are not in the preceding list.

When the `INLINE` pragma immediately precedes a declaration, it affects:

- Every call to the specified subprogram in that declaration
- Every initialization value in that declaration except the default initialization values of records

If the name of the subprogram (*`identifier`*) is overloaded (that is, if it belongs to multiple subprograms), the `INLINE` pragma applies to every subprogram with that name (see Example 12–2). For information about overloaded subprogram names, see "Overloaded Subprograms" on page 8-13.

The `PRAGMA INLINE (`*`identifier`*`, 'YES')` very strongly encourages the compiler to inline a particular call, but the compiler might not to do so if other considerations or limits make the inlining undesirable. If you specify `PRAGMA INLINE (`*`identifier`*`,'NO')`, the compiler does not inline calls to subprograms named *`identifier`* (see Example 12–3).

Multiple pragmas can affect the same declaration or statement. Each pragma applies its own effect to the statement. If `PRAGMA INLINE(`*`identifier`*`,'YES')` and `PRAGMA INLINE (`*`identifier`*`,'NO')` have the same *`identifier`*, `'NO'` overrides `'YES'` (see Example 12–4). One `PRAGMA INLINE (`*`identifier`*`,'NO')` overrides any number of occurrences of `PRAGMA INLINE (`*`identifier`*`,'YES')`, and the order of these pragmas is not important.

## Examples

- Example 12–1, "Specifying that a Subprogram Is To Be Inlined" on page 12-2
- Example 12–2, "Specifying that an Overloaded Subprogram Is To Be Inlined" on page 12-2
- Example 12–3, "Specifying that a Subprogram Is Not To Be Inlined" on page 12-3
- Example 12–4, "Applying Two INLINE Pragmas to the Same Subprogram" on page 12-3

## Related Topics

- "Subprogram Inlining" on page 12-2

# INSERT Statement Extension

The PL/SQL extension to the SQL INSERT statement lets you specify a record name in the *values_clause* of the *single_table_insert* instead of specifying a column list in the *insert_into_clause*. Effectively, this form of the INSERT statement inserts the record into the table; actually, it adds a row to the table and gives each column of the row the value of the corresponding record field.

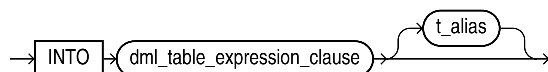
> **See Also:** *Oracle Database SQL Language Reference* for the syntax of the SQL INSERT statement

Topics:

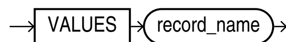
- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

### *insert_into_clause* ::=




### *values_clause* ::=




## Semantics

### *dml_table_expression_clause*

Typically a table name. For complete information, see *Oracle Database SQL Language Reference*.

### *t_alias*

An alias for the item described by *dml_table_expression_clause*.

### *record_name*

The name of a record variable of type RECORD or %ROWTYPE. The record must represent a row of the item described by *dml_table_expression_clause*. That is, for every column of the row, the record must have a field with a compatible data type (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions"). If a column has a NOT NULL constraint, then its corresponding field cannot have a NULL value.

> **See Also:** *Oracle Database SQL Language Reference* for the complete syntax of the INSERT statement

## Examples

- Example 5–52, "Initializing a Table by Inserting a Record of Default Values" on page 5-41

## Related Topics

**In this chapter:**

- "Record" on page 13-106
- "%ROWTYPE Attribute" on page 13-116

**In other chapters:**

- "Inserting Records into Tables" on page 5-40
- "Restrictions on Record Inserts and Updates" on page 5-43

# Literal

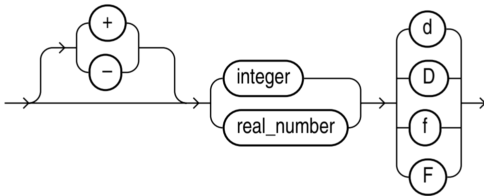
A literal is value that is not represented by an identifier or an expression.

Topics:

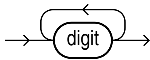
- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

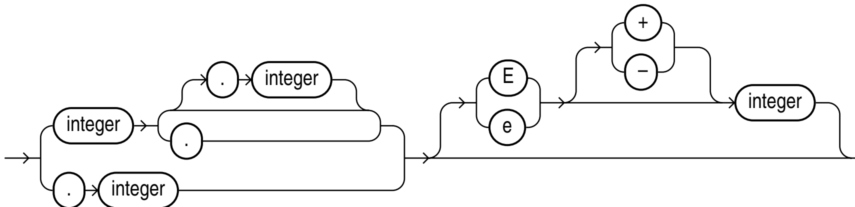
***numeric_literal*** **::=**



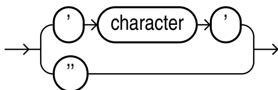

***integer*** **::=**



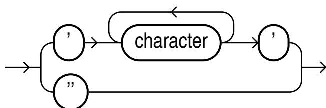

***real_number*** **::=**



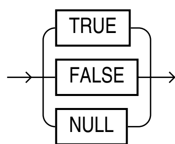

***character_literal*** **::=**




***string_literal*** **::=**



***boolean_literal* ::=**




## Semantics

***character***

A member of the PL/SQL character set. For information about the PL/SQL character set, see"Character Sets" on page 2-1.

***d* or *D***

Specifies that the numeric literal has the data type BINARY_DOUBLE.

***digit***

The character 0, 1, 2, 3, 4, 5, 6, 7, 8, or 9.

***f* or *F***

Specifies that the numeric literal has the data type BINARY_FLOAT.

## Examples

- Example 2–4, "Datetime and Interval Literals" on page 2-10
- Example 2–59, "Compiling Different Code for Different Database Versions" on page 2-52

## Related Topics

**In this chapter:**

- "Constant" on page 13-32
- "Expression" on page 13-58

**In other chapters:**

- "Literals" on page 2-8

# Named Cursor Attribute

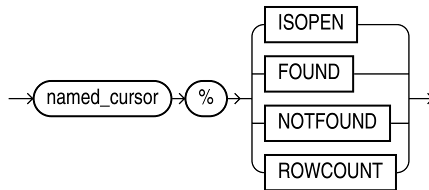
Every named cursor (explicit cursor or cursor variable) has four attributes, each of which returns information about the execution of a SQL DML statement.

Topics:

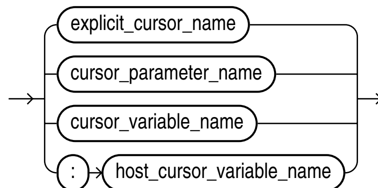
- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

***named_cursor_attribute*** **::=**




***named_cursor*** **::=**




## Semantics

***explicit_cursor_name***

The name of an explicit cursor.

***cursor_parameter_name***

The name of a formal cursor parameter.

***cursor_variable_name***

The name of a cursor variable.

***host_cursor_variable_name***

The name of a cursor variable that was declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. The data type of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Do not put space between the colon (:) and *host_cursor_variable_name*.

### %ISOPEN

*named_cursor*%ISOPEN has the value TRUE if the cursor is open, and FALSE if it is not open.

### %FOUND

*named_cursor*%FOUND has one of these values:

- If the cursor is not open, INVALID_CURSOR
- If cursor is open but no fetch was attempted, NULL.
- If the most recent fetch returned a row, TRUE.
- If the most recent fetch did not return a row, FALSE.

### %NOTFOUND

*named_cursor*%NOTFOUND has one of these values:

- If cursor is not open, INVALID_CURSOR.
- If cursor is open but no fetch was attempted, NULL.
- If the most recent fetch returned a row, FALSE.
- If the most recent fetch did not return a row, TRUE.

### %ROWCOUNT

*named_cursor*%ROWCOUNT has one of these values:

- If cursor is not open, INVALID_CURSOR.
- If cursor is open, the number of rows fetched so far.

## Usage

You can use cursor attributes in procedural statements, but not in SQL statements.

When a named cursor is opened, the rows that satisfy the associated query are the result set. Rows are fetched from the result set one at a time.

Every named cursor has its own attributes. You can open multiple named cursors, and then use %FOUND or %NOTFOUND to tell which cursors have rows left to fetch, and %ROWCOUNT to tell how many rows each cursor has fetched so far.

Because *named_cursor*%NOTFOUND has the value NULL before the first fetch, if FETCH never runs successfully, the condition *named_cursor*%NOTFOUND is never TRUE. If you use this condition to exit a loop, the loop can never end. Instead, use this condition:

```
named_cursor%NOTFOUND OR (named_cursor%NOTFOUND IS NULL);
```

## Examples

- Example 6–11, "%ISOPEN Attribute" on page 6-14
- Example 6–12, "%FOUND Attribute" on page 6-14
- Example 6–13, "%NOTFOUND Attribute" on page 6-15
- Example 6–14, "%ROWCOUNT Attribute" on page 6-16

## Related Topics

**In this chapter:**


- "Cursor Variable" on page 13-39
- "Explicit Cursor" on page 13-54
- "Implicit Cursor Attribute" on page 13-84


**In other chapters:**


- "Explicit Cursor Attributes" on page 6-14

# NULL Statement

The NULL statement is a no-op (no operation)—it passes control to the next statement without doing anything.

> **Note:** The NULL statement and the BOOLEAN value NULL are not related.

Topics:

- Syntax
- Examples
- Related Topics

## Syntax

***null_statement*** **::=**

```
→ NULL → ;
```

## Examples

- Example 4–30, "NULL Statement Allows GOTO to Label" on page 4-22
- Example 4–33, "NULL Statement Showing No Action" on page 4-23
- Example 4–34, "NULL Statement as Placeholder During Subprogram Creation" on page 4-24

## Related Topics

- "NULL Statement" on page 4-23

# OPEN Statement

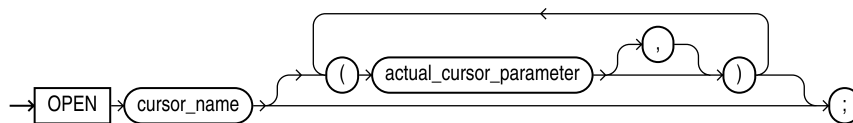
The `OPEN` statement opens an explicit cursor, allocates database resources to process the associated query, identifies the result set, and positions the cursor before the first row of the result set. (If the query has a `FOR UPDATE` clause, processing the query includes locking the rows of the result set—see "SELECT FOR UPDATE and FOR UPDATE Cursors" on page 6-44.)

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***open_statement* ::=**




## Semantics

***actual_cursor_parameter***

An actual parameter that corresponds to a formal parameter of the explicit cursor *`cursor_name`*. For more information, see "Explicit Cursors that Accept Parameters" on page 6-18.

***cursor_name***

The name of an explicit cursor that is not open.

## Examples

- Example 6–6, "Opening an Explicit Cursor" on page 6-10

## Related Topics

**In this chapter:**

- "CLOSE Statement" on page 13-21
- "Explicit Cursor" on page 13-54
- "FETCH Statement" on page 13-67
- "OPEN FOR Statement" on page 13-98

**In other chapters:**

- "Opening Explicit Cursors" on page 6-10

# OPEN FOR Statement

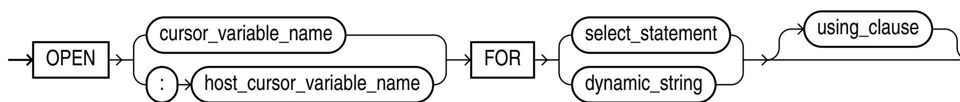
The OPEN FOR statement associates a cursor variable with a multiple-row query, allocates database resources to process the query, identifies the result set, and positions the cursor before the first row of the result set. (If the query has a FOR UPDATE clause, processing the query includes locking the rows of the result set—see "SELECT FOR UPDATE and FOR UPDATE Cursors" on page 6-44.)

Topics:

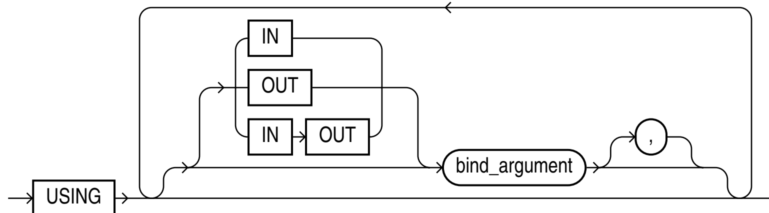
- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***open_for_statement* ::=**




***using_clause* ::=**




## Semantics

***bind_argument***

An expression whose value replaces its corresponding placeholder in *select_statement* or *dynamic_string* at run time. You must specify a *bind_argument* for every placeholder.

***cursor_variable_name***

The name of a cursor variable. If the cursor variable is the formal parameter of a subprogram, it must not have a return type. For information about cursor variables as subprogram parameters, see "Cursor Variables as Subprogram Parameters" on page 6-28.

***dynamic_string***

A string literal, string variable, or string expression; of type CHAR, VARCHAR2, or CLOB; that represents a multiple-row SQL SELECT statement (not a PL/SQL SELECT INTO statement).

> **See:** *Oracle Database SQL Language Reference* for `SELECT` statement syntax

#### *host_cursor_variable_name*

The name of a cursor variable that was declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. The data type of the cursor variable is compatible with the return type of any PL/SQL cursor variable. Do not put space between the colon (:) and *`host_cursor_variable_name`*.

#### IN, OUT, IN OUT

Parameter modes of bind arguments. An `IN` bind argument passes its value to the *`select_statement`* or *`dynamic_string`*. An `OUT` bind argument stores a value that *`dynamic_string`* returns. An `IN OUT` bind argument passes its initial value to *`dynamic_string`* and stores a value that *`dynamic_string`* returns. The default parameter mode for *`bind_argument`* is `IN`.

#### *select_statement*

A string literal, string variable, or string expression; of type `CHAR`, `VARCHAR2`, or `CLOB`; that represents a multiple-row SQL `SELECT` statement (not a PL/SQL `SELECT INTO` statement).

> **See:** *Oracle Database SQL Language Reference* for `SELECT` statement syntax

#### *using_clause*

Specifies bind arguments.

##### Restrictions on *using_clause*

- Use if and only if *`select_statement`* or *`dynamic_sql_stmt`* includes placeholders for bind arguments.
- If *`dynamic_sql_stmt`* has a `RETURNING INTO` clause, *`using_clause`* can contain only `IN` bind arguments. The bind arguments in the `RETURNING INTO` clause are `OUT` bind arguments by definition.

## Examples

- Example 6–28, "Cursor Variable as Parameter" on page 6-28
- Example 6–30, "Stored Procedure to Open a Cursor Variable" on page 6-30
- Example 6–31, "Stored Procedure to Open Cursor Variables with Different Queries" on page 6-30
- Example 6–32, "Cursor Variable with Different Return Types" on page 6-31
- Example 7–4, "Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements" on page 7-5

## Related Topics

**In this chapter:**

- "CLOSE Statement" on page 13-21
- "Cursor Variable" on page 13-39
- "EXECUTE IMMEDIATE Statement" on page 13-49

- "FETCH Statement" on page 13-67
- "OPEN Statement" on page 13-97


**In other chapters:**


- "Opening Cursor Variables" on page 6-29
- "OPEN FOR, FETCH, and CLOSE Statements" on page 7-4

# Parameter

A parameter declaration can appear in in following:

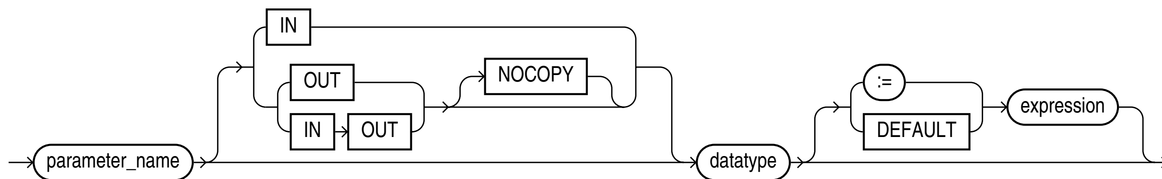
- "Function" on page 13-76
- "Procedure" on page 13-103
- "CREATE FUNCTION Statement" on page 14-32
- "CREATE PROCEDURE Statement" on page 14-49

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***parameter_declaration* ::=**




See:

- *datatype ::=* on page 13-32
- *expression ::=* on page 13-58

## Semantics

***datatype***

The data type of the parameter that you are declaring.

**Restriction on *datatype*** You cannot constrain this data type (with `NOT NULL`, for example).

***expression***

Specifies an initial value for an `IN` parameter. If the invoker of the subprogram specifies a value for the parameter, then *`expression`* is not evaluated for that invocation (see Example 8–8). Otherwise, the parameter is initialized to the value of *`expression`*. The value and the parameter must have compatible data types (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions").

**IN, OUT, IN OUT**

Parameter modes that define the action of formal parameters. For summary information about parameter modes, see Table 8–1 on page 8-10.

---

**Note:** Avoid using OUT and IN OUT with functions. The purpose of a function is to take zero or more parameters and return a single value. Functions must be free from side effects, which change the values of variables not local to the subprogram.

---

#### NOCOPY

Specify NOCOPY to instruct the database to pass this argument as fast as possible. This clause can significantly enhance performance when passing a large value like a record, an associative array, or a varray to an OUT or IN OUT parameter. IN parameter values are always passed NOCOPY.

- When you specify NOCOPY, assignments made to a package variable may show immediately in this parameter, or assignments made to this parameter may show immediately in a package variable, if the package variable is passed as the actual assignment corresponding to this parameter.
- Similarly, changes made either to this parameter or to another parameter may be visible immediately through both names if the same variable is passed to both.
- If the subprogram is exited with an unhandled exception, then any assignment made to this parameter may be visible in the caller's variable.

These effects might not occur on any particular call. Use NOCOPY only when these effects would not matter.

#### *parameter_name*

The name of the formal parameter that you are declaring, which you can reference in the body of the subprogram.

## Examples


- Example 2–27, "Assigning Values to Variables as Parameters of a Subprogram" on page 2-24
- Example 8–6, "Subprogram Parameter Mode OUT" on page 8-8
- Example 8–7, "Procedure with Initial Parameter Values" on page 8-10
- Example 8–8, "Formal Parameter with Expression as Initial Value" on page 8-11


## Related Topics

#### In this chapter:


- "Function" on page 13-76
- "Procedure" on page 13-103


#### In other chapters:


- "Assigning Values to Variables as Parameters of a Subprogram" on page 2-24
- "Subprogram Parameters" on page 8-6
- "CREATE FUNCTION Statement" on page 14-32
- "CREATE PROCEDURE Statement" on page 14-49

# Procedure

A **procedure** is a subprogram that performs a specific action.

You must declare and define a procedure before invoking it. You can either declare and define it at the same time, or you can declare it first and then define it later in the same block, subprogram, or package. A procedure invocation, or procedure call, is a statement.

---

**Note:**

- This topic applies to procedures that you declare and define inside a PL/SQL block, subprogram, or package. These procedures differ from standalone stored procedures that you create with the "CREATE PROCEDURE Statement" on page 14-49, as explained in "Overview of PL/SQL Subprograms" on page 8-1.
- A procedure declared and defined in a package is affected by the `AUTHID` clause of the package. For more information, see "CREATE PACKAGE Statement" on page 14-42.

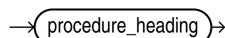
---

Topics:

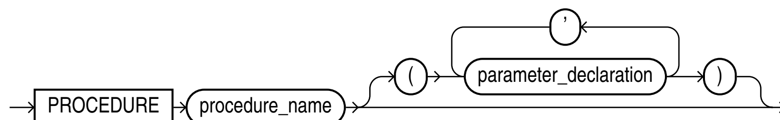
- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

### *procedure_declaration* ::=

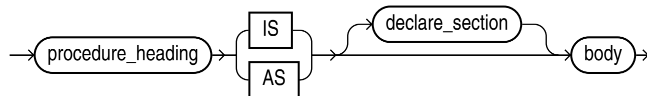



### *procedure_heading* ::=




See *parameter_declaration ::=* on page 13-101.

### *procedure_definition* ::=




See:

- *body ::=* on page 13-11

- *declare_section ::=* on page 13-8

## Semantics

### *body*

The required executable part of the procedure and, optionally, the exception-handling part of the procedure.

### *declare_section*

The optional declarative part of the procedure. Declarations are local to the procedure, can be referenced in *body*, and cease to exist when the procedure completes execution.

### *procedure_declaration*

Declares a procedure, but does not define it. The definition must appear later in the same block, subprogram, or package as the declaration.

A procedure declaration is also called a **procedure specification**, or **procedure spec**.

### *procedure_definition*

Either defines a procedure that was declared earlier or both declares and defines a procedure.

### *procedure_name*

The name that you give to the procedure that you are declaring or defining.

## Examples

- Example 8–1, "Declaring, Defining, and Invoking a Simple PL/SQL Procedure" on page 8-3

## Related Topics

**In this chapter:**

- "Function" on page 13-76
- "Parameter" on page 13-101

**In other chapters:**

- Chapter 8, "PL/SQL Subprograms"
- "CREATE PROCEDURE Statement" on page 14-49

# RAISE Statement

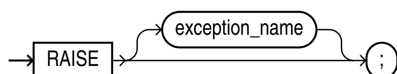
The `RAISE` statement explicitly raises a predefined or user-defined exception.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***raise_statement*** **::=**




## Semantics

***exception_name***

The name of either a predefined exception (see Table 11–2) or a user-defined exception (see "Exception" on page 13-45).

The *`exception_name`* is optional only in an exception handler, where the default is the current exception (see "Reraising an Exception" on page 11-15).

## Examples

- Example 10–6, "Creating emp_admin Package" on page 10-11
- Example 11–6, "Scope of Exceptions" on page 11-10
- Example 11–12, "Reraising an Exception" on page 11-16

## Related Topics

**In this chapter:**

- "Exception" on page 13-45
- "Exception Handler" on page 13-47

**In other chapters:**

- "Predefined Exceptions" on page 11-7
- "User-Defined Exceptions" on page 11-9

# Record

A record variable is a composite variable whose internal components, called fields, can be of different data types. You access each field with this syntax: *record_name.field_name*. A record variable can hold a table row, or some columns from a table row. Each field corresponds to a table column.

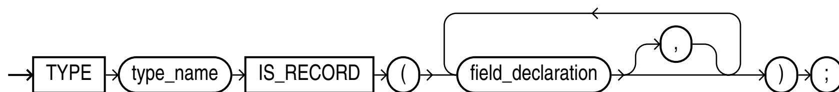
To create a record variable, you first define a record type, and then declare a variable of that type.

Topics:

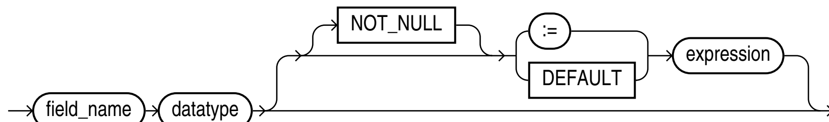
- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

### *record_type_definition* ::=



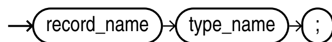

### *field_declaration* ::=




See:

- *datatype ::=* on page 13-32
- *expression ::=* on page 13-58

### *record_variable_declaration* ::=




## Semantics

### *expression*

An expression whose value has a data type compatible with *datatype* (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions"). When *record_variable_declaration* is elaborated, the value of *expression* is assigned to *record_name.field_name*. This value is the initial value of the field.

### *field_name*

The name of the field that you are declaring.

#### NOT NULL

Prevents the program from assigning the value NULL to the field.

PL/SQL subtypes NATURALN, POSITIVEN, and SIMPLE_INTEGER are predefined as NOT NULL. When declaring a field of one of these subtypes, you can omit NOT NULL.

If you declare a NOT NULL field, either by specifying NOT NULL or by specifying a subtype that is predefined as NOT NULL, you must also specify an initial value for the field (see ***expression***).

#### *record_name*

The name of the record variable that you are declaring.

#### *type_name*

The name of the record type that you are defining.

### Examples

- Example 5–8, "VARRAY of Records" on page 5-10
- Example 5–20, "Assigning Values to VARRAYs with Complex Data Types" on page 5-16
- Example 5–21, "Assigning Values to Tables with Complex Data Types" on page 5-17
- Example 5–41, "Declaring and Initializing a Simple Record Type" on page 5-34
- Example 5–42, "Declaring and Initializing Record Types" on page 5-34
- Example 5–44, "Returning a Record from a Function" on page 5-36
- Example 5–45, "Record as Procedure Parameter" on page 5-36
- Example 5–46, "Nested Record Declaration" on page 5-37
- Example 5–47, "Assigning Initial Values to a Record" on page 5-37
- Example 5–51, "Testing Records for Nullity and Equality" on page 5-39
- Example 5–52, "Initializing a Table by Inserting a Record of Default Values" on page 5-41
- Example 5–53, "Updating Rows with a Record" on page 5-42
- Example 5–54, "RETURNING INTO Clause with Record" on page 5-43
- Example 5–55, "BULK COLLECT with SELECT INTO Statement" on page 5-44
- Example 6–27, "Cursor Variable Returning a Record Type" on page 6-28
- Example 10–6, "Creating emp_admin Package" on page 10-11

### Related Topics

#### In this chapter:

- "Collection" on page 13-23
- "%ROWTYPE Attribute" on page 13-116

#### In other chapters:

- "Record Topics" on page 5-1

- "Record Definitions and Declarations" on page 5-34

# RESTRICT_REFERENCES Pragma

> **Note:** The RESTRICT_REFERENCES pragma is deprecated. Oracle recommends using DETERMINISTIC and PARALLEL_ENABLE (described in "Function" on page 13-76) instead of RESTRICT_REFERENCES.

The RESTRICT_REFERENCES pragma asserts that a user-defined subprogram does not read or write database tables or package variables.

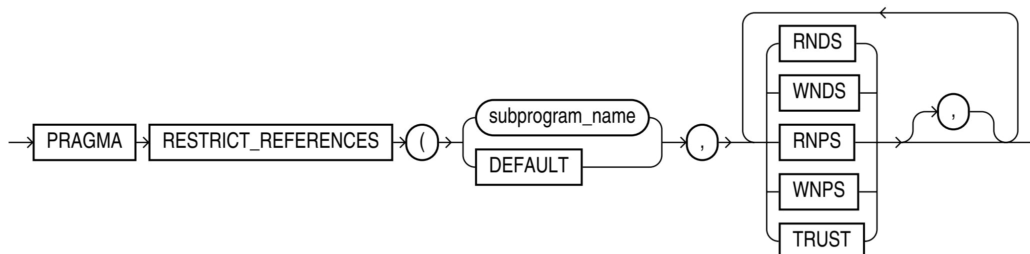
Subprograms that read or write database tables or package variables are difficult to optimize, because any call to the subprogram might produce different results or encounter errors.

Topics:

- Syntax
- Semantics
- Usage

## Syntax

***restrict_references_pragma* ::=**




## Semantics

#### *subprogram_name*

The name of a user-defined subprogram, typically a function.

If *subprogram_name* is overloaded, the pragma applies only to the most recent subprogram declaration.

#### DEFAULT

Specifies that the pragma applies to all subprograms in the package specification or ADT specification (including the system-defined constructor for ADTs).

If you also declare the pragma for an individual subprogram, it overrides the DEFAULT pragma for that subprogram.

#### RNDS

Asserts that the subprogram reads no database state (does not query database tables).

#### WNDS

Asserts that the subprogram writes no database state (does not modify tables).

#### RNPS

Asserts that the subprogram reads no package state (does not reference the values of packaged variables)

You cannot specify `RNPS` if the subprogram invokes the `SQLCODE` or `SQLERRM` function.

#### WNPS

Asserts that the subprogram writes no package state (does not change the values of packaged variables).

You cannot specify `WNPS` if the subprogram invokes the `SQLCODE` or `SQLERRM` function.

#### TRUST

Asserts that the subprogram can be trusted not to violate one or more rules.

When you specify `TRUST`, the subprogram body is not checked for violations of the constraints listed in the pragma. The subprogram is trusted not to violate them. Skipping these checks can improve performance. `TRUST` is needed for functions written in C or Java that are invoked from PL/SQL, since PL/SQL cannot verify them at run time.

## Usage

A `RESTRICT_REFERENCES` pragma can appear only in a package specification or ADT specification. Typically, this pragma is specified for functions. If a function calls procedures, specify the pragma for those procedures also.

To invoke a subprogram from parallel queries, you must specify all four constraints—`RNDS`, `WNDS`, `RNPS`, and `WNPS`. No constraint implies another.

# RETURN Statement

The RETURN statement immediately completes the execution of a subprogram and returns control to the invoker. Execution resumes with the statement following the subprogram call. In a function, the RETURN statement also sets the function identifier to the return value.

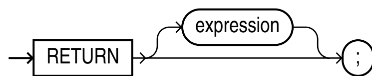
**Note:** The RETURN statement differs from the RETURN clause in a function heading, which specifies the data type of the return value.

Topics:

- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

***return_statement* ::=**




See *expression ::=* on page 13-58.

## Semantics

***expression***

Required when the RETURN statement is in a function, but not allowed when the RETURN statement is in a procedure or anonymous block.

When the RETURN statement runs, the value of *expression* is assigned to the function identifier; therefore, the value of *expression* must have a data type that is compatible with the data type in the RETURN clause of the function (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions"). For information about expressions, see "Expression" on page 13-58.

## Usage

A subprogram or anonymous block can contain multiple RETURN statements. In an anonymous block, the RETURN statement exits its own block and all enclosing blocks.

In a function, at least one execution path must lead to a RETURN statement. Otherwise, PL/SQL raises an exception at run time.

## Examples

- Example 5–44, "Returning a Record from a Function" on page 5-36

- Example 8–2, "Declaring, Defining, and Invoking a Simple PL/SQL Function" on page 8-5

## Related Topics

**In this chapter:**

- "Block" on page 13-8
- "Function" on page 13-76
- "Procedure" on page 13-103

**In other chapters:**

- "RETURN Statement" on page 8-4

# RETURNING INTO Clause

The RETURNING INTO clause specifies the variables in which to store the values returned by the statement to which the clause belongs. The variables can be either individual variables or collections. If the statement does not affect any rows, the values of the variables are undefined.

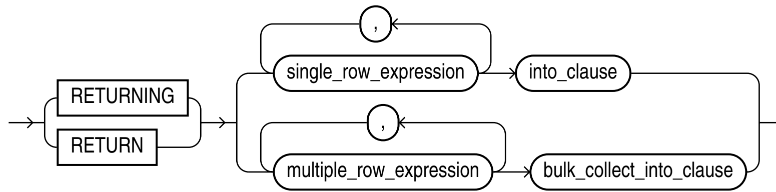
The **static** RETURNING INTO clause belongs to a DELETE, INSERT, or UPDATE statement. The **dynamic** RETURNING INTO clause belongs to the EXECUTE IMMEDIATE statement.

Topics:

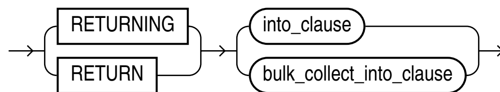
- Syntax
- Semantics
- Usage
- Examples
- Related Topics

## Syntax

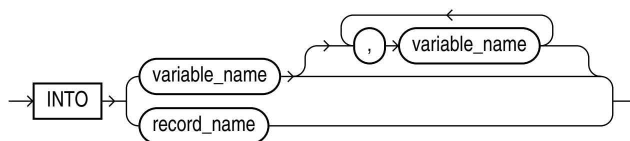
***static_returning_clause* ::=**



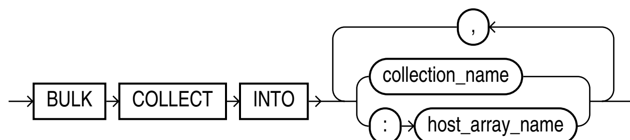

***dynamic_returning_clause* ::=**




***into_clause* ::=**




***bulk_collect_into_clause* ::=**



## Semantics

### *bulk_collect_into_clause*

Specifies one or more collections or arrays in which to store the rows that the statement returns. For each `select_item` in the statement, `bulk_collect_into_clause` must have a corresponding, type-compatible `collection_name` or `host_array_name`.

For the reason to use this clause, see "Reducing Loop Overhead with Bulk SQL" on page 12-10 on page 12-10.

**Restriction on *bulk_collect_into_clause*** Use the `bulk_collect_into_clause` clause in `dynamic_returning_clause` if and only if `dynamic_sql_stmt` (which appears in "EXECUTE IMMEDIATE Statement" on page 13-49) can return multiple rows.

### *collection_name*

The name of a collection in which to store the rows that the statement returns.

**Restrictions on *collection_name***

- The `collection_name` cannot be the name of an associative array that is indexed by a string.
- When the statement requires implicit data type conversions, `collection_name` cannot be the name of a collection of a composite type.

### *host_array_name*

The name of an array in which to store the rows that the statement returns. The array must be declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. Do not put space between the colon (:) and `host_array_name`.

### *into_clause*

Specifies the variables or record in which to store the column values that the statement returns.

**Restriction on *into_clause*** Use `into_clause` in `dynamic_returning_clause` if and only if `dynamic_sql_stmt` (which appears in "EXECUTE IMMEDIATE Statement" on page 13-49) returns a single row.

### *multiple_row_expression*

An expression that returns multiple rows of a table.

### *record_name*

The name of a record in which to store the row that the statement returns. For each `select_item` in the statement, the record must have a corresponding, type-compatible field.

### *single_row_expression*

An expression that returns a single row of a table.

### *variable_name*

Either the name of a variable in which to store a column that the statement returns or the name of a weakly typed cursor variable that is declared in a PL/SQL host

environment and passed to PL/SQL as a bind argument. Each *select_item* in the statement must have a corresponding, type-compatible variable.

**Restriction on *variable_name*** The *variable_name* cannot be the name of a BOOLEAN variable.

## Usage

You cannot use the RETURNING INTO clause for remote or parallel deletes.

For DML statements that have a RETURNING clause, you can place OUT bind 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 or define variables replace corresponding placeholders in the dynamic SQL statement. Every placeholder must be associated with a bind argument in the USING clause or RETURNING INTO clause (or both) or with a define variable in the INTO clause.

The value a of bind argument cannot be a Boolean literal (TRUE, FALSE, or NULL). To pass the value NULL to the dynamic SQL statement, see "Uninitialized Variable for NULL in USING Clause" on page 7-4.

## Examples


- Example 5–54, "RETURNING INTO Clause with Record" on page 5-43
- Example 6–1, "Static SQL Statements" on page 6-2
- Example 12–19, "BULK COLLECT with RETURNING INTO Clause" on page 12-27
- Example 12–20, "FORALL with BULK COLLECT" on page 12-28


## Related Topics

**In this chapter:**


- "EXECUTE IMMEDIATE Statement" on page 13-49
- "FETCH Statement" on page 13-67
- "SELECT INTO Statement" on page 13-118


**In other chapters:**


- "EXECUTE IMMEDIATE Statement" on page 7-2

# %ROWTYPE Attribute

The `%ROWTYPE` attribute lets you declare a record that represents a row of a database table or view: For every column of the row, the record has a field with the same name and data type.

The record fields do not inherit the constraints or initial values of the corresponding columns.

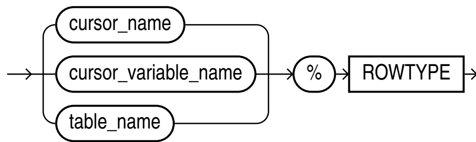
If the structure of the row changes, the structure of the record changes accordingly.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***%rowtype_attribute* ::=**




## Semantics

***cursor_name***
The name of an explicit cursor.

***cursor_variable_name***
The name of a strongly typed cursor variable.

***table_name***
The name of a database table or view that is accessible when the declaration is elaborated.

## Examples

- Example 2–15, "Declaring Record that Represents Table Row" on page 2-16
- Example 2–16, "Declaring a Record that Represents a Subset of Table Columns" on page 2-17
- Example 2–17, "Declaring a Record that Represents a Row from a Join" on page 2-17
- Example 3–17, "Column Constraints Inherited by Subtypes" on page 3-26
- Example 5–7, "Specifying Collection Element Types with %TYPE and %ROWTYPE" on page 5-9

- Example 5–20, "Assigning Values to VARRAYs with Complex Data Types" on page 5-16
- Example 5–42, "Declaring and Initializing Record Types" on page 5-34
- Example 6–25, "Cursor Variables Returning %ROWTYPE Variables" on page 6-27
- Example 6–26, "%ROWTYPE Attribute Provides Data Type" on page 6-27

## Related Topics

**In this chapter:**

- "Cursor Variable" on page 13-39
- "Explicit Cursor" on page 13-54
- "Record" on page 13-106
- "%TYPE Attribute" on page 13-127

**In other chapters:**

- "%ROWTYPE Attribute" on page 2-16

# SELECT INTO Statement

The SELECT INTO statement retrieves values from one or more database tables (as the SQL SELECT statement does) and stores them in either variables or a record (which the SQL SELECT statement does not do).

By default, the SELECT INTO statement retrieves one or more columns from a single row. With the BULK COLLECT clause, this statement retrieves an entire result set into one or more collections. The SELECT INTO statement with the BULK COLLECT clause is faster than equivalent loops with FETCH statements. For more information, see "Reducing Loop Overhead with Bulk SQL" on page 12-10.

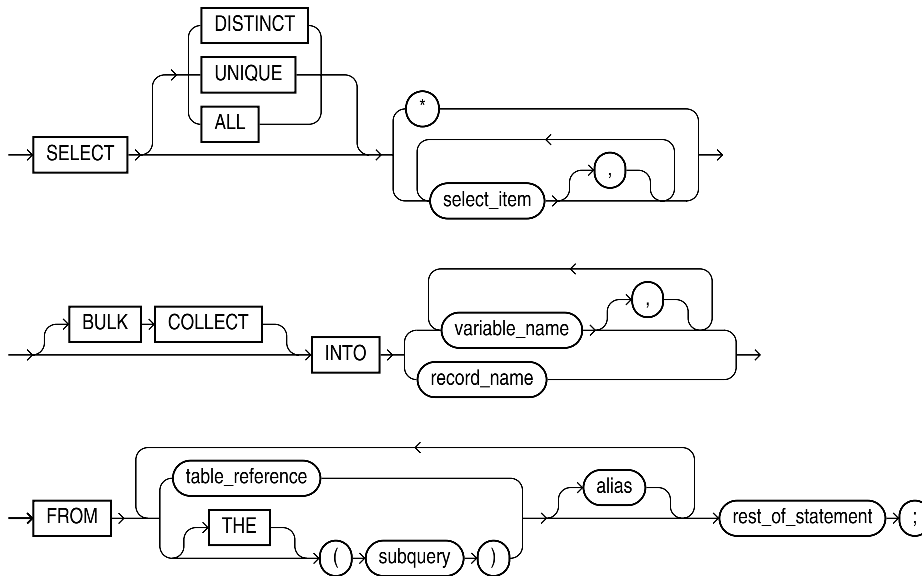
> **See Also:** *Oracle Database SQL Language Reference* for the syntax of the SQL SELECT statement

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

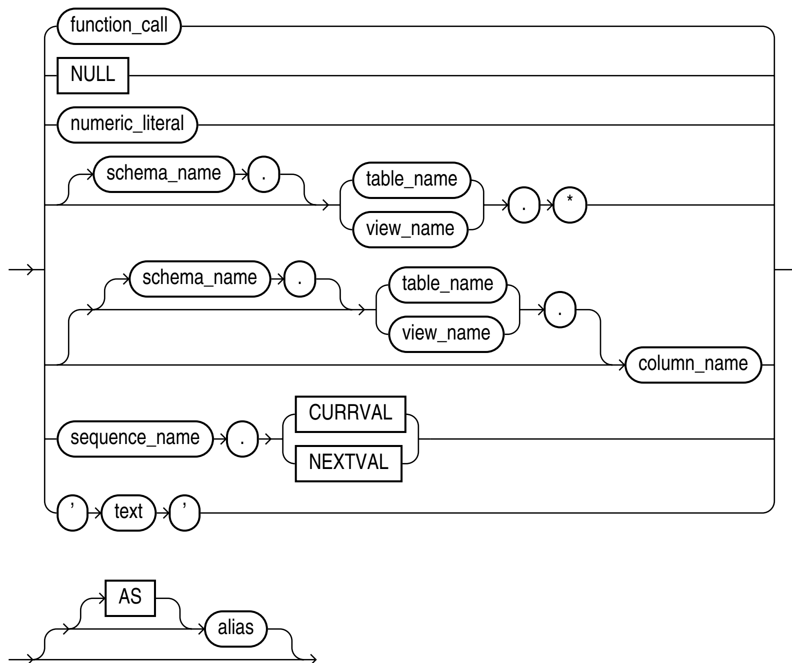
## Syntax

### *select_into_statement* ::=



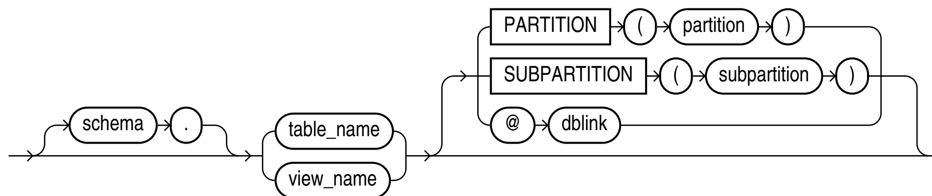

See *table_reference ::=* on page 13-119.

***select_item* ::=**




See *function_call ::=* on page 13-61.

***table_reference* ::=**




## Semantics

### *alias*

Another (usually short) name for the referenced column, table, or view.

### BULK COLLECT

Enables the `SELECT INTO` statement to return multiple rows, which it stores in one or more collections of records (which must already exist). You can declare associative arrays or nested tables that grow as needed to hold the entire result set.

Without `BULK COLLECT`, the `SELECT INTO` statement must return only one row. Otherwise, PL/SQL raises the predefined exception `TOO_MANY_ROWS` and the values of the variables in the `INTO` clause are undefined.

### *numeric_literal*

A literal that represents a number or a value that can be implicitly converted to a number (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions" on page 3-29).

#### *record_name*

A user-defined or %ROWTYPE record into which rows of values are selected. The record must have a corresponding, type-compatible field for each *select_item*.

#### *rest_of_statement*

Anything that can follow the FROM clause in a SQL SELECT statement (except the SAMPLE clause). For the syntax of the SQL SELECT statement, see *Oracle Database SQL Language Reference*.

#### *schema_name*

The name of the schema that contains the table or view. The default is your own schema.

#### *select_item*

If the SELECT INTO statement returns no rows, PL/SQL raises the exception NO_DATA_FOUND. To guard against this exception, select the result of the aggregate function COUNT(*), which returns a single value even if no rows match the condition.

#### *subquery*

A SQL SELECT statement (not a PL/SQL SELECT INTO statement) that provides a set of rows for processing.

#### *subquery2*

A SQL SELECT statement (not a PL/SQL SELECT INTO statement) that returns a single column value, which must be either a nested table or a varray. (The operator TABLE tells the database that the value is a collection, not a scalar value.)

#### *table_name*

The name of a database table.

#### *table_reference*

A reference to a table or view for which you have the SELECT privilege, which is accessible when you run the SELECT INTO statement.

#### *variable_name*

The name of a variable into which a *select_item* value is fetched. For each *select_item* value returned by the query, there must be a corresponding, type-compatible variable in the list. With BULK COLLECT, *variable_name* can be the name of a collection of records.

**Restriction on *variable_name*** You cannot select into a BOOLEAN variable.

#### *view_name*

The name of a database view.

## Examples


- Example 2–26, "SELECT INTO Assigns Values to Variables" on page 2-23
- Example 6–2, "CURRVAL and NEXTVAL Pseudocolumns" on page 6-4
- Example 6–40, "ROLLBACK Statement" on page 6-39
- Example 6–41, "SAVEPOINT and ROLLBACK Statements" on page 6-41

- Example 6–48, "Declaring an Autonomous Function in a Package" on page 6-49
- Example 7–13, "Validation Checks Guarding Against SQL Injection" on page 7-16


## Related Topics

**In this chapter:**


- "Assignment Statement" on page 13-3
- "FETCH Statement" on page 13-67
- "%ROWTYPE Attribute" on page 13-116


**In other chapters:**


- "Selecting At Most One Row (SELECT INTO Statement)" on page 6-20
- "Reducing Loop Overhead with Bulk SQL" on page 12-10


> **See Also:** *Oracle Database SQL Language Reference* for information about the SQL `SELECT` statement

# SERIALLY_REUSABLE Pragma

The SERIALLY_REUSABLE pragma specifies that the package state is needed for only one call to the server (for example, an OCI call to the database or a stored procedure call through a database link). After this call, the storage for the package variables can be reused, reducing the memory overhead for long-running sessions.

This pragma is appropriate for packages that declare large temporary work areas that are used once in the same session.

Topics:

- Syntax
- Usage
- Examples
- Related Topics

## Syntax

***serially_resuable_pragma*** **::=**

→ PRAGMA → SERIALLY_REUSABLE → ( ; )

## Usage

The SERIALLY_REUSABLE pragma can appear in the specification of a bodiless package, or in both the specification and body of a package. The pragma cannot appear only in the body of a package.

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). When the call to the server ends, the memory returns to the pool. Each time the package is used, its public variables are set to their initial values (if any) or to NULL.

Trying to access a serially reusable package from a database trigger or from a PL/SQL subprogram that was invoked from a SQL statement causes an error.

## Examples

- Example 10–1, "Serially Reusable Package Specification" on page 10-5
- Example 10–2, "Serially Reusable Package Specification and Package Body" on page 10-6
- Example 10–3, "Open Cursors in Serially Reusable Packages at Call Boundaries" on page 10-7

## Related Topics

**In this chapter:**

- "AUTONOMOUS_TRANSACTION Pragma" on page 13-6
- "EXCEPTION_INIT Pragma" on page 13-43
- "INLINE Pragma" on page 13-87

- "RESTRICT_REFERENCES Pragma" on page 13-109

**In other chapters:**

- "Serially Reusable Packages" on page 10-3

# SQLCODE Function

In an exception handler, the `SQLCODE` function returns the numeric code of the exception being handled. (Outside an exception handler, `SQLCODE` returns `0`.)

For an exception that the database raises, the numeric code is the number of the associated Oracle Database error. This number is negative except for the error "no data found", whose numeric code is +100.

For a user-defined exception, the numeric code is either +1 (the default) or the Oracle Database error number associated with the exception by the `EXCEPTION_INIT` pragma.

A SQL statement cannot invoke `SQLCODE`. To use the value of `SQLCODE` in a SQL statement, assign it to a local variable first.

If a function invokes `SQLCODE`, and you use the `RESTRICT_REFERENCES` pragma to assert its purity, you cannot specify the constraints `WNPS` and `RNPS`.

Topics:

- Syntax
- Examples
- Related Topics

## Syntax

***sqlcode_function*** **::=**

```
→ SQLCODE →
```

## Examples

- Example 11–14, "Displaying SQLCODE and SQLERRM" on page 11-19

## Related Topics

**In this chapter:**


- "Block" on page 13-8
- "EXCEPTION_INIT Pragma" on page 13-43
- "Exception Handler" on page 13-47
- "RESTRICT_REFERENCES Pragma" on page 13-109
- "SQLERRM Function" on page 13-125


**In other chapters:**


- "Associating Exceptions with Numbers" on page 11-10
- "Error Code and Error Message Retrieval" on page 11-18


> **See Also:** *Oracle Database Error Messages* for a list of Oracle Database error messages and information about them, including their numbers

# SQLERRM Function

The SQLERRM function returns the error message associated with an error number.

You cannot use SQLERRM directly in a SQL statement. Assign the value of SQLERRM to a local variable first.

If a function invokes SQLERRM, and you use the RESTRICT_REFERENCES pragma to assert its purity, you cannot specify the constraints WNPS and RNPS.

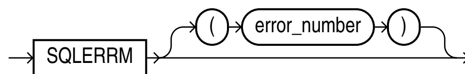
> **Note:** DBMS_UTILTY.FORMAT_ERROR_STACK is recommended over SQLERRM, except when using the FORALL statement with its SAVE EXCEPTIONS clause. For more information, see "Error Code and Error Message Retrieval" on page 11-18.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***sqlerrm_function*** **::=**




## Semantics

***error_number***

An expression whose value is an Oracle Database error number. For a list of Oracle Database error numbers, see *Oracle Database Error Messages*.

The default error number is the one associated with the current value of SQLCODE. Like SQLCODE, SQLERRM without *error_number* is useful only in an exception handler. Outside an exception handler, or if the value of *error_number* is zero, SQLERRM returns ORA-0000.

If the value of *error_number* is +100, SQLERRM returns ORA-01403.

If the value of *error_number* is a positive number other than +100, SQLERRM returns this message:

```
-error_number: non-ORACLE exception
```

If the value of *error_number* is a negative number whose absolute value is an Oracle Database error number, SQLERRM returns the error message associated with that error number. For example:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('SQLERRM(-6511): ' || TO_CHAR(SQLERRM(-6511)));
END;
```

```
/
```

Result:

```
SQLERRM(-6511): ORA-06511: PL/SQL: cursor already open
```

If the value of *error_number* is a negative number whose absolute value is not an Oracle Database error number, SQLERRM returns this message:

```
ORA-error_number: Message error_number not found;  product=RDBMS;
facility=ORA
```

For example:

```
BEGIN
  DBMS_OUTPUT.PUT_LINE('SQLERRM(-50000): ' || TO_CHAR(SQLERRM(-50000)));
END;
/
```

Result:

```
SQLERRM(-50000): ORA-50000: Message 50000 not found;  product=RDBMS;
facility=ORA
```

## Examples

- Example 11–14, "Displaying SQLCODE and SQLERRM" on page 11-19
- Example 12–11, "FORALL Statement and SQL%BULK_EXCEPTIONS" on page 12-18

## Related Topics

### In this chapter:

- "Block" on page 13-8
- "EXCEPTION_INIT Pragma" on page 13-43
- "RESTRICT_REFERENCES Pragma" on page 13-109
- "SQLCODE Function" on page 13-124

### In other chapters:

- "Error Code and Error Message Retrieval" on page 11-18

> **See Also:** *Oracle Database Error Messages* for a list of Oracle Database error messages and information about them

# %TYPE Attribute

The `%TYPE` attribute lets you declare a constant, variable, array element, record field, or subprogram parameter to be of the same data type a previously declared variable, field, record, nested table, or database column. The item declared with `%TYPE` is the **referencing item**, and the previously declared item is the **referenced item**.

The referencing item inherits the following from the referenced item:

- Data type
- Constraints (unless the referenced item is a database column)

The referencing item does not inherit the initial value of the referenced item.

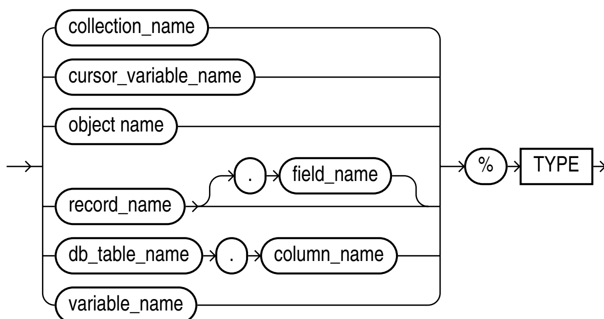
If the declaration of the referenced item changes, the declaration of the referencing item changes accordingly.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***%type_attribute*** **::=**




## Semantics

***collection_name***

The name of a collection.

***column_name***

The name of a column of a database table.

***cursor_variable_name***

The name of a cursor variable. Only the value of another cursor variable can be assigned to a cursor variable.

***db_table_name***

The name of a database table that is accessible when the declaration is elaborated.

***field_name***

The name of a record field.

***object_name***

The name of an instance of an ADT.

***record_name***

The name of a record.

***variable_name***

The name of a variable.

## Examples


- Example 2–13, "Declaring Variable of Same Type as Database Column" on page 2-15
- Example 2–14, "Declaring Variable of Same Type as Another Variable" on page 2-16
- Example 5–5, "Declaring a Procedure Parameter as a Nested Table" on page 5-9
- Example 5–7, "Specifying Collection Element Types with %TYPE and %ROWTYPE" on page 5-9
- Example 5–42, "Declaring and Initializing Record Types" on page 5-34
- Example 6–10, "Fetching Bulk Data with a Cursor" on page 6-13


## Related Topics

**In this chapter:**


- "Constant" on page 13-32
- "%ROWTYPE Attribute" on page 13-116
- "Variable" on page 13-131


**In other chapters:**


- "%TYPE Attribute" on page 2-15

# UPDATE Statement Extensions

PL/SQL extends the *update_set_clause* and *where_clause* of the SQL UPDATE statement as follows:

- In the *update_set_clause*, you can specify a record. For each selected row, the UPDATE statement updates each column with the value of the corresponding record field.
- In the *where_clause*, you can specify a CURRENT OF clause, which restricts the UPDATE statement to the current row of the specified cursor.

> **See Also:** *Oracle Database SQL Language Reference* for the syntax of the SQL UPDATE statement

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***update_set_clause* ::=**

```
→ SET → ROW → record_name →
```

***where_clause* ::=**

```
→ WHERE → CURRENT → OF → (for_update_cursor_name) →
```

## Semantics

***record_name***

The name of a record variable of type RECORD or %ROWTYPE. The record must represent a row of the item described by *dml_table_expression_clause*. That is, for every column of the row, the record must have a field with a compatible data type. If a column has a NOT NULL constraint, then its corresponding field cannot have a NULL value.

***for_update_cursor_name***

The name of a FOR UPDATE cursor; that is, an explicit cursor associated with a FOR SELECT UPDATE statement.

> **See Also:** *Oracle Database SQL Language Reference* for the complete syntax of the UPDATE statement

## Examples

- Example 5–53, "Updating Rows with a Record" on page 5-42

- Example 6–44, "FOR UPDATE Cursor in CURRENT OF Clause of UPDATE Statement" on page 6-45

## Related Topics

**In this chapter:**

- "Explicit Cursor" on page 13-54
- "Record" on page 13-106
- "%ROWTYPE Attribute" on page 13-116

**In other chapters:**

- "Updating Rows with Records" on page 5-42
- "Restrictions on Record Inserts and Updates" on page 5-43
- "SELECT FOR UPDATE and FOR UPDATE Cursors" on page 6-44

# Variable

A variable holds a value that can change.

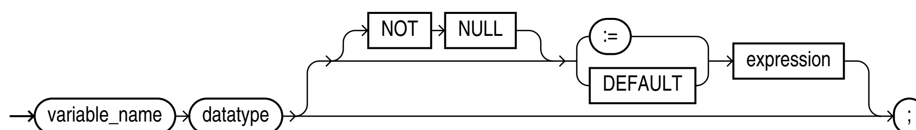
A variable declaration specifies the name, data type, and value of the variable and allocates storage for it. The declaration can also assign an initial value and impose the NOT NULL constraint.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***variable_declaration* ::=**




See:

- *datatype ::=* on page 13-32
- *expression ::=* on page 13-58

## Semantics

***expression***

The value to be assigned to the variable when the declaration is elaborated. The value of *expression* must be of a data type that is compatible with the data type of the variable (see Table 3–10, " Possible Implicit PL/SQL Data Type Conversions").

***datatype***

The data type of the variable that you are declaring.

**Restriction on *datatype*** If you use %ROWTYPE to declare a record variable, you cannot use the assignment operator (:=) or the keyword DEFAULT to assign its initial value. For information about %ROWTYPE, see "%ROWTYPE Attribute" on page 13-116. For information about assigning values to records, see "Assignments to Records" on page 5-37.

***variable_name***

The name of the variable that you are declaring.

## Examples

- Example 2–7, "Variable Declarations" on page 2-13

- Example 2–9, "Variable and Constant Declarations with Initial Values" on page 2-14
- Example 2–10, "Variable Initialized to NULL by Default" on page 2-14
- Example 2–11, "Variable Declaration with NOT NULL Constraint" on page 2-14

## Related Topics

**In this chapter:**

- "Assignment Statement" on page 13-3
- "Collection" on page 13-23
- "Constant" on page 13-32
- "Expression" on page 13-58
- "%ROWTYPE Attribute" on page 13-116
- "%TYPE Attribute" on page 13-127

**In other chapters:**

- "Variable Declarations" on page 2-12
- "Predefined PL/SQL Scalar Data Types and Subtypes" on page 3-2

# WHILE LOOP Statement

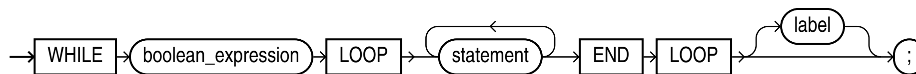
The WHILE LOOP statement runs one or more statements while a condition is TRUE. The WHILE LOOP statement ends when the condition becomes FALSE or NULL, when a statement inside the loop transfers control outside the loop, or when an exception is raised.

Topics:

- Syntax
- Semantics
- Examples
- Related Topics

## Syntax

***while_loop_statement* ::=**




See:

- *boolean_expression ::=* on page 13-59
- *statement ::=* on page 13-11

## Semantics

***boolean_expression***

This expression is evaluated at the beginning of each iteration of the loop. If its value is TRUE, the statements after LOOP run. Otherwise, control transfers to the statement after the WHILE LOOP statement.

***label***

A label that identifies *while_loop_statement* (see ***label*** on page 13-14). CONTINUE, EXIT, and GOTO statements can reference this label.

Labels improve readability, especially when LOOP statements are nested, but only if you ensure that the label at the end of the LOOP statement matches one of the labels at the beginning of the same statement (the compiler does not check).

***statement***

To prevent an infinite loop, at least one statement must change the value of *boolean_expression* to FALSE or NULL, or transfer control outside the loop. The statements that can transfer control outside the loop are:

- "CONTINUE Statement" on page 13-35 (when it transfers control to the next iteration of an enclosing labeled loop)
- "EXIT Statement" on page 13-52
- "GOTO Statement" on page 13-80
- "RAISE Statement" on page 13-105

## Examples

- Example 4–14, "WHILE LOOP Statements" on page 4-13

## Related Topics

**In this chapter:**

- "Basic LOOP Statement" on page 13-16
- "CONTINUE Statement" on page 13-35
- "Cursor FOR LOOP Statement" on page 13-37
- "EXIT Statement" on page 13-52
- "Explicit Cursor" on page 13-54
- "FETCH Statement" on page 13-67
- "FOR LOOP Statement" on page 13-70
- "FORALL Statement" on page 13-73
- "OPEN Statement" on page 13-97

**In other chapters:**

- "WHILE LOOP Statement" on page 4-13

# 14

# SQL Statements for Stored PL/SQL Units

This chapter explains how to use the SQL statements that create, change, and drop stored PL/SQL units.

For instructions for reading the syntax diagrams in this chapter, see *Oracle Database SQL Language Reference*.

## CREATE [OR REPLACE] Statements

Each of these SQL statements creates a PL/SQL unit and stores it in the database:

- CREATE FUNCTION Statement
- CREATE LIBRARY Statement
- CREATE PACKAGE Statement
- CREATE PACKAGE BODY Statement
- CREATE PROCEDURE Statement
- CREATE TRIGGER Statement
- CREATE TYPE Statement
- CREATE TYPE BODY Statement

Each of these `CREATE` statements has an optional `OR REPLACE` clause. Specify `OR REPLACE` to re-create an existing PL/SQL unit—that is, to change its declaration or definition without dropping it, re-creating it, and regranting object privileges previously granted on it. If you redefine a PL/SQL unit, the database recompiles it.

None of these `CREATE` statements can appear in a PL/SQL block.

## ALTER Statements

To recompile an existing PL/SQL unit without re-creating it (without changing its declaration or definition), use one of these SQL statements:

- ALTER FUNCTION Statement
- ALTER LIBRARY Statement
- ALTER PACKAGE Statement
- ALTER PROCEDURE Statement
- ALTER TRIGGER Statement
- ALTER TYPE Statement

Two reasons to use an `ALTER` statement are:

- To explicitly recompile a stored unit that has become invalid, thus eliminating the need for implicit run-time recompilation and preventing associated run-time compilation errors and performance overhead.
- To recompile a unit with different compilation parameters.

  For information about compilation parameters, see "PL/SQL Units and Compilation Parameters" on page 1-11.

The ALTER TYPE statement has additional uses. For details, see "ALTER TYPE Statement" on page 14-17.

### DROP Statements

To drop an existing PL/SQL unit from the database, use one of these SQL statements:

- DROP FUNCTION Statement
- DROP LIBRARY Statement
- DROP PACKAGE Statement
- DROP PROCEDURE Statement
- DROP TRIGGER Statement
- DROP TYPE Statement
- DROP TYPE BODY Statement

# ALTER FUNCTION Statement

The ALTER FUNCTION statement explicitly recompiles a standalone stored function. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

> **Note:** This statement does not change the declaration or definition of an existing function. To redeclare or redefine a standalone stored function, use the "CREATE FUNCTION Statement" on page 14-32 with the OR REPLACE clause.

Topics:

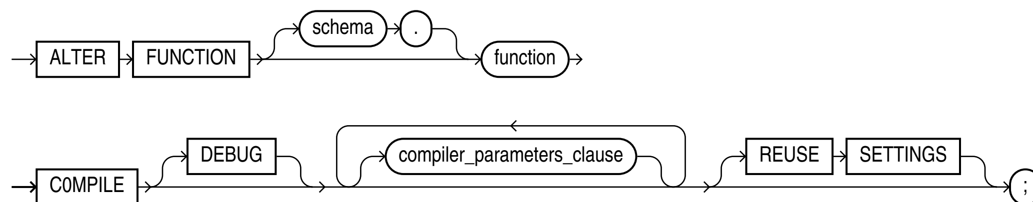
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

## Prerequisites

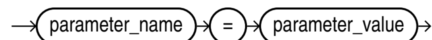
If the function is in the SYS schema, you must be connected as SYSDBA. Otherwise, the function must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

## Syntax

***alter_function*::=**




***compiler_parameters_clause*::=**




## Semantics

***schema***

The name of the schema containing the function. The default is your own schema.

***function***

The name of the function to be recompiled.

### COMPILE

Recompiles the function, whether it is valid or invalid.

First, if any of the objects upon which the function depends are invalid, the database recompiles them.

The database also invalidates any local objects that depend upon the function, such as subprograms that call the recompiled function or package bodies that define subprograms that call the recompiled function.

If the database recompiles the function successfully, then the function becomes valid. Otherwise, the database returns an error and the function remains invalid.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them after compilation. To avoid this process, specify the `REUSE SETTINGS` clause.

### DEBUG

Has the same effect as `PLSQL_OPTIMIZE_LEVEL=1`—instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. Oracle recommends using `PLSQL_OPTIMIZE_LEVEL=1` instead of `DEBUG`.

### *compiler_parameters_clause*

Specifies a value for one of the PL/SQL compilation parameters in Table 1–1. The compile-time value of each of these parameters is stored with the metadata of the PL/SQL unit being compiled.

You can specify each parameter only once in each statement. Each setting is valid only for the PL/SQL unit being compiled and does not affect other compilations in this session or system. To affect the entire session or system, you must set a value for the parameter using the `ALTER SESSION` or `ALTER SYSTEM` statement.

If you omit any parameter from this clause and you specify `REUSE SETTINGS`, then if a value was specified for the parameter in an earlier compilation of this PL/SQL unit, the database uses that earlier value. If you omit any parameter and either you do not specify `REUSE SETTINGS` or no value was specified for the parameter in an earlier compilation, then the database obtains the value for that parameter from the session environment.

### REUSE SETTINGS

Prevents Oracle Database from dropping and reacquiring compiler switch settings. With this clause, Oracle preserves the existing settings and uses them for the recompilation of any parameters for which values are not specified elsewhere in this statement.

## Example

**Recompiling a Function: Example** To explicitly recompile the function `get_bal` owned by the sample user `oe`, issue this statement:

```
ALTER FUNCTION oe.get_bal COMPILE;
```

If the database encounters no compilation errors while recompiling `get_bal`, then `get_bal` becomes valid. The database can subsequently run it without recompiling it at run time. If recompiling `get_bal` results in compilation errors, then the database returns an error, and `get_bal` remains invalid.

The database also invalidates all objects that depend upon `get_bal`. If you subsequently reference one of these objects without explicitly recompiling it first, then the database recompiles it implicitly at run time.

## Related Topics


- "CREATE FUNCTION Statement" on page 14-32
- "DROP FUNCTION Statement" on page 14-91

# ALTER LIBRARY Statement

The ALTER LIBRARY statement explicitly recompiles a library. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

---

**Note:** This statement does not change the declaration or definition of an existing library. To redeclare or redefine a library, use the "CREATE LIBRARY Statement" on page 14-40 with the OR REPLACE clause.

---

Topics:

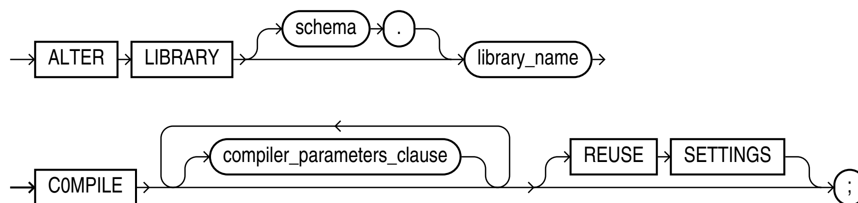
- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

## Prerequisites

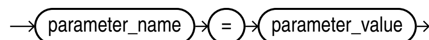
If the library is in the SYS schema, you must be connected as SYSDBA. Otherwise, the library must be in your own schema or you must have the ALTER ANY LIBRARY system privilege.

## Syntax

***alter_library*::=**




***compiler_parameters_clause*::=**




## Semantics

***library_name***

The name of the library to be recompiled.

**COMPILE**

Recompiles the library.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them after compilation. To avoid this process, specify the REUSE SETTINGS clause.

#### DEBUG

Has the same effect as PLSQL_OPTIMIZE_LEVEL=1—instructs the PL/SQL compiler to generate and store the code for use by the PL/SQL debugger. Oracle recommends using PLSQL_OPTIMIZE_LEVEL=1 instead of DEBUG.

#### *compiler_parameters_clause*

Specifies a value for one of the PL/SQL compilation parameters in Table 1–1. The compile-time value of each of these parameters is stored with the metadata of the PL/SQL unit being compiled.

You can specify each parameter only once in each statement. Each setting is valid only for the PL/SQL unit being compiled and does not affect other compilations in this session or system. To affect the entire session or system, you must set a value for the parameter using the ALTER SESSION or ALTER SYSTEM statement.

If you omit any parameter from this clause and you specify REUSE SETTINGS, then if a value was specified for the parameter in an earlier compilation of this PL/SQL unit, the database uses that earlier value. If you omit any parameter and either you do not specify REUSE SETTINGS or no value was specified for the parameter in an earlier compilation, then the database obtains the value for that parameter from the session environment.

#### REUSE SETTINGS

Prevents Oracle from dropping and reacquiring compiler switch settings. With this clause, Oracle preserves the existing settings and uses them for the recompilation of any parameters for which values are not specified elsewhere in this statement.

### Examples

**Recompiling a Library: Example** To explicitly recompile the library my_ext_lib owned by the sample user hr, issue this statement:

```
ALTER LIBRARY hr.my_ext_lib COMPILE;
```

If the database encounters no compilation errors while recompiling my_ext_lib, then my_ext_lib becomes valid. The database can subsequently run it without recompiling it at run time. If recompiling my_ext_lib results in compilation errors, then the database returns an error, and my_ext_lib remains invalid.

The database also invalidates all objects that depend upon my_ext_lib. If you subsequently reference one of these objects without explicitly recompiling it first, then the database recompiles it implicitly at run time.

### Related Topics

- "CREATE LIBRARY Statement" on page 14-40
- "DROP LIBRARY Statement" on page 14-93

# ALTER PACKAGE Statement

The ALTER PACKAGE statement explicitly recompiles a package specification, body, or both. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

Because all objects in a package are stored as a unit, the ALTER PACKAGE statement recompiles all package objects together. You cannot use the ALTER PROCEDURE statement or ALTER FUNCTION statement to recompile individually a procedure or function that is part of a package.

---

**Note:** This statement does not change the declaration or definition of an existing package. To redeclare or redefine a package, use the "CREATE PACKAGE Statement" on page 14-42, or the "CREATE PACKAGE BODY Statement" on page 14-45 with the OR REPLACE clause.

---

Topics:

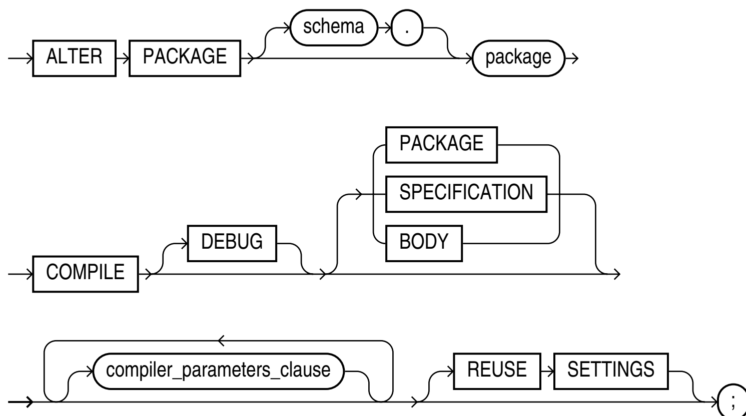
- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

## Prerequisites

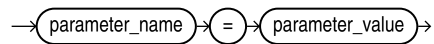
If the package is in the SYS schema, you must be connected as SYSDBA. Otherwise, the package must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

## Syntax

***alter_package*::=**



***compiler_parameters_clause*::=**




## Semantics

***schema***

The name of the schema containing the package. The default is your own schema.

***package***

The name of the package to be recompiled.

**COMPILE**

Recompiles the package specification or body.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them after compilation. To avoid this process, specify the `REUSE SETTINGS` clause.

> **See Also:** "Recompiling a Package: Examples" on page 14-10

**SPECIFICATION**

Recompiles only the package specification, whether it is valid or invalid. You might want to recompile a package specification to check for compilation errors after modifying the specification.

When you recompile a package specification, the database invalidates any local objects that depend on the specification, such as procedures that call procedures or functions in the package. The body of a package also depends on its specification. If you subsequently reference one of these dependent objects without first explicitly recompiling it, then the database recompiles it implicitly at run time.

**BODY**

Recompiles only the package body, whether it is valid or invalid. You might want to recompile a package body after modifying it. Recompiling a package body does not invalidate objects that depend upon the package specification.

When you recompile a package body, the database first recompiles the objects on which the body depends, if any of those objects are invalid. If the database recompiles the body successfully, then the body becomes valid.

**PACKAGE**

Recompiles both the package specification and (if it exists) the package body, whether they are valid or invalid. This is the default. The recompilation of the package specification and body lead to the invalidation and recompilation of dependent objects as described for `SPECIFICATION` and `BODY`.

***compiler_parameters_clause***

Has the same behavior for a package as it does for a function. See the `ALTER FUNCTION` *compiler_parameters_clause* on page 14-4.

**REUSE SETTINGS**

Has the same behavior for a package as it does for a function. See REUSE SETTINGS on page 14-4.

## Examples

**Recompiling a Package: Examples** This statement explicitly recompiles the specification and body of the `hr.emp_mgmt` package. See "Creating a Package: Example" on page 14-43 for the example that creates this package.

```
ALTER PACKAGE emp_mgmt COMPILE PACKAGE;
```

If the database encounters no compilation errors while recompiling the `emp_mgmt` specification and body, then `emp_mgmt` becomes valid. The user `hr` can subsequently call or reference all package objects declared in the specification of `emp_mgmt` without run-time recompilation. If recompiling `emp_mgmt` results in compilation errors, then the database returns an error and `emp_mgmt` remains invalid.

The database also invalidates all objects that depend upon `emp_mgmt`. If you subsequently reference one of these objects without explicitly recompiling it first, then the database recompiles it implicitly at run time.

To recompile the body of the `emp_mgmt` package in the schema `hr`, issue this statement:

```
ALTER PACKAGE hr.emp_mgmt  COMPILE BODY;
```

If the database encounters no compilation errors while recompiling the package body, then the body becomes valid. The user `hr` can subsequently call or reference all package objects declared in the specification of `emp_mgmt` without run-time recompilation. If recompiling the body results in compilation errors, then the database returns an error message and the body remains invalid.

Because this statement recompiles the body and not the specification of `emp_mgmt`, the database does not invalidate dependent objects.

## Related Topics


- "CREATE PACKAGE Statement" on page 14-42
- "DROP PACKAGE Statement" on page 14-94

# ALTER PROCEDURE Statement

The ALTER PROCEDURE statement explicitly recompiles a standalone stored procedure. Explicit recompilation eliminates the need for implicit run-time recompilation and prevents associated run-time compilation errors and performance overhead.

To recompile a procedure that is part of a package, recompile the entire package using the "ALTER PACKAGE Statement" on page 14-8).

> **Note:** This statement does not change the declaration or definition of an existing procedure. To redeclare or redefine a standalone stored procedure, use the "CREATE PROCEDURE Statement" on page 14-49 with the OR REPLACE clause.

The ALTER PROCEDURE statement is very similar to the ALTER FUNCTION statement. See "ALTER FUNCTION Statement" on page 14-3 for more information.

Topics:

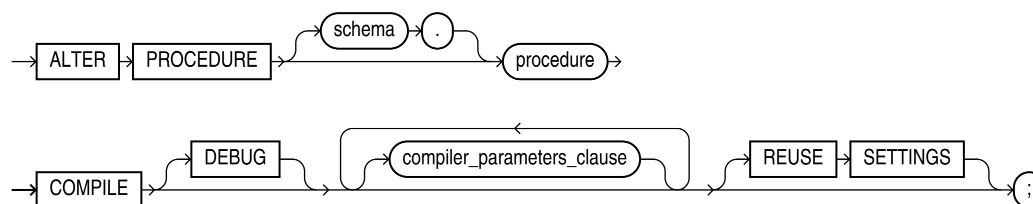

- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics


## Prerequisites

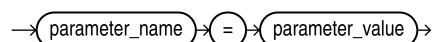
If the procedure is in the SYS schema, you must be connected as SYSDBA. Otherwise, the procedure must be in your own schema or you must have ALTER ANY PROCEDURE system privilege.

## Syntax

***alter_procedure*::=**




***compiler_parameters_clause*::=**




## Semantics

### *schema*

The name of the schema containing the procedure. The default is your own schema.

#### *procedure*

The name of the procedure to be recompiled.

#### COMPILE

Recompiles the procedure, whether it is valid or invalid.

First, if any of the objects upon which the procedure depends are invalid, the database recompiles them.

The database also invalidates any local objects that depend upon the procedure, such as subprograms that call the recompiled procedure or package bodies that define subprograms that call the recompiled procedure.

If the database recompiles the procedure successfully, then the procedure becomes valid. Otherwise, the database returns an error and the procedure remains invalid.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them after compilation. To avoid this process, specify the `REUSE SETTINGS` clause.

> **See Also:** "Recompiling a Procedure: Example" on page 14-12

#### DEBUG

Has the same behavior for a procedure as it does for a function. See DEBUG on page 14-4.

> **See Also:** *Oracle Database Advanced Application Developer's Guide* for information about debugging procedures

#### *compiler_parameters_clause*

Has the same behavior for a procedure as it does for a function. See the `ALTER FUNCTION` *compiler_parameters_clause* on page 14-4.

#### REUSE SETTINGS

Has the same behavior for a procedure as it does for a function. See REUSE SETTINGS on page 14-4.

## Example

**Recompiling a Procedure: Example** To explicitly recompile the procedure `remove_emp` owned by the user `hr`, issue this statement:

```
ALTER PROCEDURE hr.remove_emp COMPILE;
```

If the database encounters no compilation errors while recompiling `remove_emp`, then `remove_emp` becomes valid. The database can subsequently run it without recompiling it at run time. If recompiling `remove_emp` results in compilation errors, then the database returns an error and `remove_emp` remains invalid.

the database also invalidates all dependent objects. These objects include any procedures, functions, and package bodies that call `remove_emp`. If you subsequently reference one of these objects without first explicitly recompiling it, then the database recompiles it implicitly at run time.

## Related Topics

- "CREATE PROCEDURE Statement" on page 14-49

- "DROP PROCEDURE Statement" on page 14-96

# ALTER TRIGGER Statement

The ALTER TRIGGER statement enables, disables, or compiles a database trigger.

> **Note:** This statement does not change the declaration or definition of an existing trigger. To redeclare or redefine a trigger, use the "CREATE TRIGGER Statement" on page 14-53 with the OR REPLACE clause.

Topics:

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

## Prerequisites

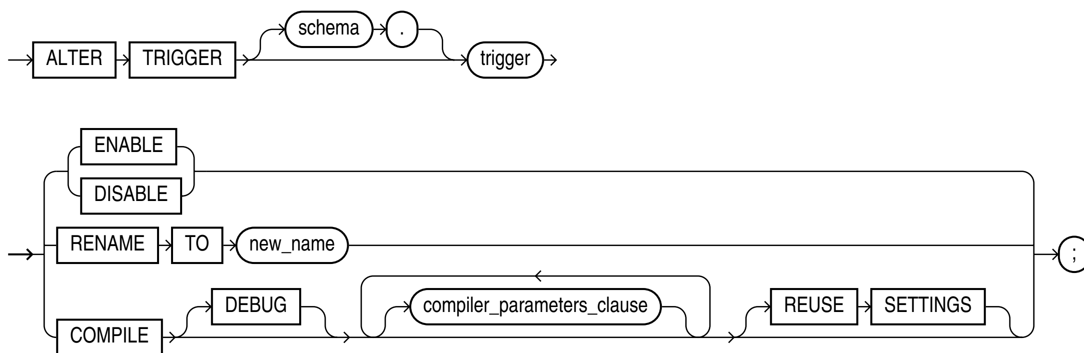
If the trigger is in the SYS schema, you must be connected as SYSDBA. Otherwise, the trigger must be in your own schema or you must have ALTER ANY TRIGGER system privilege.

In addition, to alter a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER system privilege.

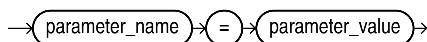
> **See Also:** "CREATE TRIGGER Statement" on page 14-53 for more information about triggers based on DATABASE triggers

## Syntax

***alter_trigger*::=**




***compiler_parameters_clause*::=**



## Semantics

***schema***

The name of the schema containing the trigger. The default is your own schema.

***trigger***

The name of the trigger to be altered.

**ENABLE**

Enables the trigger. You can also use the ENABLE ALL TRIGGERS clause of ALTER TABLE to enable all triggers associated with a table. See the ALTER TABLE statement in *Oracle Database SQL Language Reference*.

**DISABLE**

Disables the trigger. You can also use the DISABLE ALL TRIGGERS clause of ALTER TABLE to disable all triggers associated with a table.

> **See Also:**
>
> - "Enabling Triggers: Example" on page 14-16
> - "Disabling Triggers: Example" on page 14-16

**RENAME TO *new_name***

Renames the trigger. The database renames the trigger and leaves it in the same state it was in before being renamed.

When you rename a trigger, the database rebuilds the remembered source of the trigger in the USER_SOURCE, ALL_SOURCE, and DBA_SOURCE data dictionary views. As a result, comments and formatting may change in the TEXT column of those views even though the trigger source did not change.

**COMPILE**

Recompiles the trigger, whether it is valid or invalid.

First, if any of the objects upon which the trigger depends are invalid, the database recompiles them.

If the database recompiles the trigger successfully, then the trigger becomes valid. Otherwise, the database returns an error and the trigger remains invalid.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them after compilation. To avoid this process, specify the REUSE SETTINGS clause.

**DEBUG**

Has the same behavior for a trigger as it does for a function. See DEBUG on page 14-4.

> **See Also:** *Oracle Database Advanced Application Developer's Guide* for information about debugging a trigger using the same facilities available for stored subprograms

***compiler_parameters_clause***

Has the same behavior for a trigger as it does for a function. See the ALTER FUNCTION *compiler_parameters_clause* on page 14-4.

### REUSE SETTINGS

Has the same behavior for a trigger as it does for a function. See REUSE SETTINGS on page 14-4.

## Examples

**Disabling Triggers: Example** The sample schema `hr` has a trigger named `update_job_history` created on the `employees` table. The trigger is fired whenever an `UPDATE` statement changes an employee's `job_id`. The trigger inserts into the `job_history` table a row that contains the employee's ID, begin and end date of the last job, and the job ID and department.

When this trigger is created, the database enables it automatically. You can subsequently disable the trigger with this statement:

```
ALTER TRIGGER update_job_history DISABLE;
```

When the trigger is disabled, the database does not fire the trigger when an `UPDATE` statement changes an employee's job.

**Enabling Triggers: Example** After disabling the trigger, you can subsequently enable it with this statement:

```
ALTER TRIGGER update_job_history ENABLE;
```

After you reenable the trigger, the database fires the trigger whenever an `UPDATE` statement changes an employee's job. If an employee's job is updated while the trigger is disabled, then the database does not automatically fire the trigger for this employee until another transaction changes the `job_id` again.

## Related Topics

- "CREATE TRIGGER Statement" on page 14-53
- "DROP TRIGGER Statement" on page 14-98

# ALTER TYPE Statement

The ALTER TYPE statement does one of the following to a type that was created with "CREATE TYPE Statement" on page 14-69 and "CREATE TYPE BODY Statement" on page 14-86:

- **Evolves** the type; that is, adds or drops member attributes or methods.

  For more information about type evolution, see *Oracle Database Object-Relational Developer's Guide*.
- Changes the specification of the type by adding object member subprogram specifications.
- Recompiles the specification or body of the type.
- Resets the version of the type to 1, so that it is no longer considered to be evolved.

Topics:

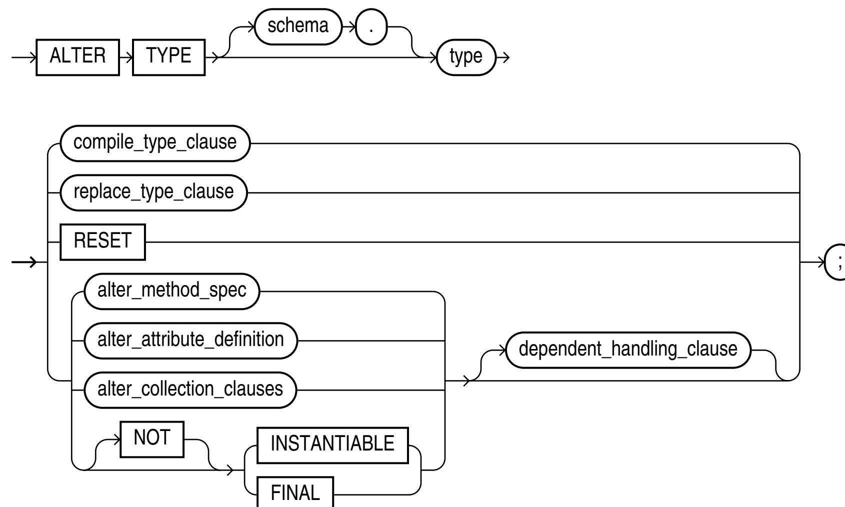
- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

## Prerequisites

If the type is in the SYS schema, you must be connected as SYSDBA. Otherwise, the type must be in your own schema and you must have CREATE TYPE or CREATE ANY TYPE system privilege, or you must have ALTER ANY TYPE system privileges.

## Syntax

***alter_type*::=**

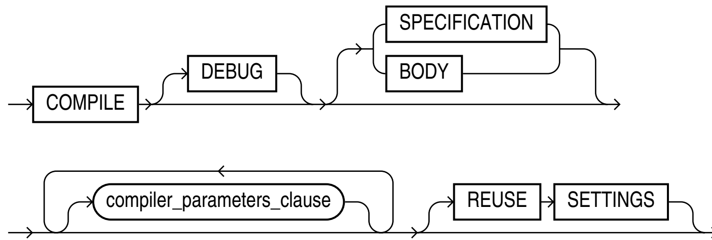



See:

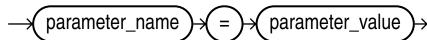
- *alter_attribute_definition::=* on page 14-20

- *alter_method_spec::=* on page 14-20
- *alter_collection_clauses::=* on page 14-20
- *compile_type_clause::=* on page 14-18
- *dependent_handling_clause::=* on page 14-20
- *replace_type_clause::=* on page 14-18

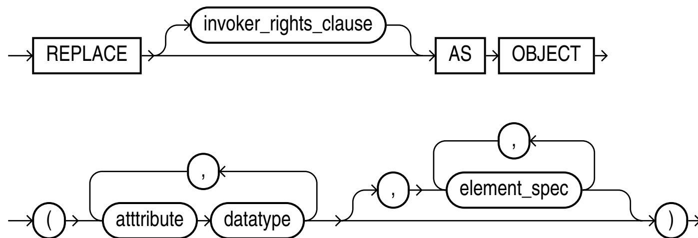
### *compile_type_clause*::=



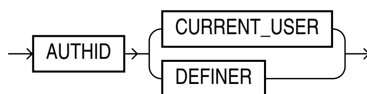

### *compiler_parameters_clause*::=



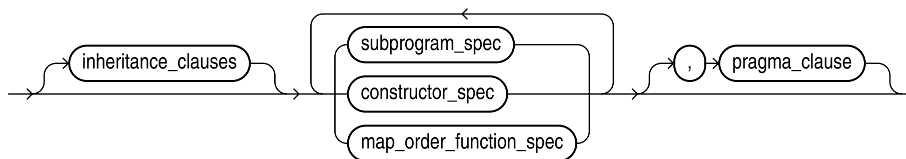

### *replace_type_clause*::=




### *invoker_rights_clause*::=

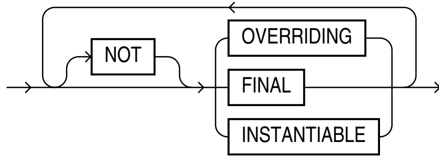



### *element_spec*::=

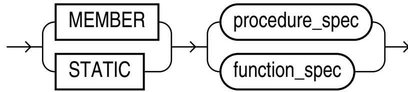



(*inheritance_clauses::=* on page 14-19, *subprogram_spec::=* on page 14-19, *constructor_spec::=* on page 14-19, *map_order_function_spec::=* on page 14-19, *pragma_clause::=* on page 14-20)

***inheritance_clauses*::=**

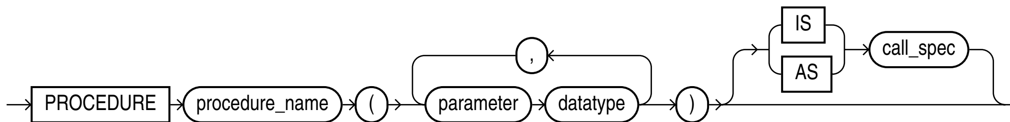



***subprogram_spec*::=**

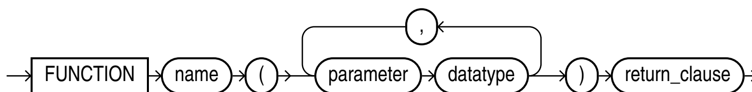



(*procedure_spec::=* on page 14-19, *function_spec::=* on page 14-19)

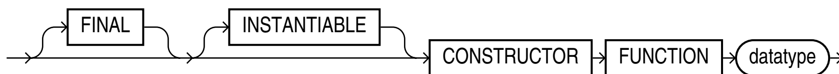
***procedure_spec*::=**

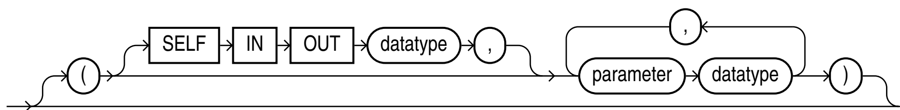


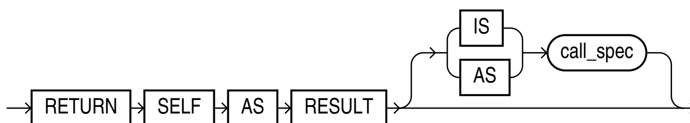

***function_spec*::=**



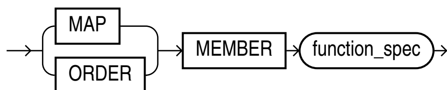

***constructor_spec*::=**



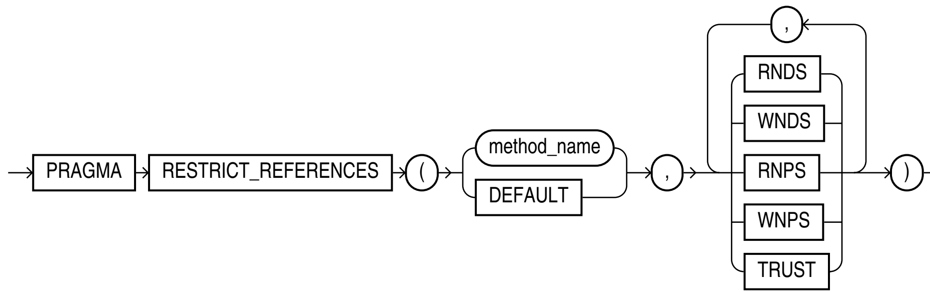




***map_order_function_spec*::=**

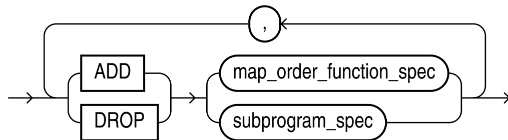



(*function_spec::=* on page 14-19)

**pragma_clause::=**

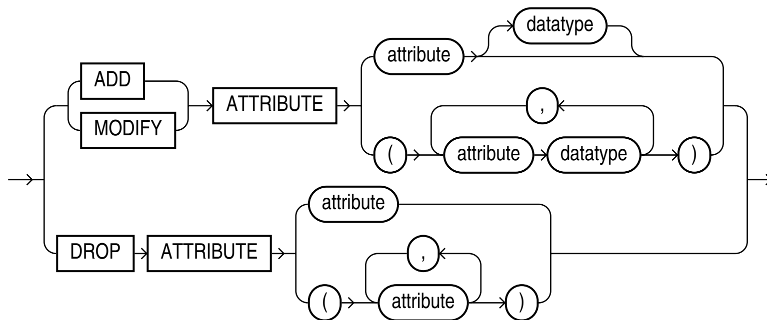



**alter_method_spec::=**

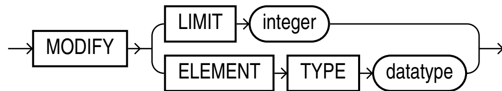



(*map_order_function_spec::=* on page 14-19, *subprogram_spec::=* on page 14-19)

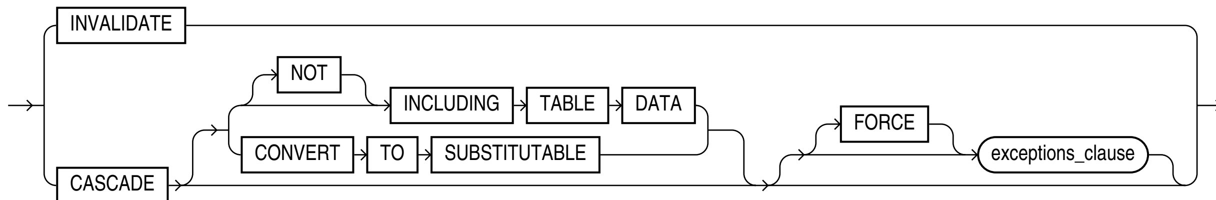
**alter_attribute_definition::=**



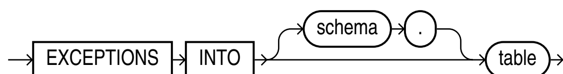

**alter_collection_clauses::=**




**dependent_handling_clause::=**




**exceptions_clause::=**



## Semantics

#### *schema*
The name of the schema containing the type. The default is your own schema.

#### *type*
The name of an ADT, varray type, or nested table type.

**Restriction on *type*** You cannot evolve an editioned ADT.

The ALTER TYPE statement fails with ORA-22348 if either of the following is true:

- *type* is an editioned ADT and the ALTER TYPE statement has no *compile_type_clause*.

  (You can use the ALTER TYPE statement to recompile an editioned object type, but not for any other purpose.)

- *type* has a dependent that is an editioned ADT and the ALTER TYPE statement has a CASCADE clause.

An **editioned object** is a schema object that has an editionable object type and was created by a user for whom editions are enabled. For more information about editioned objects, see *Oracle Database Advanced Application Developer's Guide*.

#### *compile_type_clause*
Recompiles the type specification and body. This is the default if neither SPECIFICATION nor BODY is specified.

During recompilation, the database drops all persistent compiler switch settings, retrieves them again from the session, and stores them after compilation. To avoid this process, specify the REUSE SETTINGS clause.

If recompiling the type results in compilation errors, then the database returns an error and the type remains invalid. You can see the associated compiler error messages with the SQL*Plus command SHOW ERRORS.

> **See Also:**
>
> - "Recompiling a Type: Example" on page 14-29
> - "Recompiling a Type Specification: Example" on page 14-29

#### DEBUG
Has the same behavior for a type as it does for a function. See DEBUG on page 14-4.

#### SPECIFICATION
Recompiles only the type specification.

#### BODY
Recompiles only the type body.

#### *compiler_parameters_clause*
Has the same behavior for a type as it does for a function. See the ALTER FUNCTION *compiler_parameters_clause* on page 14-4.

### REUSE SETTINGS

Has the same behavior for a type as it does for a function. See REUSE SETTINGS on page 14-4.

### *replace_type_clause*

Adds member subprogram specifications.

**Restriction on *replace_type_clause*** This clause is valid only for ADTs, not for nested tables or varrays.

### *attribute*

The name of an object attribute. Attributes are data items with a name and a type specifier that form the structure of the object.

### *element_spec*

The elements of the redefined object.

### *inheritance_clauses*

Specifies the relationship between supertypes and subtypes.

### *subprogram_spec*

The `MEMBER` and `STATIC` clauses let you specify for the ADT a function or procedure subprogram which is referenced as an attribute.

You must specify a corresponding method body in the ADT body for each procedure or function specification.

> **See Also:**
>
> - "CREATE TYPE Statement" on page 14-69 for a description of the difference between member and static methods, and for examples
> - "CREATE TYPE BODY Statement" on page 14-86
> - "Overloaded Subprograms" on page 8-13 for information about overloading subprogram names in a package

### *procedure_spec*

The specification of a procedure subprogram.

### *function_spec*

The specification of a function subprogram.

### *pragma_clause*

Denies member functions read/write access to database tables, packaged variables, or both, and thereby helps to avoid side effects.

> **Note:** This clause is deprecated. Oracle recommends against using this clause unless you must do so for backward compatibility of your applications. The database now runs purity checks at run time. If you must use this clause for backward compatibility of your applications, see its description in "CREATE TYPE Statement" on page 14-69.

**Restriction on *pragma_clause*** The *pragma_clause* is not valid when dropping a method.

> **See Also:** *Oracle Database Advanced Application Developer's Guide* for more information about pragmas

### *map_order_function_spec*

You can declare either one MAP method or one ORDER method, regardless how many MEMBER or STATIC methods you declare. However, a subtype can override a MAP method if the supertype defines a NOT FINAL MAP method. If you declare either method, then you can compare object instances in SQL.

If you do not declare either method, then you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method must be specified to determine the equality of two ADTs.

- For MAP, specify a member function (MAP method) that returns the relative position of a given instance in the ordering of all instances of the object. A map method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined scalar type. The database uses the ordering for comparison conditions and ORDER BY clauses.

  If *type* will be referenced in queries involving sorts (through ORDER BY, GROUP BY, DISTINCT, or UNION clauses) or joins, and you want those queries to be parallelized, then you must specify a MAP member function.

  If the argument to the MAP method is null, then the MAP method returns null and the method is not invoked.

  An object specification can contain only one MAP method, which must be a function. The result type must be a predefined SQL scalar type, and the MAP function can have no arguments other than the implicit SELF argument.

  A subtype cannot define a new MAP method, but it can override an inherited MAP method.

- For ORDER, specify a member function (ORDER method) that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, zero, or positive value indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

  If either argument to the ORDER method is null, then the ORDER method returns null and the method is not invoked.

  When instances of the same ADT definition are compared in an ORDER BY clause, the ORDER method function is invoked.

  An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

  A subtype cannot define an ORDER method, nor can it override an inherited ORDER method.

### *invoker_rights_clause*

Specifies the AUTHID property of the member functions and procedures of the ADT. For information about the AUTHID property, see "Invoker's Rights and Definer's Rights (AUTHID Property)" on page 8-19.

**Restriction on *invoker_rights_clause*** You can specify this clause only for an ADT, not for a nested table or varray.

### *alter_method_spec*

Adds a method to or drops a method from `type`. The database disables any function-based indexes that depend on the type.

In one `ALTER TYPE` statement you can add or drop multiple methods, but you can reference each method only once.

### ADD

When you add a method, its name must not conflict with any existing attributes in its type hierarchy.

**See Also:** "Adding a Member Function: Example" on page 14-28

### DROP

When you drop a method, the database removes the method from the target type.

**Restriction on DROP** You cannot drop from a subtype a method inherited from its supertype. Instead you must drop the method from the supertype.

### *subprogram_spec*

The `MEMBER` and `STATIC` clauses let you add a procedure subprogram to or drop it from the ADT.

**Restriction on *subprogram_spec*** You cannot define a `STATIC` method on a subtype that redefines a `MEMBER` method in its supertype, or vice versa.

### *map_order_function_spec*

If you declare either a `MAP` or `ORDER` method, then you can compare object instances in SQL.

**Restriction on *map_order_function_spec*** You cannot add an `ORDER` method to a subtype.

### *alter_attribute_definition*

Adds, drops, or modifies an attribute of an ADT. In one `ALTER TYPE` statement, you can add, drop, or modify multiple member attributes or methods, but you can reference each attribute or method only once.

### ADD ATTRIBUTE

The name of the attribute must not conflict with existing attributes or methods in the type hierarchy. The database adds the attribute to the end of the locally defined attribute list.

If you add the attribute to a supertype, then it is inherited by all of its subtypes. In subtypes, inherited attributes always precede declared attributes. Therefore, you might need to update the mappings of the implicitly altered subtypes after adding an attribute to a supertype.

**See Also:** "Adding a Collection Attribute: Example" on page 14-28

### DROP ATTRIBUTE

When you drop an attribute from a type, the database drops the column corresponding to the dropped attribute and any indexes, statistics, and constraints referencing the dropped attribute.

You need not specify the data type of the attribute you are dropping.

#### Restrictions on DROP ATTRIBUTE

- You cannot drop an attribute inherited from a supertype. Instead you must drop the attribute from the supertype.
- You cannot drop an attribute that is part of a partitioning, subpartitioning, or cluster key.
- You cannot drop an attribute of a primary-key-based object identifier of an object table or a primary key of an index-organized table.
- You cannot drop all of the attributes of a root type. Instead you must drop the type. However, you can drop all of the locally declared attributes of a subtype.

### MODIFY ATTRIBUTE

Modifies the data type of an existing scalar attribute. For example, you can increase the length of a `VARCHAR2` or `RAW` attribute, or you can increase the precision or scale of a numeric attribute.

**Restriction on MODIFY ATTRIBUTE** You cannot expand the size of an attribute referenced in a function-based index, domain index, or cluster key.

### [NOT] FINAL

Indicates whether any further subtypes can be created for this type:

- Specify `FINAL` if no further subtypes can be created for this type.
- Specify `NOT FINAL` if further subtypes can be created under this type.

If you change the property between `FINAL` and `NOT FINAL`, then you must specify the `CASCADE` clause of the *dependent_handling_clause* on page 14-27 to convert data in dependent columns and tables.

- If you change a type from `NOT FINAL` to `FINAL`, then you must specify `CASCADE` [`INCLUDING TABLE DATA`]. You cannot defer data conversion with `CASCADE NOT INCLUDING TABLE DATA`.
- If you change a type from `FINAL` to `NOT FINAL`, then:
  - Specify `CASCADE INCLUDING TABLE DATA` to create substitutable tables and columns of that type, but you are not concerned about the substitutability of the existing dependent tables and columns. The database marks all existing dependent columns and tables `NOT SUBSTITUTABLE AT ALL LEVELS`, so you cannot insert the subtype instances of the altered type into these existing columns and tables.
  - Specify `CASCADE CONVERT TO SUBSTITUTABLE` to create substitutable tables and columns of the type and also store subtype instances of the altered type in existing dependent tables and columns. The database marks all existing dependent columns and tables `SUBSTITUTABLE AT ALL LEVELS` except those that are explicitly marked `NOT SUBSTITUTABLE AT ALL LEVELS`.

**See Also:** *Oracle Database Object-Relational Developer's Guide* for a full discussion of ADT evolution

**Restriction on FINAL** You cannot change a user-defined type from `NOT FINAL` to `FINAL` if the type has any subtypes.

### [NOT] INSTANTIABLE

Indicates whether any object instances of this type can be constructed:

- Specify `INSTANTIABLE` if object instances of this type can be constructed.
- Specify `NOT INSTANTIABLE` if no constructor (default or user-defined) exists for this type. You must specify these keywords for any type with noninstantiable methods and for any type that has no attributes (either inherited or specified in this statement).

**Restriction on NOT INSTANTIABLE** You cannot change a user-defined type from `INSTANTIABLE` to `NOT INSTANTIABLE` if the type has any table dependents.

### RESET

Resets the version of this type to 1, so that it is no longer considered to be evolved.

---

**Note:** Resetting the version of this type to 1 invalidates all of its dependents.

---

`RESET` is intended for evolved ADTs that are preventing their owners from being editions-enabled. For information about enabling editions for users, see *Oracle Database Advanced Application Developer's Guide*.

To see the version number of an ADT, select `VERSION#` from the static data dictionary view `*_TYPE_VERSIONS`. For example:

```
SELECT Version# FROM DBA_TYPE_VERSIONS
WHERE Owner = schema
AND Name = 'type_name'
AND Type = 'TYPE'
```

For an evolved ADT, the preceding query returns multiple rows with different version numbers. `RESET` deletes every row whose version number is less than the maximum version number, and resets the version number of the remaining rows to 1—see "Evolving and Resetting an ADT: Example" on page 14-30.

**Restriction on RESET** You cannot specify `RESET` if the type has any table dependents (direct or indirect).

### *alter_collection_clauses*

These clauses are valid only for collection types.

### MODIFY LIMIT *integer*

Increases the number of elements in a varray. It is not valid for nested tables. Specify an integer greater than the current maximum number of elements in the varray.

**See Also:** "Increasing the Number of Elements of a Collection Type: Example" on page 14-29

### ELEMENT TYPE *datatype*

Increases the precision, size, or length of a scalar data type of a varray or nested table. This clause is not valid for collections of ADTs.

- For a collection of `NUMBER`, you can increase the precision or scale.
- For a collection of `RAW`, you can increase the maximum size.
- For a collection of `VARCHAR2` or `NVARCHAR2`, you can increase the maximum length.

> **See Also:** "Increasing the Length of a Collection Type: Example" on page 14-29

### *dependent_handling_clause*

Specifies how the database is to handle objects that are dependent on the modified type. If you omit this clause, then the `ALTER TYPE` statement terminates if *`type`* has any dependent type or table.

### INVALIDATE

Invalidates all dependent objects without any checking mechanism.

> **Note:** the database does not validate the type change, so use this clause with caution. For example, if you drop an attribute that is a partitioning or cluster key, then you cannot write to the table.

### CASCADE

Propagates the type change to dependent types and tables. The database terminates the statement if any errors are found in the dependent types or tables unless you also specify `FORCE`.

If you change the property of the type between `FINAL` and `NOT FINAL`, then you must specify this clause to convert data in dependent columns and tables. See "[NOT] FINAL" on page 14-25.

### INCLUDING TABLE DATA

Converts data stored in all user-defined columns to the most recent version of the column type. This is the default.

> **Note:** You must specify this clause if your column data is in Oracle database version 8.0 image format. This clause is also required if you are changing the type property between `FINAL` and `NOT FINAL`

- For each attribute added to the column type, the database adds an attribute to the data and initializes it to null.
- For each attribute dropped from the referenced type, the database removes the corresponding attribute data from each row in the table.

If you specify `INCLUDING TABLE DATA`, then all of the tablespaces containing the table data must be in read/write mode.

If you specify `NOT INCLUDING TABLE DATA`, then the database upgrades the metadata of the column to reflect the changes to the type but does not scan the dependent column and update the data as part of this `ALTER TYPE` statement. However, the

dependent column data remains accessible, and the results of subsequent queries of the data reflect the type modifications.

> **See Also:** *Oracle Database Object-Relational Developer's Guide* for more information about the implications of not including table data when modifying type attribute

### CONVERT TO SUBSTITUTABLE

Specify this clause if you are changing the type from FINAL to NOT FINAL and you want to create substitutable tables and columns of the type and also store subtype instances of the altered type in existing dependent tables and columns. See "[NOT] FINAL" on page 14-25 for more information.

### *exceptions_clause*

Specify FORCE if you want the database to ignore the errors from dependent tables and indexes and log all errors in the specified exception table. The exception table must have been created by running the DBMS_UTILITY.CREATE_ALTER_TYPE_ERROR_TABLE procedure.

## Examples

**Adding a Member Function: Example** This example uses the ADT data_typ1. See "ADT Examples" on page 14-81 for the example that creates this ADT. A method is added to data_typ1 and its type body is modified to correspond. The date formats are consistent with the order_date column of the oe.orders sample table:

```
ALTER TYPE data_typ1
   ADD MEMBER FUNCTION qtr(der_qtr DATE)
   RETURN CHAR CASCADE;

CREATE OR REPLACE TYPE BODY data_typ1 IS
  MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
  BEGIN
  RETURN (year + invent);
  END;
     MEMBER FUNCTION qtr(der_qtr DATE) RETURN CHAR IS
     BEGIN
       IF (der_qtr < TO_DATE('01-APR', 'DD-MON')) THEN
         RETURN 'FIRST';
       ELSIF (der_qtr < TO_DATE('01-JUL', 'DD-MON')) THEN
         RETURN 'SECOND';
       ELSIF (der_qtr < TO_DATE('01-OCT', 'DD-MON')) THEN
         RETURN 'THIRD';
       ELSE
         RETURN 'FOURTH';
       END IF;
     END;
   END;
/
```

**Adding a Collection Attribute: Example** This example adds the author attribute to the textdoc_tab object column of the text table. See "ADT Examples" on page 14-81 for the example that creates the underlying textdoc_typ type.

```
CREATE TABLE text (
   doc_id       NUMBER,
   description  textdoc_tab)
   NESTED TABLE description STORE AS text_store;
```

```
ALTER TYPE textdoc_typ
   ADD ATTRIBUTE (author VARCHAR2) CASCADE;
```

The CASCADE keyword is required because both the textdoc_tab and text table are dependent on the textdoc_typ type.

**Increasing the Number of Elements of a Collection Type: Example** This example increases the maximum number of elements in the varray phone_list_typ_demo. See "ADT Examples" on page 14-81 for the example that creates this type.

```
ALTER TYPE phone_list_typ_demo
  MODIFY LIMIT 10 CASCADE;
```

**Increasing the Length of a Collection Type: Example** This example increases the length of the varray element type phone_list_typ:

```
ALTER TYPE phone_list_typ
  MODIFY ELEMENT TYPE VARCHAR(64) CASCADE;
```

**Recompiling a Type: Example** This example recompiles type cust_address_typ in the hr schema:

```
ALTER TYPE cust_address_typ2 COMPILE;
```

**Recompiling a Type Specification: Example** This example compiles the type specification of link2.

```
CREATE TYPE link1 AS OBJECT
  (a NUMBER);
/
CREATE TYPE link2 AS OBJECT
  (a NUMBER,
   b link1,
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER);
/
CREATE TYPE BODY link2 AS
   MEMBER FUNCTION p(c1 NUMBER) RETURN NUMBER IS
      BEGIN
         dbms_output.put_line(c1);
         RETURN c1;
      END;
   END;
/
```

In this example, both the specification and body of link2 are invalidated because link1, which is an attribute of link2, is altered.

```
ALTER TYPE link1 ADD ATTRIBUTE (b NUMBER) INVALIDATE;
```

You must recompile the type by recompiling the specification and body in separate statements:

```
ALTER TYPE link2 COMPILE SPECIFICATION;

ALTER TYPE link2 COMPILE BODY;
```

Alternatively, you can compile both specification and body at the same time:

```
ALTER TYPE link2 COMPILE;
```

**Evolving and Resetting an ADT: Example** This example creates an ADT in the schema `Usr`, evolves that ADT, and then tries to enable editions for `Usr`, which fails. Then the example resets the version of the ADT to 1 and succeeds in enabling editions for `Usr`. To show the version numbers of the newly created, evolved, and reset ADT, the example uses the static data dictionary view `DBA_TYPE_VERSIONS`.

```
-- Create ADT in schema Usr:
create type Usr.My_ADT authid Definer is object(a1 number)

-- Show version number of ADT:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADT'
/
```

Result:

```
T
--------------------------------------------------------------------------------
1
type     My_ADT authid Definer is object(a1 number)


1 row selected.

-- Evolve ADT:
alter type Usr.My_ADT add attribute (a2 number)
/

-- Show version number of evolved ADT:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADT'
/
```

Result:

```
T
--------------------------------------------------------------------------------
1
type     My_ADT authid Definer is object(a1 number)

2
type     My_ADT authid Definer is object(a1 number)

2
 alter type     My_ADT add attribute (a2 number)


3 rows selected.

-- Try to enable editions for Usr:
alter user Usr enable editions
/
```

Result:

```
alter user Usr enable editions
*
```

```
ERROR at line 1:
ORA-38820: user has evolved object type

-- Reset version of ADT to 1:
alter type Usr.My_ADT reset
/

-- Show version number of reset ADT:
select Version#||Chr(10)||Text t
from DBA_Type_Versions
where Owner = 'USR'
and Type_Name = 'MY_ADT'
/
```

Result:

```
T
--------------------------------------------------------------------------------
1
type     My_ADT authid Definer is object(a1 number)

1
 alter type     My_ADT add attribute (a2 number)


2 rows selected.

-- Try to enable editions for Usr:
alter user Usr enable editions
/
```

Result:

```
User altered.
```

## Related Topics

- "CREATE TYPE Statement" on page 14-69
- "CREATE TYPE BODY Statement" on page 14-86
- "DROP TYPE Statement" on page 14-99

# CREATE FUNCTION Statement

The CREATE FUNCTION statement creates or replaces a standalone stored function or a call specification.

A **standalone stored function** is a function (a subprogram that returns a single value) that is stored in the database.

> **Note:** A standalone stored function that you create with the CREATE FUNCTION statement differs from a function that you declare and define in a PL/SQL block or package. For information about the latter, see "Function" on page 13-76.

A **call specification** declares a Java method or a third-generation language (3GL) subprogram so that it can be called from PL/SQL. You can also use the SQL CALL statement to call such a method or subprogram. The call specification tells the database which Java method, or which named function in which shared library, to invoke when a call is made. It also tells the database what type conversions to make for the arguments and return value.

> **Note:** To be callable from SQL statements, a stored function must obey certain rules that control side effects. See "Subprogram Side Effects" on page 8-27.

Topics:

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

## Prerequisites

To create or replace a standalone stored function in your own schema, you must have the CREATE PROCEDURE system privilege. To create or replace a standalone stored function in another user's schema, you must have the CREATE ANY PROCEDURE system privilege.

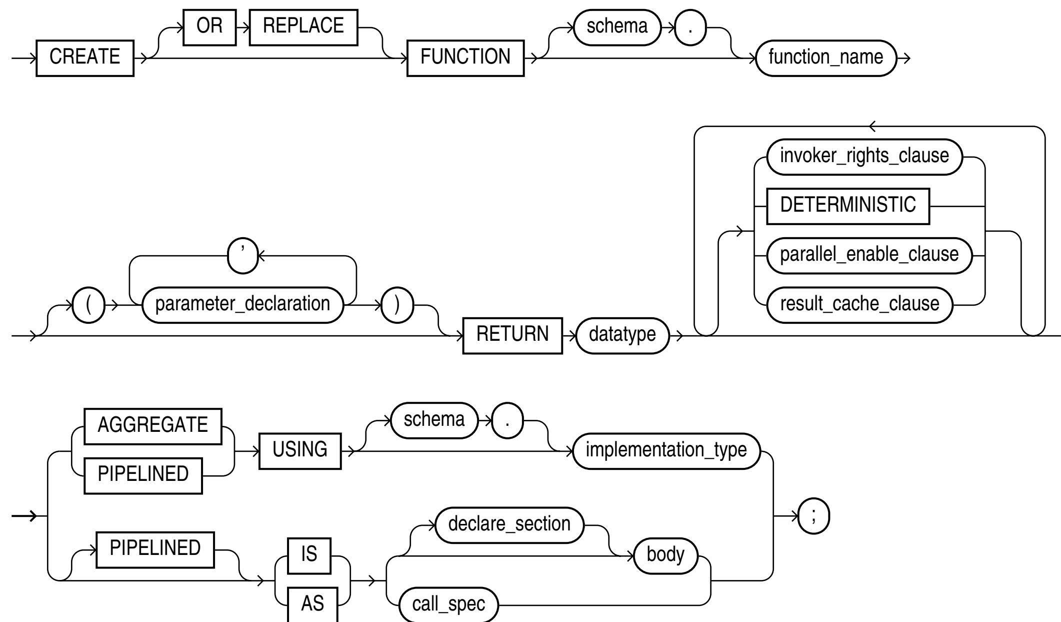
To invoke a call specification, you may need additional privileges, for example, EXECUTE privileges on a C library for a C call specification.

To embed a CREATE FUNCTION statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

> **See Also:** For more information about such prerequisites:
>
> - *Oracle Database Advanced Application Developer's Guide*
> - *Oracle Database Java Developer's Guide*

## Syntax

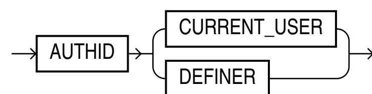
***create_function*** **::=**



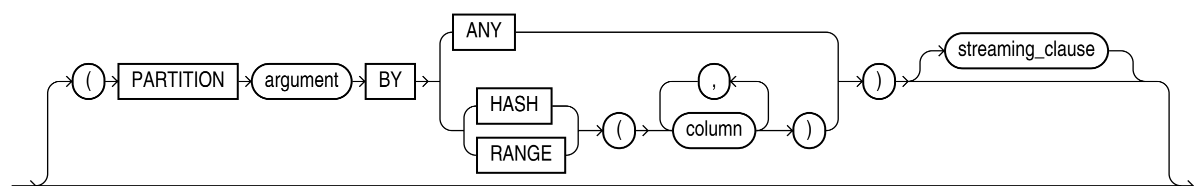

See:

- *body ::=* on page 13-11
- *datatype ::=* on page 13-32
- *declare_section ::=* on page 13-8
- *parameter_declaration ::=* on page 13-101
- *relies_on_clause ::=* on page 13-77

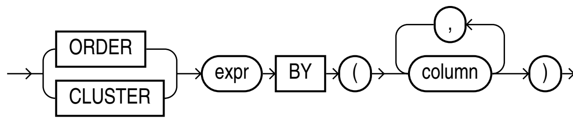
***invoker_rights_clause*** **::=**



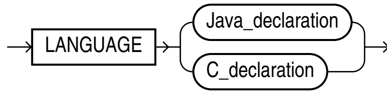

***parallel_enable_clause*** **::=**



***streaming_clause*** **::=**

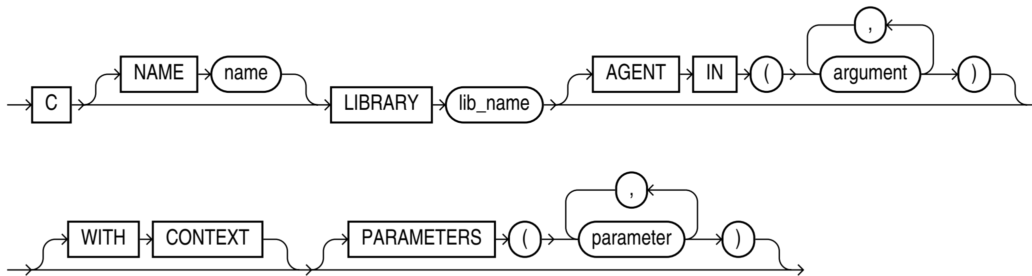



***call_spec*** **::=**




***Java_declaration*** **::=**

***C_declaration*** **::=**




## Semantics

### OR REPLACE

Re-creates the function if it exists, and recompiles it.

Users who were granted privileges on the function before it was redefined can still access the function without being regranted the privileges.

If any function-based indexes depend on the function, then the database marks the indexes `DISABLED`.

### *schema*

The name of the schema containing the function. The default is your own schema.

### *function_name*

The name of the function to be created.

### RETURN *datatype*

For *`datatype`*, specify the data type of the return value of the function. The return value can have any data type supported by PL/SQL.

---

**Note:** Oracle SQL does not support calling of functions with Boolean parameters or returns. Therefore, if SQL statements will call your user-defined functions, you must design them to return numbers (0 or 1) or character strings ('`TRUE`' or '`FALSE`').

---

The data type cannot specify a length, precision, or scale. The database derives the length, precision, or scale of the return value from the environment from which the function is called.

If the return type is ANYDATASET and you intend to use the function in the FROM clause of a query, then you must also specify the PIPELINED clause and define a describe method (ODCITableDescribe) as part of the implementation type of the function.

You cannot constrain this data type (with NOT NULL, for example).

**See Also:**

- Chapter 3, "PL/SQL Data Types," for information about PL/SQL data types
- *Oracle Database Data Cartridge Developer's Guide* for information about defining the ODCITableDescribe function

#### *invoker_rights_clause*

Specifies the AUTHID property of the function. For information about the AUTHID property, see "Invoker's Rights and Definer's Rights (AUTHID Property)" on page 8-19.

#### DETERMINISTIC

Indicates that the function returns the same result value whenever it is called with the same values for its parameters.

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 the 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, then 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 are not captured if the database chooses not to reexecute the function.

These semantic rules govern the use of the DETERMINISTIC clause:

- You can declare a schema-level subprogram DETERMINISTIC.
- You can declare a package-level subprogram DETERMINISTIC in the package specification but not in the package body.
- You cannot declare DETERMINISTIC a private subprogram (declared inside another subprogram or inside a package body).
- A DETERMINISTIC subprogram can call another subprogram whether the called program is declared DETERMINISTIC or not.

**See Also:**

- *Oracle Database Data Warehousing Guide* for information about materialized views
- *Oracle Database SQL Language Reference* for information about function-based indexes

### *parallel_enable_clause*

Indicates that the function can run from a parallel execution server of a parallel query operation. The function must not use session state, such as package variables, as those variables are not necessarily shared among the parallel execution servers.

- The optional `PARTITION` *`argument`* `BY` clause is used only with functions that have a `REF CURSOR` argument type. It lets you define the partitioning of the inputs to the function from the `REF CURSOR` argument.

  Partitioning the inputs to the function affects the way the query is parallelized when the function is used as a table function in the `FROM` clause of the query. `ANY` indicates that the data can be partitioned randomly among the parallel execution servers. Alternatively, you can specify `RANGE` or `HASH` partitioning on a specified column list.

- The optional *`streaming_clause`* lets you order or cluster the parallel processing by a specified column list.
  - `ORDER BY` indicates that the rows on a parallel execution server must be locally ordered.
  - `CLUSTER BY` indicates that the rows on a parallel execution server must have the same key values as specified by the *`column_list`*.
  - *`expr`* identifies the `REF CURSOR` parameter name of the table function on which partitioning was specified, and on whose columns you are specifying ordering or clustering for each slave in a parallel query execution.

The columns specified in all of these optional clauses refer to columns that are returned by the `REF CURSOR` argument of the function.

> **See Also:** For more information about user-defined aggregate functions:
>
> - *Oracle Database Advanced Application Developer's Guide*
> - *Oracle Database Data Cartridge Developer's Guide*

### PIPELINED { IS | USING }

Instructs the database to return the results of a **table function** iteratively. A table function returns a collection type (a nested table or varray). You query table functions by using the `TABLE` keyword before the function name in the `FROM` clause of the query. For example:

```
SELECT * FROM TABLE(function_name(...))
```

the database then returns rows as they are produced by the function.

- If you specify the keyword `PIPELINED` alone (`PIPELINED IS` ...), then the PL/SQL function body must use the `PIPE` keyword. This keyword instructs the database to return single elements of the collection out of the function, instead of returning the whole collection as a single value.
- You can specify the `PIPELINED USING` *`implementation_type`* clause to predefine an interface containing the start, fetch, and close operations. The implementation type must implement the `ODCITable` interface and must exist at the time the table function is created. This clause is useful for table functions implemented in external languages such as C++ and Java.

  If the return type of the function is `ANYDATASET`, then you must also define a describe method (`ODCITableDescribe`) as part of the implementation type of the function.

**See Also:**

- "Performing Multiple Transformations with Pipelined Table Functions" on page 12-40
- *Oracle Database Data Cartridge Developer's Guide* for information about ODCI subprograms

#### AGGREGATE USING

Identifies this function as an **aggregate function**, or one that evaluates a group of rows and returns a single row. You can specify aggregate functions in the select list, `HAVING` clause, and `ORDER BY` clause.

When you specify a user-defined aggregate function in a query, you can treat it as an **analytic function** (one that operates on a query result set). To do so, use the `OVER` *`analytic_clause`* syntax available for built-in analytic functions. See *Oracle Database SQL Language Reference* for syntax and semantics of analytic functions.

In the `USING` clause, specify the name of the implementation type of the function. The implementation type must be an ADT containing the implementation of the `ODCIAggregate` subprograms. If you do not specify *`schema`*, then the database assumes that the implementation type is in your own schema.

**Restriction on AGGREGATE USING** If you specify this clause, then you can specify only one input argument for the function.

**See Also:** *Oracle Database Data Cartridge Developer's Guide* for information about ODCI subprograms

#### *body*

The required executable part of the function and, optionally, the exception-handling part of the function.

#### *declare_section*

The optional declarative part of the function. Declarations are local to the function, can be referenced in *`body`*, and cease to exist when the function completes execution.

#### *call_spec*

Maps a Java or C method name, parameter types, and return type to their SQL counterparts. In *`Java_declaration`*, '*`string`*' identifies the Java implementation of the method.

**See Also:** *Oracle Database Java Developer's Guide*

#### EXTERNAL

In earlier releases, `EXTERNAL` was an alternative way of declaring a C method. This clause is deprecated and is supported for backward compatibility only. Oracle recommends that you use the `LANGUAGE C` syntax.

**See Also:** *Oracle Database Advanced Application Developer's Guide* for information about calling external procedures

### Examples

**Creating a Function: Examples** This statement creates the function `get_bal` on the sample table `oe.orders`:

```
CREATE FUNCTION get_bal(acc_no IN NUMBER)
   RETURN NUMBER
   IS acc_bal NUMBER(11,2);
   BEGIN
      SELECT order_total
      INTO acc_bal
      FROM orders
      WHERE customer_id = acc_no;
      RETURN(acc_bal);
    END;
/
```

The `get_bal` function returns the balance of a specified account.

When you call the function, you must specify the argument `acc_no`, the number of the account whose balance is sought. The data type of `acc_no` is `NUMBER`.

The function returns the account balance. The `RETURN` clause of the `CREATE FUNCTION` statement specifies the data type of the return value to be `NUMBER`.

The function uses a `SELECT` statement to select the `balance` column from the row identified by the argument `acc_no` in the `orders` table. The function uses a `RETURN` statement to return this value to the environment in which the function is called.

The function created in the preceding example can be used in a SQL statement. For example:

```
SELECT get_bal(165) FROM DUAL;

GET_BAL(165)
------------
        2519
```

The hypothetical following statement creates a PL/SQL standalone function `get_val` that registers the C subprogram `c_get_val` as an external function. (The parameters have been omitted from this example.)

```
CREATE FUNCTION get_val
   ( x_val IN NUMBER,
    y_val IN NUMBER,
    image IN LONG RAW )
   RETURN BINARY_INTEGER AS LANGUAGE C
      NAME "c_get_val"
      LIBRARY c_utils
      PARAMETERS (...);
```

**Creating Aggregate Functions: Example** The next statement creates an aggregate function called `SecondMax` to aggregate over number values. It assumes that the ADT `SecondMaxImpl` subprograms contains the implementations of the `ODCIAggregate` subprograms:

```
CREATE FUNCTION SecondMax (input NUMBER) RETURN NUMBER
    PARALLEL_ENABLE AGGREGATE USING SecondMaxImpl;
```

> **See Also:** *Oracle Database Data Cartridge Developer's Guide* for the complete implementation of type and type body for `SecondMaxImpl`

Use such an aggregate function in a query like this statement, which queries the sample table `hr.employees`:

```
SELECT SecondMax(salary) "SecondMax", department_id
      FROM employees
```

```
      GROUP BY department_id
      HAVING SecondMax(salary) > 9000
      ORDER BY "SecondMax", department_id;

SecondMax DEPARTMENT_ID
--------- -------------
    13500            80
    17000            90
```

**Packaged Procedure in a Function: Example** This statement creates a function that uses a DBMS_LOB.GETLENGTH procedure to return the length of a CLOB column:

```
CREATE OR REPLACE FUNCTION text_length(a CLOB)
   RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN DBMS_LOB.GETLENGTH(a);
END;
```

## Related Topics

**In this chapter:**

- "ALTER FUNCTION Statement" on page 14-3
- "CREATE PROCEDURE Statement" on page 14-49
- "DROP FUNCTION Statement" on page 14-91

**In other chapters:**

- "Function" on page 13-76 for information about creating a function in a PL/SQL block
- "Parameter" on page 13-101
- Chapter 8, "PL/SQL Subprograms"

**See Also:**

- *Oracle Database SQL Language Reference* for information about the CALL statement)
- *Oracle Database Advanced Application Developer's Guide* for information about restrictions on user-defined functions that are called from SQL statements and expressions
- *Oracle Database Advanced Application Developer's Guide* for more information about call specifications

# CREATE LIBRARY Statement

The `CREATE LIBRARY` statement creates a **library**; that is, a schema object associated with an operating-system shared library. You can use the name of this library in the *`call_spec`* of `CREATE FUNCTION` or `CREATE PROCEDURE` statements, or when declaring a function or procedure in a package or type, so that SQL and PL/SQL can call third-generation-language (3GL) functions and procedures.

Topics:

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

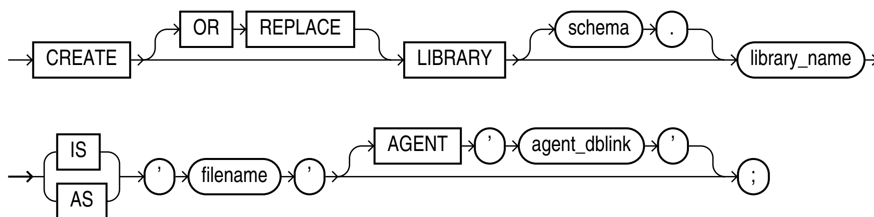
## Prerequisites

To create a library in your own schema, you must have the `CREATE LIBRARY` system privilege. To create a library in another user's schema, you must have the `CREATE ANY LIBRARY` system privilege. To use the procedures and functions stored in the library, you must have the `EXECUTE` object privilege on the library.

The `CREATE LIBRARY` statement is valid only on platforms that support shared libraries and dynamic linking.

## Syntax

***create_library*::=**




## Semantics

### OR REPLACE

Re-creates the library if it exists, and recompiles it.

Users who were granted privileges on the library before it was redefined can still access the library without being regranted the privileges.

### *schema*

The name of the schema containing the library. The default is your own schema.

### *library_name*

The name that will represent this library when a user declares a function or procedure with a *`call_spec`*.

***filename***

A string literal, enclosed in single quotation marks. This string should be the path or filename your operating system recognizes as naming the shared library.

The *`filename`* is not interpreted during execution of the `CREATE LIBRARY` statement. The existence of the library file is not checked until an attempt is made to run a routine from it.

**AGENT '*agent_dblink*'**

Causes external procedures to be run from a database link other than the server. Oracle Database will use the database link specified by *`agent_dblink`* to run external procedures. If you omit this clause, then the default agent on the server (`extproc`) will run external procedures.

## Examples

**Creating a Library: Examples** The following statement creates library `ext_lib`:

```
CREATE LIBRARY ext_lib AS '/OR/lib/ext_lib.so';
/
```

The following statement re-creates library `ext_lib`:

```
CREATE OR REPLACE LIBRARY ext_lib IS '/OR/newlib/ext_lib.so';
/
```

**Specifying an External Procedure Agent: Example** The following example creates a library `app_lib` and specifies that external procedures will be run from the public database `sales.hq.example.com`:

```
CREATE LIBRARY app_lib as '${ORACLE_HOME}/lib/app_lib.so'
   AGENT 'sales.hq.example.com';
/
```

> **See Also:** *Oracle Database SQL Language Reference* for information about creating database links

## Related Topics

- "ALTER LIBRARY Statement" on page 14-6
- "DROP LIBRARY Statement" on page 14-93

# CREATE PACKAGE Statement

The CREATE PACKAGE statement creates or replaces the specification for a stored **package**, which is an encapsulated collection of related procedures, functions, and other program objects stored as a unit in the database. The **package specification** declares these objects. The **package body**, specified subsequently, defines these objects.

Topics:


- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

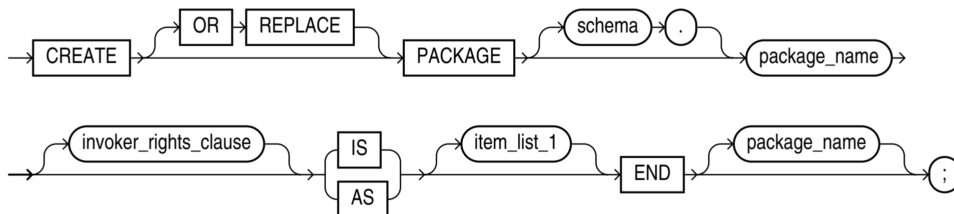

## Prerequisites

To create or replace a package in your own schema, you must have the CREATE PROCEDURE system privilege. To create or replace a package in another user's schema, you must have the CREATE ANY PROCEDURE system privilege.

To embed a CREATE PACKAGE statement inside an the database precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

## Syntax

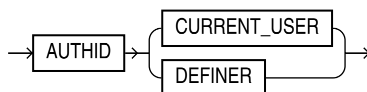
***create_package*** **::=**




See:


- *invoker_rights_clause ::=* on page 14-42
- *item_list_1 ::=* on page 13-8


***invoker_rights_clause*** **::=**




## Semantics

### OR REPLACE

Re-creates the package if it exists, and recompiles it.

Users who were granted privileges on the package before it was redefined can still access the package without being regranted the privileges.

If any function-based indexes depend on the package, then the database marks the indexes DISABLED.

#### *schema*

The name of the schema containing the package. The default is your own schema.

#### *item_list_1*

Declares package elements. If an item in item_list_1 is a pragma, it must one of these:

- "RESTRICT_REFERENCES Pragma" on page 13-109
- "SERIALLY_REUSABLE Pragma" on page 13-122

#### *package_name*

A package stored in the database. For naming conventions, see "Identifiers" on page 2-5.

#### *invoker_rights_clause*

Specifies the AUTHID property of the functions and procedures in the package, and of the explicit cursors declared in the package specification. For information about the AUTHID property, see "Invoker's Rights and Definer's Rights (AUTHID Property)" on page 8-19.

#### *item_list_1*

Declares a list of items. For syntax, see "Block" on page 13-8.

If an item in *item_list_1* is a pragma, it must one of these:

- "RESTRICT_REFERENCES Pragma" on page 13-109
- "SERIALLY_REUSABLE Pragma" on page 13-122

## Example

**Creating a Package: Example** This statement creates the specification of the emp_mgmt package.

```
CREATE OR REPLACE PACKAGE emp_mgmt AS
   FUNCTION hire (last_name VARCHAR2, job_id VARCHAR2,
      manager_id NUMBER, salary NUMBER,
      commission_pct NUMBER, department_id NUMBER)
      RETURN NUMBER;
   FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
      RETURN NUMBER;
   PROCEDURE remove_emp(employee_id NUMBER);
   PROCEDURE remove_dept(department_id NUMBER);
   PROCEDURE increase_sal(employee_id NUMBER, salary_incr NUMBER);
   PROCEDURE increase_comm(employee_id NUMBER, comm_incr NUMBER);
   no_comm EXCEPTION;
   no_sal EXCEPTION;
END emp_mgmt;
/
```

The specification for the emp_mgmt package declares these public program objects:

- The functions `hire` and `create_dept`
- The procedures `remove_emp`, `remove_dept`, `increase_sal`, and `increase_comm`
- The exceptions `no_comm` and `no_sal`

All of these objects are available to users who have access to the package. After creating the package, you can develop applications that call any of these public procedures or functions or raise any of the public exceptions of the package.

Before you can call this package's procedures and functions, you must define these procedures and functions in the package body. For an example of a `CREATE PACKAGE BODY` statement that creates the body of the `emp_mgmt` package, see "CREATE PACKAGE BODY Statement" on page 14-45.

## Related Topics

**In this chapter:**


- "ALTER PACKAGE Statement" on page 14-8
- "CREATE PACKAGE Statement" on page 14-42
- "CREATE PACKAGE BODY Statement" on page 14-45
- "DROP PACKAGE Statement" on page 14-94


**In other chapters:**


- "Function" on page 13-76
- "Procedure" on page 13-103
- Chapter 10, "PL/SQL Packages"

# CREATE PACKAGE BODY Statement

The CREATE PACKAGE BODY statement creates or replaces the body of a stored **package**, which is an encapsulated collection of related procedures, stored functions, and other program objects stored as a unit in the database. The **package body** defines these objects. The **package specification**, defined in an earlier CREATE PACKAGE statement, declares these objects.

Packages are an alternative to creating procedures and functions as standalone schema objects.

Topics:


- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

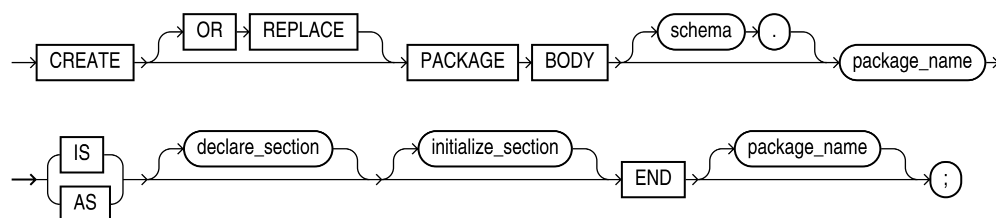

## Prerequisites

To create or replace a package in your own schema, you must have the CREATE PROCEDURE system privilege. To create or replace a package in another user's schema, you must have the CREATE ANY PROCEDURE system privilege. In both cases, the package body must be created in the same schema as the package.

To embed a CREATE PACKAGE BODY statement inside an the database precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

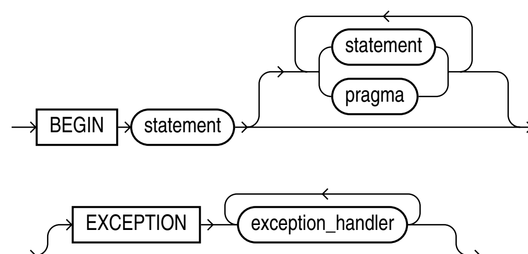
## Syntax

***create_package_body* ::=**




See *declare_section ::=* on page 13-8.

***initialize_section* ::=**



## Semantics

### OR REPLACE

Re-creates the package body if it exists, and recompiles it.

Users who were granted privileges on the package body before it was redefined can still access the package without being regranted the privileges.

### *schema*

The name of the schema containing the package. The default is your own schema.

### *package_name*

The name of the package to be created.

### *declare_section*

Declares package objects.

### *initialize_section*

Initializes package variables and does any other one-time setup steps.

### *body*

Defines package objects.

## Examples

**Creating a Package Body: Example** This statement creates the body of the `emp_mgmt` package created in "Creating a Package: Example" on page 14-43.

```
CREATE OR REPLACE PACKAGE BODY emp_mgmt AS
   tot_emps NUMBER;
   tot_depts NUMBER;
FUNCTION hire
   (last_name VARCHAR2, job_id VARCHAR2,
    manager_id NUMBER, salary NUMBER,
    commission_pct NUMBER, department_id NUMBER)
   RETURN NUMBER IS new_empno NUMBER;
BEGIN
   SELECT employees_seq.NEXTVAL
      INTO new_empno
      FROM DUAL;
   INSERT INTO employees
      VALUES (new_empno, 'First', 'Last','first.example@oracle.com',
              '(415)555-0100','18-JUN-02','IT_PROG',90000000,00,
              100,110);
      tot_emps := tot_emps + 1;
   RETURN(new_empno);
END;
FUNCTION create_dept(department_id NUMBER, location_id NUMBER)
   RETURN NUMBER IS
      new_deptno NUMBER;
   BEGIN
      SELECT departments_seq.NEXTVAL
         INTO new_deptno
         FROM dual;
      INSERT INTO departments
         VALUES (new_deptno, 'department name', 100, 1700);
```

```
      tot_depts := tot_depts + 1;
      RETURN(new_deptno);
   END;
PROCEDURE remove_emp (employee_id NUMBER) IS
   BEGIN
      DELETE FROM employees
      WHERE employees.employee_id = remove_emp.employee_id;
      tot_emps := tot_emps - 1;
   END;
PROCEDURE remove_dept(department_id NUMBER) IS
   BEGIN
      DELETE FROM departments
      WHERE departments.department_id = remove_dept.department_id;
      tot_depts := tot_depts - 1;
      SELECT COUNT(*) INTO tot_emps FROM employees;
   END;
PROCEDURE increase_sal(employee_id NUMBER, salary_incr NUMBER) IS
   curr_sal NUMBER;
   BEGIN
      SELECT salary INTO curr_sal FROM employees
      WHERE employees.employee_id = increase_sal.employee_id;
      IF curr_sal IS NULL
         THEN RAISE no_sal;
      ELSE
         UPDATE employees
         SET salary = salary + salary_incr
         WHERE employee_id = employee_id;
      END IF;
   END;
PROCEDURE increase_comm(employee_id NUMBER, comm_incr NUMBER) IS
   curr_comm NUMBER;
   BEGIN
      SELECT commission_pct
      INTO curr_comm
      FROM employees
      WHERE employees.employee_id = increase_comm.employee_id;
      IF curr_comm IS NULL
         THEN RAISE no_comm;
      ELSE
         UPDATE employees
         SET commission_pct = commission_pct + comm_incr;
      END IF;
   END;
END emp_mgmt;
/
```

The package body defines the public program objects declared in the package specification:

- The functions `hire` and `create_dept`
- The procedures `remove_emp`, `remove_dept`, `increase_sal`, and `increase_comm`

These objects are declared in the package specification, so they can be called by application programs, procedures, and functions outside the package. For example, if you have access to the package, you can create a procedure `increase_all_comms` separate from the `emp_mgmt` package that calls the `increase_comm` procedure.

These objects are defined in the package body, so you can change their definitions without causing the database to invalidate dependent schema objects. For example, if

you subsequently change the definition of `hire`, then the database need not recompile `increase_all_comms` before running it.

The package body in this example also declares private program objects, the variables `tot_emps` and `tot_depts`. These objects are declared in the package body rather than the package specification, so they are accessible to other objects in the package, but they are not accessible outside the package. For example, you cannot develop an application that explicitly changes the value of the variable `tot_depts`. However, the function `create_dept` is part of the package, so `create_dept` can change the value of `tot_depts`.

## Related Topics

**In this chapter:**

- "CREATE PACKAGE Statement" on page 14-42

**In other chapters:**

- "Function" on page 13-76
- "Procedure" on page 13-103
- Chapter 10, "PL/SQL Packages"

# CREATE PROCEDURE Statement

The CREATE PROCEDURE statement creates or replaces a standalone stored procedure or a call specification.

A **standalone stored procedure** is a procedure (a subprogram that performs a specific action) that is stored in the database.

---

**Note:** A standalone stored procedure that you create with the CREATE PROCEDURE statement differs from a procedure that you declare and define in a PL/SQL block or package. For information about the latter, see "Procedure" on page 13-103.

---

A **call specification** declares a Java method or a third-generation language (3GL) subprogram so that it can be called from PL/SQL. You can also use the SQL CALL statement to call such a method or subprogram. The call specification tells the database which Java method, or which named procedure in which shared library, to invoke when a call is made. It also tells the database what type conversions to make for the arguments and return value.

Topics:

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

## Prerequisites

To create or replace a standalone stored procedure in your own schema, you must have the CREATE PROCEDURE system privilege. To create or replace a standalone stored procedure in another user's schema, you must have the CREATE ANY PROCEDURE system privilege.

To invoke a call specification, you may need additional privileges, for example, the EXECUTE object privilege on the C library for a C call specification.

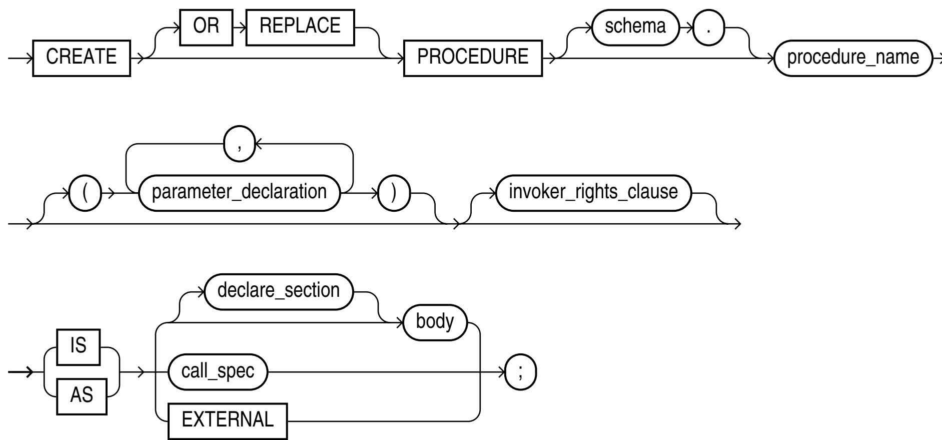
To embed a CREATE PROCEDURE statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

**See Also:** For more information about such prerequisites:

- *Oracle Database Advanced Application Developer's Guide*
- *Oracle Database Java Developer's Guide*

## Syntax

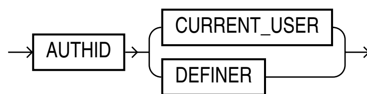
***create_procedure* ::=**



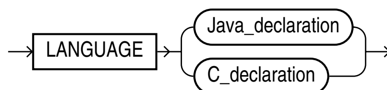

See:

- *body ::=* on page 13-11
- *declare_section ::=* on page 13-8
- *parameter_declaration ::=* on page 13-101

***invoker_rights_clause* ::=**

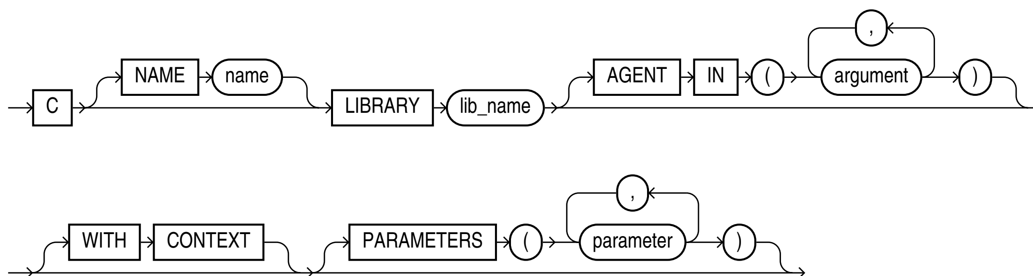



***call_spec* ::=**




***Java_declaration* ::=**

***C_declaration* ::=**



## Semantics

### OR REPLACE

Re-creates the procedure if it exists, and recompiles it.

Users who were granted privileges on the procedure before it was redefined can still access the procedure without being regranted the privileges.

If any function-based indexes depend on the procedure, then the database marks the indexes `DISABLED`.

### *schema*

The name of the schema containing the procedure. The default is your own schema.

### *procedure_name*

The name of the procedure to be created.

### *invoker_rights_clause*

Specifies the `AUTHID` property of the procedure. For information about the `AUTHID` property, see "Invoker's Rights and Definer's Rights (AUTHID Property)" on page 8-19.

### *body*

The required executable part of the procedure and, optionally, the exception-handling part of the procedure.

### *declare_section*

The optional declarative part of the procedure. Declarations are local to the procedure, can be referenced in *`body`*, and cease to exist when the procedure completes execution.

### *call_spec*

Maps a Java or C method name, parameter types, and return type to their SQL counterparts.

In the *`Java_declaration`*, *`string`* identifies the Java implementation of the method.

**See Also:**

- *Oracle Database Java Developer's Guide* for an explanation of the parameters and semantics of the *`Java_declaration`*
- *Oracle Database Advanced Application Developer's Guide* for information about calling external procedures

### EXTERNAL

The `EXTERNAL` clause is an alternative way of declaring a C method. In most cases, Oracle recommends that you use the `LANGUAGE C` syntax. However, `EXTERNAL` is required if a default argument is used as a parameter or if a parameter uses a PL/SQL data type that must be mapped (for example, Boolean). `EXTERNAL` causes the PL/SQL layer to be loaded so that the parameters can be properly evaluated.

## Examples

**Creating a Procedure: Example** This statement creates the procedure `remove_emp` in the schema `hr`.

```
CREATE PROCEDURE remove_emp (employee_id NUMBER) AS
   tot_emps NUMBER;
   BEGIN
      DELETE FROM employees
      WHERE employees.employee_id = remove_emp.employee_id;
   tot_emps := tot_emps - 1;
   END;
/
```

The `remove_emp` procedure removes a specified employee. When you call the procedure, you must specify the `employee_id` of the employee to be removed.

The procedure uses a `DELETE` statement to remove from the `employees` table the row of `employee_id`.

> **See Also:** "Creating a Package Body: Example" on page 14-46 to see how to incorporate this procedure into a package

In this example, external procedure `c_find_root` expects a pointer as a parameter. Procedure `find_root` passes the parameter by reference using the `BY REFERENCE` phrase.

```
CREATE PROCEDURE find_root
   ( x IN REAL )
   IS LANGUAGE C
      NAME c_find_root
      LIBRARY c_utils
      PARAMETERS ( x BY REFERENCE );
```

## Related Topics

### In this chapter:


- "ALTER PROCEDURE Statement" on page 14-11
- "CREATE FUNCTION Statement" on page 14-32
- "DROP PROCEDURE Statement" on page 14-96


### In other chapters:


- "Parameter" on page 13-101
- "Procedure" on page 13-103
- Chapter 8, "PL/SQL Subprograms"


> **See Also:**
>
> - *Oracle Database SQL Language Reference* for information about the `CALL` statement)
> - *Oracle Database Advanced Application Developer's Guide* for more information about call specifications

# CREATE TRIGGER Statement

The CREATE TRIGGER statement creates or replaces a **database trigger**, which is either of these:

- A stored PL/SQL block associated with a table, a schema, or the database
- An anonymous PL/SQL block or a call to a procedure implemented in PL/SQL or Java

The database automatically runs a trigger when specified conditions occur.

Topics:


- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

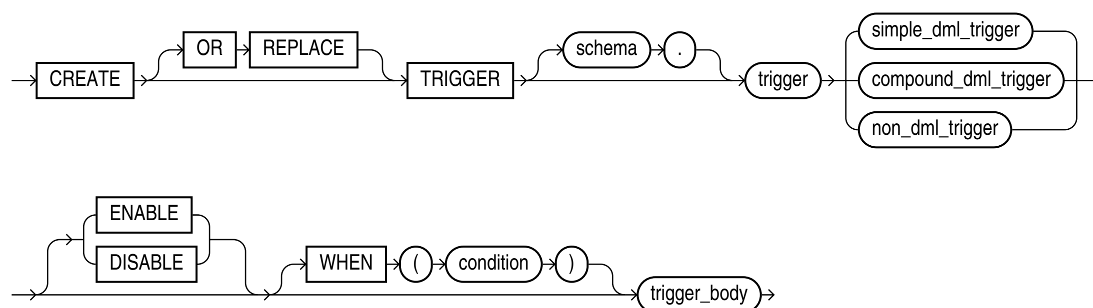

## Prerequisites

- To create a trigger in your own schema on a table in your own schema or on your own schema (SCHEMA), you must have the CREATE TRIGGER system privilege.
- To create a trigger in any schema on a table in any schema, or on another user's schema (*schema*.SCHEMA), you must have the CREATE ANY TRIGGER system privilege.
- In addition to the preceding privileges, to create a trigger on DATABASE, you must have the ADMINISTER DATABASE TRIGGER system privilege.
- In addition to the preceding privileges, to create a crossedition trigger, you must be enabled for editions. For information about enabling editions for a user, see *Oracle Database Advanced Application Developer's Guide*.

If the trigger issues SQL statements or calls procedures or functions, then the owner of the trigger must have the privileges necessary to perform these operations. These privileges must be granted directly to the owner rather than acquired through roles.

## Syntax

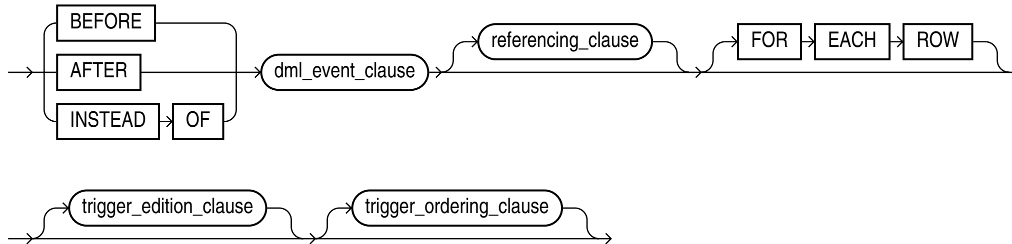
***create_trigger*** **::=**




See:

- *non_dml_trigger ::=* on page 14-54
- *trigger_body ::=* on page 14-55

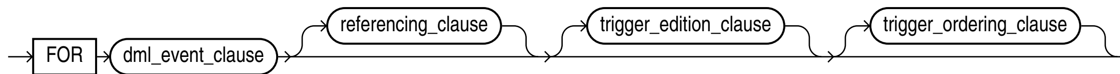
### *simple_dml_trigger* ::=




See:

- *dml_event_clause ::=* on page 14-55
- *referencing_clause ::=* on page 14-55
- *trigger_edition_clause ::=* on page 14-55
- *trigger_ordering_clause ::=* on page 14-55

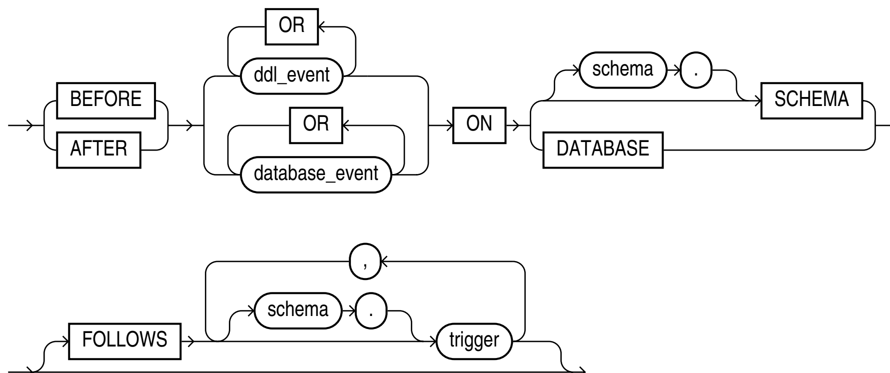
### *compound_dml_trigger* ::=



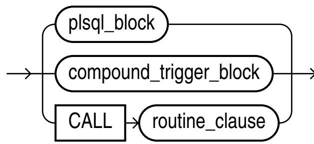

See:

- *dml_event_clause ::=* on page 14-55
- *referencing_clause ::=* on page 14-55
- *trigger_edition_clause ::=* on page 14-55
- *trigger_ordering_clause ::=* on page 14-55

### *non_dml_trigger* ::=



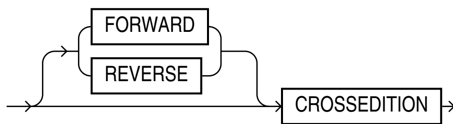
**trigger_body ::=**



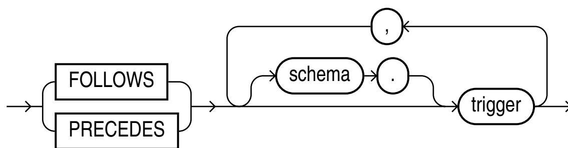

See:

- *compound_trigger_block ::=* on page 14-56
- *plsql_block ::=* on page 13-8
- *routine_clause* in *Oracle Database SQL Language Reference*

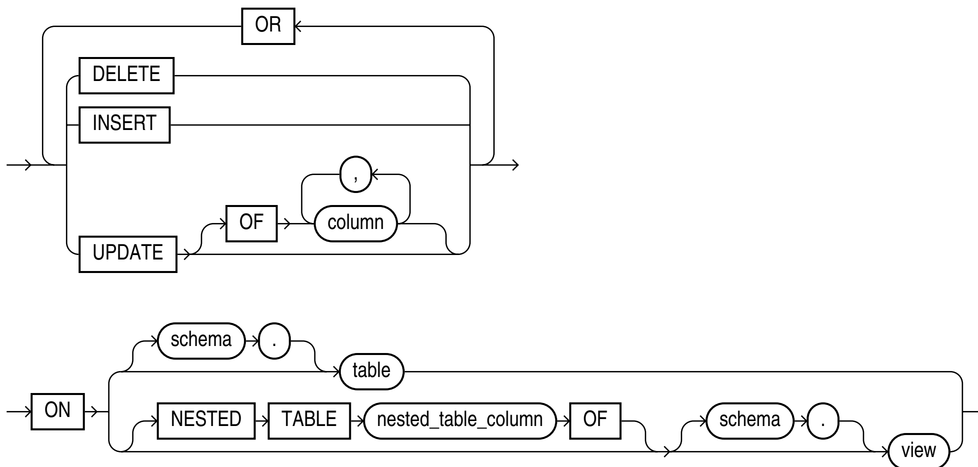
**trigger_edition_clause ::=**



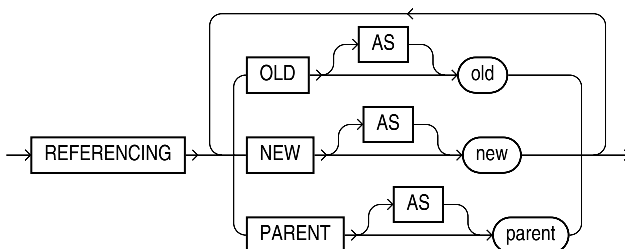

**trigger_ordering_clause ::=**



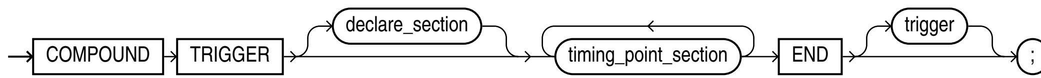

***dml_event_clause*** **::=**




***referencing_clause*** **::=**

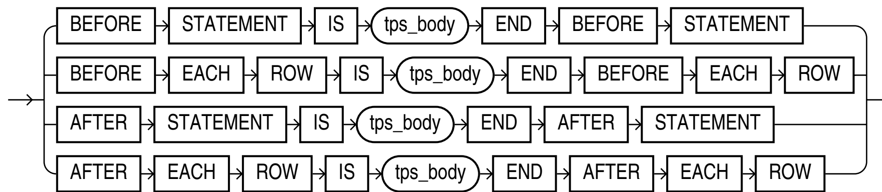


**compound_trigger_block ::=**

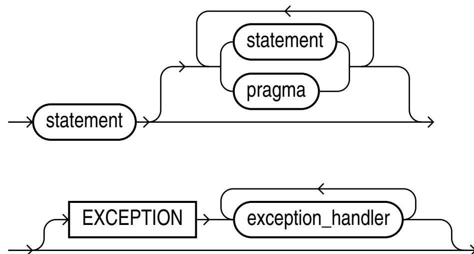



See *declare_section ::=* on page 13-8.

**timing_point_section ::=**




**tps_body ::=**




See *declare_section ::=* on page 13-8.

## Semantics

### OR REPLACE

Re-creates the trigger if it exists, and recompiles it.

### *schema*

The name of the schema containing the trigger. The default is your own schema.

### *trigger*

The name of the trigger to be created.

If a trigger produces compilation errors, then it is still created, but it fails on execution. A trigger that fails on execution effectively blocks all triggering DML statements until it is disabled, replaced by a version without compilation errors, or dropped. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

---

**Note:** If you create a trigger on a base table of a materialized view, then you must ensure that the trigger does not fire during a refresh of the materialized view. During refresh, the `DBMS_MVIEW` procedure `I_AM_A_REFRESH` returns `TRUE`.

---

#### CROSSEDITION

Creates the trigger as a crossedition trigger. A crossedition trigger must be defined on a table, not a view. Crossedition triggers are valid only with simple or compound DML triggers, not with DDL or database event triggers. A crossedition trigger is intended to fire when DML changes are made in a database while an online application that uses the database is being patched or upgraded with edition-based redefinition. The body of a crossedition trigger is designed to handle these DML changes so that they can be appropriately applied after the changes to the application code are completed.

The handling of DML changes during edition-based redefinition of an online application can entail multiple steps. Therefore, it is likely, though not required, that a crossedition trigger is also a **compound trigger**, which requires the `FOR` clause, rather than the `BEFORE`, `AFTER`, or `INSTEAD OF` keywords.

#### FORWARD

Creates the trigger as a forward crossedition trigger, which is the type of trigger described in **CROSSEDITION**. This is the default.

#### REVERSE

Creates the trigger as a reverse crossedition trigger, which is intended to fire when the application, after being patched or upgraded with edition-based redefinition, makes DML changes. Such a trigger can be used to propagate data to columns or tables used by the application before it was patched or upgraded.

> **See Also:** *Oracle Database Advanced Application Developer's Guide* for more information crossedition triggers

#### *simple_dml_trigger*

Creates the trigger as a single trigger on a DML event.

#### BEFORE

Causes the database to fire the trigger before running the triggering event. For row triggers, the trigger is fired before each affected row is changed.

##### Restrictions on BEFORE

- You cannot specify a `BEFORE` trigger on a view unless it is an editioning view.
- In a `BEFORE` statement trigger, or in `BEFORE` statement section of a compound trigger, you cannot specify either `:NEW` or `:OLD`. A `BEFORE` row trigger or a `BEFORE` row section of a compound trigger can read and write into the `:OLD` or `:NEW` fields.

#### AFTER

Causes the database to fire the trigger after running the triggering event. For row triggers, the trigger is fired after each affected row is changed.

##### Restrictions on AFTER

- You cannot specify a `AFTER` trigger on a view unless it is an editioning view.
- In an `AFTER` statement trigger or in `AFTER` statement section of a compound trigger, you cannot specify either `:NEW` or `:OLD`. An `AFTER` row trigger or `AFTER` row section of a compound trigger can only read but not write into the `:OLD` or `:NEW` fields.

---

**Note:** When you create a materialized view log for a table, the database implicitly creates an AFTER ROW trigger on the table. This trigger inserts a row into the materialized view log whenever an INSERT, UPDATE, or DELETE statement modifies data in the master table. You cannot control the order in which multiple row triggers fire. Therefore, do not write triggers intended to affect the content of the materialized view.

---

**See Also:**

- *Oracle Database SQL Language Reference* for more information about materialized view logs
- *Oracle Database Advanced Application Developer's Guide* for information about editioning views

### INSTEAD OF

Causes the database to fire the trigger instead of running the triggering event. You can achieve the same effect when you specify an INSTEAD OF ROW section in a compound trigger.

#### Restrictions on INSTEAD OF

- You can specify an INSTEAD OF trigger only on a noneditioning view (not an editioning view or table).

---

**Note:** The database fine-grained access control lets you define row-level security policies on views. These policies enforce specified rules in response to DML operations. If an INSTEAD OF trigger is also defined on the view, then the database does not enforce the row-level security policies, because the database fires the INSTEAD OF trigger instead of running the DML on the view.

---

- INSTEAD OF triggers are valid for DML events on noneditioning views. They are not valid for DDL or database events, and you cannot specify an INSTEAD OF trigger on a table.
- You can read both the :OLD and the :NEW value, but you cannot write either the :OLD or the :NEW value.
- If a view is inherently updatable and has INSTEAD OF triggers, then the triggers take preference. The database fires the triggers instead of performing DML on the view.
- If the view belongs to a hierarchy, then the trigger is not inherited by subviews.

**See Also:** "Creating an INSTEAD OF Trigger: Example" on page 14-67

### *dml_event_clause*

Specifies one of three DML statements that can cause the trigger to fire. The database fires the trigger in the existing user transaction.

You cannot specify the MERGE keyword in the *DML_event_clause*. If you want a trigger to fire in relation to a MERGE operation, then you must create triggers on the INSERT and UPDATE operations to which the MERGE operation decomposes.

**See Also:** "Creating a DML Trigger: Examples" on page 14-65

### DELETE

Causes the database to fire the trigger whenever a DELETE statement removes a row from the table or removes an element from a nested table.

### INSERT

Causes the database to fire the trigger whenever an INSERT statement adds a row to a table or adds an element to a nested table.

### UPDATE

Causes the database to fire the trigger whenever an UPDATE statement changes a value in a column specified after OF. If you omit OF, then the database fires the trigger whenever an UPDATE statement changes a value in any column of the table or nested table.

For an UPDATE trigger, you can specify ADT, varray, and REF columns after OF to indicate that the trigger must fire whenever an UPDATE statement changes a value in a column. However, you cannot change the values of these columns in the body of the trigger itself.

**Note:** Using OCI functions or the DBMS_LOB package to update LOB values or LOB attributes of object columns does not cause the database to fire triggers defined on the table containing the columns or the attributes.

#### Restrictions on UPDATE

- You cannot specify UPDATE OF for an INSTEAD OF trigger. The database fires INSTEAD OF triggers whenever an UPDATE changes a value in any column of the view.
- You cannot specify a nested table or LOB column in the UPDATE OF clause.

  Performing DML operations directly on nested table columns does not cause the database to fire triggers defined on the table containing the nested table column.

**See Also:** AS *subquery* clause of CREATE VIEW in *Oracle Database SQL Language Reference* for a list of constructs that prevent inserts, updates, or deletes on a view

### ON { *schema.table* | *schema.view* }

Specifies the database object on which the trigger is to be created:

- Table or view
- Object table or object view
- A column of nested-table type

If you omit *schema*, the database assumes the table is in your own schema.

**Restriction on *schema.table*** You cannot create a trigger on a table in the schema SYS.

#### NESTED TABLE *nested_table_column*

Specifies the *nested_table_column* of a view upon which the trigger is being defined. Such a trigger fires only if the DML operates on the elements of the nested table.

**Restriction on NESTED TABLE** You can specify NESTED TABLE only for INSTEAD OF triggers.

#### *referencing_clause*

Specifies correlation names. You can use correlation names in the trigger body and WHEN condition of a row trigger to refer specifically to old and new values of the current row. The default correlation names are OLD and NEW. If your row trigger is associated with a table named OLD or NEW, then use this clause to specify different correlation names to avoid confusion between the table name and the correlation name.

- If the trigger is defined on a nested table, then OLD and NEW refer to the row of the nested table, and PARENT refers to the current row of the parent table.
- If the trigger is defined on an object table or view, then OLD and NEW refer to object instances.

**Restriction on the *referencing_clause*** The *referencing_clause* is not valid with INSTEAD OF triggers on CREATE DDL events.

#### FOR EACH ROW

Creates the trigger as a row trigger. The database fires a row trigger once for each row that is affected by the triggering statement and meets the optional trigger constraint defined in the WHEN condition.

Except for INSTEAD OF triggers, if you omit this clause, then the trigger is a statement trigger. The database fires a statement trigger only once when the triggering statement is issued if the optional trigger constraint is met.

INSTEAD OF trigger statements are implicitly activated for each row.

**Restriction on FOR EACH ROW** This clause is valid only for simple DML triggers, not for compound DML triggers or for DDL or database event triggers.

#### *compound_dml_trigger*

Defines a compound trigger on a DML event. The body of a COMPOUND trigger can have up to four sections, so that you can specify a before statement, before row, after row, or after statement operation in one trigger.

The *dml_event_clause* and the *referencing_clause* have the same semantics for compound DML triggers as for simple DML triggers.

**Restriction on *compound_dml_trigger*** You cannot specify the FOR EACH ROW clause for a compound trigger.

> **See Also:** "Compound Trigger Restrictions" on page 9-17 for additional restrictions

#### *non_dml_trigger*

Defines a single trigger on a DDL or database event.

#### *ddl_event*

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 create BEFORE and AFTER triggers for these events. The database fires the trigger in the existing user transaction.

**Restriction on *ddl_event*** You cannot specify as a triggering event any DDL operation performed through a PL/SQL procedure.

> **See Also:** "Creating a DDL Trigger: Example" on page 14-66

The following *ddl_event* values are valid.

#### ALTER

Causes the database to fire the trigger whenever an ALTER statement modifies a database object in the data dictionary. An ALTER DATABASE statement does not fire the trigger.

#### ANALYZE

Causes the database to fire the trigger whenever the database collects or deletes statistics or validates the structure of a database object.

> **See Also:** *Oracle Database SQL Language Reference* for information about using the SQL statement ANALYZE to collect statistics

#### ASSOCIATE STATISTICS

Causes the database to fire the trigger whenever the database associates a statistics type with a database object.

#### AUDIT

Causes the database to fire the trigger whenever the database tracks the occurrence of a SQL statement or tracks operations on a schema object.

#### COMMENT

Causes the database to fire the trigger whenever a comment on a database object is added to the data dictionary.

#### CREATE

Causes the database to fire the trigger whenever a CREATE statement adds a database object to the data dictionary. The CREATE DATABASE or CREATE CONTROLFILE statement does not fire the trigger.

#### DISASSOCIATE STATISTICS

Causes the database to fire the trigger whenever the database disassociates a statistics type from a database object.

**DROP**

Causes the database to fire the trigger whenever a `DROP` statement removes a database object from the data dictionary.

**GRANT**

Causes the database to fire the trigger whenever a user grants system privileges or roles or object privileges to another user or to a role.

**NOAUDIT**

Causes the database to fire the trigger whenever whenever a `NOAUDIT` statement instructs the database to stop tracking a SQL statement or operations on a schema object.

**RENAME**

Causes the database to fire the trigger whenever a `RENAME` statement changes the name of a database object.

**REVOKE**

Causes the database to fire the trigger whenever a `REVOKE` statement removes system privileges or roles or object privileges from a user or role.

**TRUNCATE**

Causes the database to fire the trigger whenever a `TRUNCATE` statement removes the rows from a table or cluster and resets its storage characteristics.

**DDL**

Causes the database to fire the trigger whenever any of the preceding DDL statements is issued.

***database_event***

One or more particular states of the database that can cause the trigger to fire. You can create triggers for these events on `DATABASE` or `SCHEMA` unless otherwise noted. For each of these triggering events, the database opens an autonomous transaction scope, fires the trigger, and commits any separate transaction (regardless of any existing user transaction).

> **See Also:**
>
> - "Creating a Database Event Trigger: Example" on page 14-66
> - "Triggers that Publish Database Events" on page 9-50 for more information about responding to database events through triggers

Each database event is valid in either a `BEFORE` trigger or an `AFTER` trigger, but not both. These *`database_event`* values are valid:

**AFTER STARTUP**

Causes the database to fire the trigger whenever the database is opened. This event is valid only with `DATABASE`, not with `SCHEMA`.

**BEFORE SHUTDOWN**

Causes the database to fire the trigger whenever an instance of the database is shut down. This event is valid only with `DATABASE`, not with `SCHEMA`.

### AFTER DB_ROLE_CHANGE

In a Data Guard configuration, causes the database to fire the trigger whenever a role change occurs from standby to primary or from primary to standby. This event is valid only with `DATABASE`, not with `SCHEMA`.

### AFTER LOGON

Causes the database to fire the trigger whenever a client application logs onto the database.

### BEFORE LOGOFF

Causes the database to fire the trigger whenever a client application logs off the database.

### AFTER SERVERERROR

Causes the database to fire the trigger whenever a server error message is logged.

These errors do not cause a `SERVERERROR` trigger to fire:

- ORA-01403: no data found
- ORA-01422: exact fetch returns more than requested number of rows
- ORA-01423: error encountered while checking for extra rows in exact fetch
- ORA-01034: ORACLE not available
- ORA-04030: out of process memory when trying to allocate *string* bytes (*string*, *string*)

### AFTER SUSPEND

Causes the database to fire the trigger whenever a server error causes a transaction to be suspended.

> **See Also:** "Autonomous Transactions" on page 6-47 for information about autonomous transactions

### DATABASE

Defines the trigger on the entire database. The trigger fires whenever any database user initiates the triggering event.

### SCHEMA

Defines the trigger on the current schema. The trigger fires whenever any user connected as *`schema`* initiates the triggering event.

> **See Also:** "Creating a SCHEMA Trigger: Example" on page 14-67

### FOLLOWS | PRECEDES

Specifies the relative firing of triggers that have the same timing point. It is especially useful when creating crossedition triggers, which must fire in a specific order to achieve their purpose.

Use `FOLLOWS` to indicate that the trigger being created must fire after the specified triggers. You can specify `FOLLOWS` for a conventional trigger or for a forward crossedition trigger.

Use `PRECEDES` to indicate that the trigger being created must fire before the specified triggers. You can specify `PRECEDES` only for a reverse crossedition trigger.

The specified triggers must exist, and they must have been successfully compiled. They need not be enabled.

If you are creating a noncrossedition trigger, then the specified triggers must be all of the following:

- Noncrossedition triggers
- Defined on the same table as the trigger being created
- Visible in the same edition as the trigger being created

If you are creating a crossedition trigger, then the specified triggers must be all of the following:

- Crossedition triggers
- Defined on the same table or editioning view as the trigger being created, unless you specify `FOLLOWS` or `PRECEDES`.

  If you specify `FOLLOWS`, then the specified triggers must be forward crossedition triggers, and if you specify `PRECEDES`, then the specified triggers must be reverse crossedition triggers. However, the specified triggers need not be on the same table or editioning view as the trigger being created.
- Visible in the same edition as the trigger being created

In the following definitions, A, B, C, and D are either noncrossedition triggers or forward crossedition triggers:

- If B specifies A in its `FOLLOWS` clause, then B **directly follows** A.
- If C directly follows B, and B directly follows A, then C **indirectly follows** A.
- If D directly follows C, and C indirectly follows A, then D indirectly follows A.
- If B directly or indirectly follows A, then B **explicitly follows** A (that is, the firing order of B and A is explicitly specified by one or more `FOLLOWS` clauses).

In the following definitions, A, B, C, and D are reverse crossedition triggers:

- If A specifies B in its `PRECEDES` clause, then A **directly precedes** B.
- If A directly precedes B, and B directly precedes C, then A **indirectly precedes** C.
- If A directly precedes B, and B indirectly precedes D, then A indirectly precedes D.
- If A directly or indirectly precedes B, then A **explicitly precedes** B (that is, the firing order of A and B is explicitly specified by one or more `PRECEDES` clauses).

### ENABLE | DISABLE

Creates the trigger in an enabled or disabled state. Creating a trigger in a disabled state lets you ensure that the trigger compiles without errors before you put it into use.

Specify `DISABLE` to create the trigger in disabled form. You can subsequently issue an `ALTER TRIGGER ... ENABLE` or `ALTER TABLE ... ENABLE ALL TRIGGERS` statement to enable the trigger. If you omit this clause, then the trigger is enabled when it is created.

---

**Note:** This clause is especially useful if you are creating a crossedition trigger, which affects the online application being redefined if compilation errors arise.

---

**See Also:**

- "ALTER TRIGGER Statement" on page 14-14 for information about the ENABLE clause
- *Oracle Database SQL Language Reference* for information about using CREATE TABLE ... ENABLE ALL TRIGGERS

#### WHEN (*condition*)

Specifies the SQL condition that must be satisfied for the database to fire the trigger. This condition must contain correlation names and cannot contain a query.

The NEW and OLD keywords, when specified in the WHEN clause, are not considered bind variables, so are not preceded by a colon (:). However, you must precede NEW and OLD with a colon in all references other than the WHEN clause.

**See Also:**

- *Oracle Database SQL Language Reference* for the syntax description of *condition*
- "Calling a Procedure in a Trigger Body: Example" on page 14-66

**Restrictions on WHEN (*condition*)**

- If you specify this clause for a DML event trigger, then you must also specify FOR EACH ROW. The database evaluates this condition for each row affected by the triggering statement.
- You cannot specify trigger conditions for INSTEAD OF trigger statements.
- You can reference object columns or their attributes, or varray, nested table, or LOB columns. You cannot invoke PL/SQL functions or methods in the trigger condition.

#### *trigger_body*

The PL/SQL block, PL/SQL compound trigger block, or call procedure that the database runs to fire the trigger.

#### *compound_trigger_block*

Timing point sections can be in any order, but no timing point section can be repeated. The *declare_section* of a compound trigger block cannot include PRAGMA AUTONOMOUS_TRANSACTION.

## Examples

**Creating a DML Trigger: Examples** This example shows the basic syntax for a BEFORE statement trigger. Write such a trigger to place restrictions on DML statements issued on a table, for example, when such statements might be issued.

```
CREATE TRIGGER schema.trigger_name
    BEFORE
    DELETE OR INSERT OR UPDATE
    ON schema.table_name
       pl/sql_block
```

the database fires such a trigger whenever a DML statement affects the table. This trigger is a BEFORE statement trigger, so the database fires it once before running the triggering statement.

The next example shows a partial BEFORE row trigger. The PL/SQL block might specify, for example, that an employee's salary must fall in the established salary range for the employee's job:

```
CREATE TRIGGER hr.salary_check
      BEFORE INSERT OR UPDATE OF salary, job_id ON hr.employees
      FOR EACH ROW
         WHEN (new.job_id <> 'AD_VP')
      pl/sql_block
```

the database fires this trigger whenever one of these statements is issued:

- An INSERT statement that adds rows to the employees table
- An UPDATE statement that changes values of the salary or job_id columns of the employees table

salary_check is a BEFORE row trigger, so the database fires it before changing each row that is updated by the UPDATE statement or before adding each row that is inserted by the INSERT statement.

salary_check has a trigger condition that prevents it from checking the salary of the administrative vice president (AD_VP).

**Creating a DDL Trigger: Example** This example creates an AFTER statement trigger on any DDL statement CREATE. Such a trigger can be used to audit the creation of data dictionary objects in your schema.

```
CREATE TRIGGER audit_db_object AFTER CREATE
   ON SCHEMA
      pl/sql_block
```

**Calling a Procedure in a Trigger Body: Example** You can create the salary_check trigger described in the preceding example by calling a procedure instead of providing the trigger body in a PL/SQL block. Assume you have defined a procedure check_sal in the hr schema, which verifies that an employee's salary is in an appropriate range. Then you can create the trigger salary_check as follows:

```
CREATE TRIGGER salary_check
   BEFORE INSERT OR UPDATE OF salary, job_id ON employees
   FOR EACH ROW
   WHEN (new.job_id <> 'AD_VP')
   CALL check_sal(:new.job_id, :new.salary, :new.last_name)
```

The procedure check_sal can be implemented in PL/SQL, C, or Java. Also, you can specify :OLD values in the CALL clause instead of :NEW values.

**Creating a Database Event Trigger: Example** This example shows the basic syntax for a trigger to log all errors. The hypothetical PL/SQL block does some special processing for a particular error (invalid logon, error number 1017). This trigger is an AFTER statement trigger, so it is fired after an unsuccessful statement execution, such as unsuccessful logon.

```
CREATE TRIGGER log_errors AFTER SERVERERROR ON DATABASE
   BEGIN
      IF (IS_SERVERERROR (1017)) THEN
         <special processing of logon error>
      ELSE
         <log error number>
      END IF;
   END;
```

**Creating an INSTEAD OF Trigger: Example** In this example, an `oe.order_info` view is created to display information about customers and their orders:

```
CREATE VIEW order_info AS
   SELECT c.customer_id, c.cust_last_name, c.cust_first_name,
          o.order_id, o.order_date, o.order_status
   FROM customers c, orders o
   WHERE c.customer_id = o.customer_id;
```

Normally this view is not updatable, because the primary key of the `orders` table (`order_id`) is not unique in the result set of the join view. To make this view updatable, create an `INSTEAD OF` trigger on the view to process `INSERT` statements directed to the view.

```
CREATE OR REPLACE TRIGGER order_info_insert
   INSTEAD OF INSERT ON order_info
   DECLARE
     duplicate_info EXCEPTION;
     PRAGMA EXCEPTION_INIT (duplicate_info, -00001);
   BEGIN
     INSERT INTO customers
       (customer_id, cust_last_name, cust_first_name)
     VALUES (
     :new.customer_id,
     :new.cust_last_name,
     :new.cust_first_name);
   INSERT INTO orders (order_id, order_date, customer_id)
   VALUES (
     :new.order_id,
     :new.order_date,
     :new.customer_id);
   EXCEPTION
     WHEN duplicate_info THEN
       RAISE_APPLICATION_ERROR (
         num=> -20107,
         msg=> 'Duplicate customer or order ID');
   END order_info_insert;
/
```

You can now insert into both base tables through the view (if all `NOT NULL` columns receive values):

```
INSERT INTO order_info VALUES
   (999, 'Smith', 'John', 2500, '13-MAR-2001', 0);
```

For more information about `INSTEAD OF` triggers, see "INSTEAD OF Triggers" on page 9-8.

**Creating a SCHEMA Trigger: Example** This example creates a `BEFORE` statement trigger on the sample schema `hr`. When a user connected as `hr` attempts to drop a database object, the database fires the trigger before dropping the object:

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

## Related Topics

**In this chapter:**


- "ALTER TRIGGER Statement" on page 14-14
- "DROP TRIGGER Statement" on page 14-98


**In other chapters:**


- "Trigger Creation" on page 9-5


**See Also:** *Oracle Database Advanced Application Developer's Guide* for more information about crossedition triggers

# CREATE TYPE Statement

The CREATE TYPE statement creates or replaces the specification of one of these:

- Abstract Data Type (ADT) (including a SQLJ object type)
- Named varying array (varray) type
- Nested table type
- Incomplete object type

  An **incomplete type** is a type created by a forward type definition. It is called incomplete because it has a name but no attributes or methods. It can be referenced by other types, and so can be used to define types that refer to each other. However, you must fully specify the type before you can use it to create a table or an object column or a column of a nested table type.

The CREATE TYPE statement specifies the name of the type and its attributes, methods, and other properties. The CREATE TYPE BODY statement contains the code for the methods that implement the type.

**Notes:**

- If you create a type whose specification declares only attributes but no methods, then you need not specify a type body.
- If you create a SQLJ object type, then you cannot specify a type body. The implementation of the type is specified as a Java class.
- A standalone stored type that you create with the CREATE TYPE statement differs from a type that you define in a PL/SQL block or package. For information about the latter, see "Collection" on page 13-23.
- With the CREATE TYPE statement, you can create nested table and varray types, but not associative arrays. In a PL/SQL block or package, you can define all three collection types.

Topics:

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

## Prerequisites

To create a type in your own schema, you must have the CREATE TYPE system privilege. To create a type in another user's schema, you must have the CREATE ANY TYPE system privilege. You can acquire these privileges explicitly or be granted them through a role.

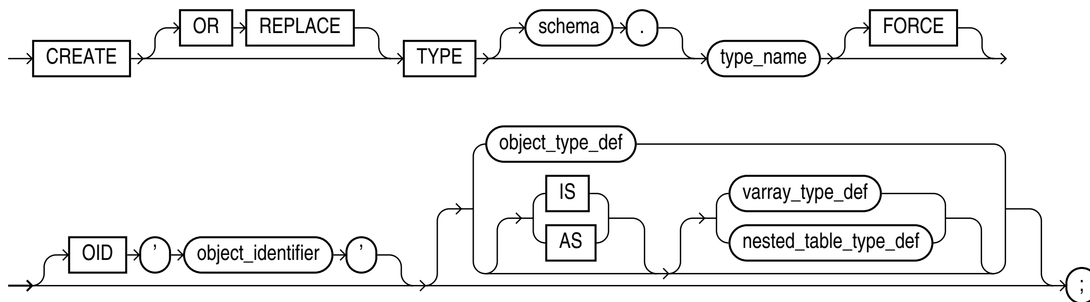
To create a subtype, you must have the UNDER ANY TYPE system privilege or the UNDER object privilege on the supertype.

The owner of the type must be explicitly granted the EXECUTE object privilege to access all other types referenced in the definition of the type, or the type owner must be granted the EXECUTE ANY TYPE system privilege. The owner cannot obtain these privileges through roles.

If the type owner intends to grant other users access to the type, then the owner must be granted the EXECUTE object privilege on the referenced types with the GRANT OPTION or the EXECUTE ANY TYPE system privilege with the ADMIN OPTION. Otherwise, the type owner has insufficient privileges to grant access on the type to other users.

## Syntax

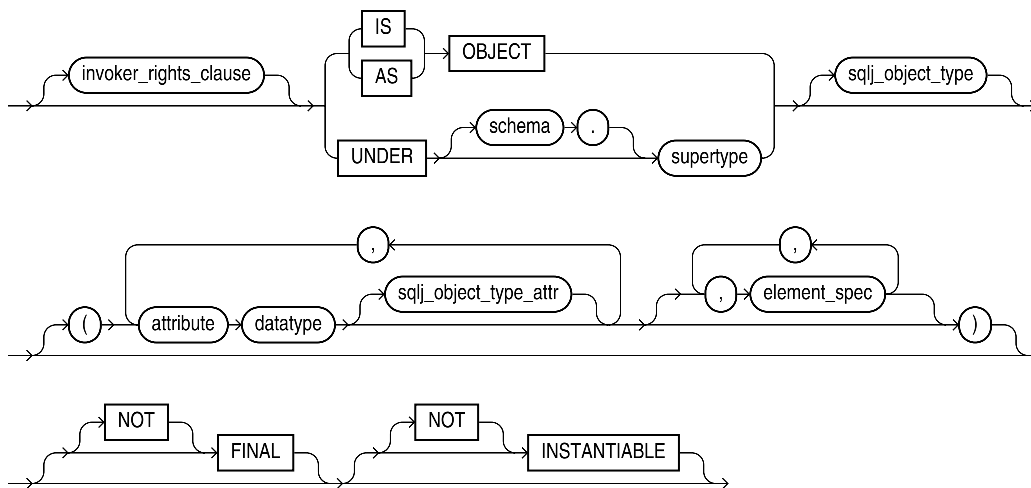
### *create_type* ::=




See:

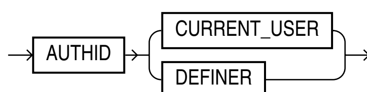
- *nested_table_type_def ::=* on page 13-24
- *varray_type_def ::=* on page 13-24

### *object_type_def* ::=

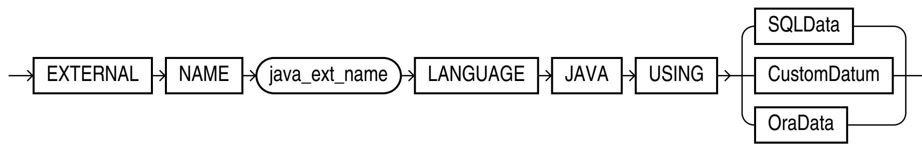



See *element_spec ::=* on page 14-71.

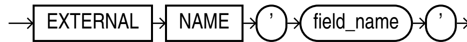
### *invoker_rights_clause* ::=



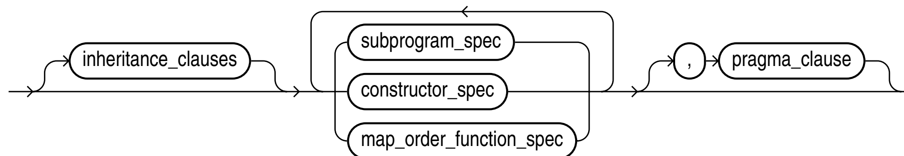
### *sqlj_object_type* ::=




### *sqlj_object_type_attr* ::=



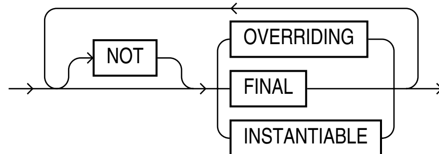

### *element_spec* ::=



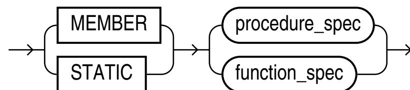

See:

- *constructor_spec ::=* on page 14-72
- *map_order_function_spec ::=* on page 14-72
- *pragma_clause ::=* on page 14-72

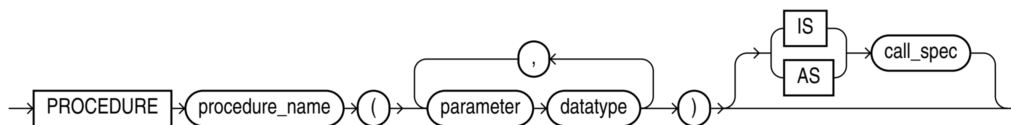
### *inheritance_clauses* ::=




### *subprogram_spec* ::=

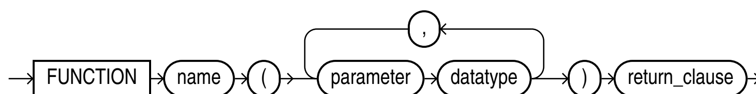



### *procedure_spec* ::=



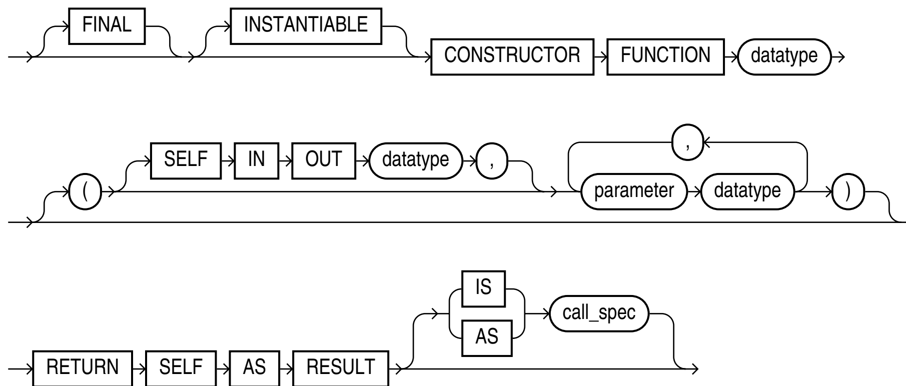

See *call_spec ::=* on page 14-73.

### *function_spec* ::=



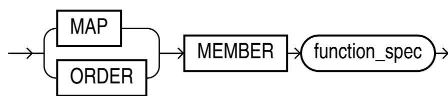

See *return_clause ::=* on page 14-72.

### *constructor_spec* ::=



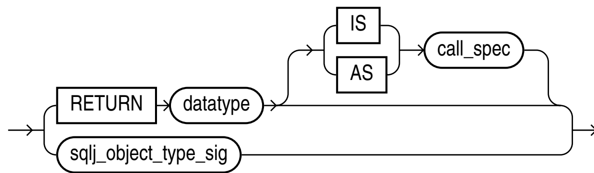

See *call_spec ::=* on page 14-73.

### *map_order_function_spec* ::=



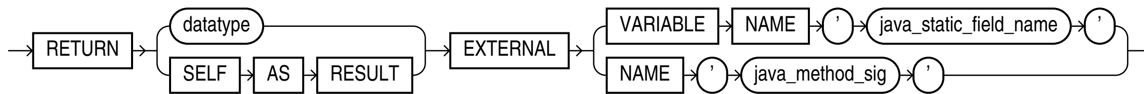

See *function_spec ::=* on page 14-71.

### *return_clause* ::=

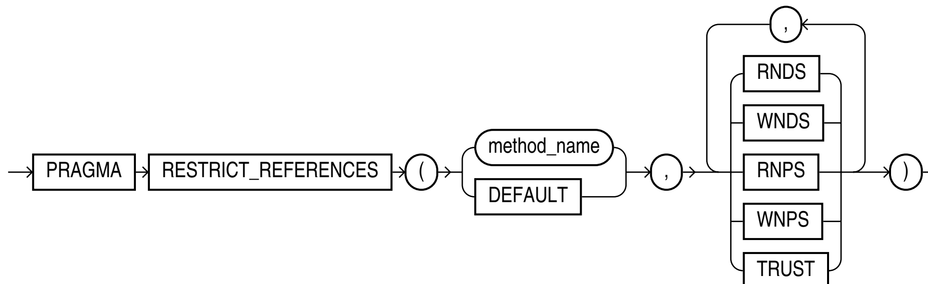



See (*call_spec ::=* on page 14-73.

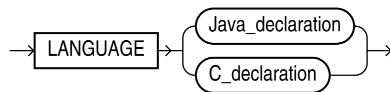
### *sqlj_object_type_sig* ::=




### *pragma_clause* ::=

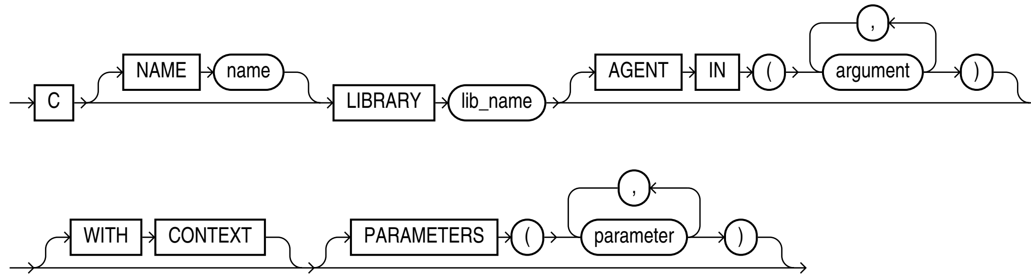


***call_spec*** **::=**




***Java_declaration*** **::=**

***C_declaration*** **::=**




## Semantics

### OR REPLACE

Re-creates the type if it exists, and recompiles it.

Users who were granted privileges on the type before it was redefined can still access the type without being regranted the privileges.

If any function-based indexes depend on the type, then the database marks the indexes `DISABLED`.

### *schema*

The name of the schema containing the type. The default is your own schema.

### *type_name*

The name of an ADT, a nested table type, or a varray type.

If creating the type results in compilation errors, then the database returns an error. You can see the associated compiler error messages with the SQL*Plus command `SHOW ERRORS`.

The database implicitly defines a constructor method for each user-defined type that you create. A **constructor** is a system-supplied procedure that is used in SQL statements or in PL/SQL code to construct an instance of the type value. The name of the constructor method is the name of the user-defined type. You can also create a user-defined constructor using the *`constructor_spec`* syntax.

The parameters of the ADT constructor method are the data attributes of the ADT. They occur in the same order as the attribute definition order for the ADT. The parameters of a nested table or varray constructor are the elements of the nested table or the varray.

#### FORCE

If *type_name* exists and has type dependents, but not table dependents, FORCE forces the statement to replace the type. (If *type_name* has table dependents, the statement fails with or without FORCE.)

> **Note:** If type t1 has type dependent t2, and type t2 has table dependents, then type t1 also has table dependents.

> **See Also:** *Oracle Database Object-Relational Developer's Guide*

#### *object_type_def*

Creates an ADT. The variables that form the data structure are called **attributes**. The member subprograms that define the behavior of the ADT are called **methods**. The keywords AS OBJECT are required when creating an ADT.

> **See Also:** "ADT Examples" on page 14-81

#### OID '*object_identifier*'

Establishes type equivalence of identical objects in multiple databases. See *Oracle Database Object-Relational Developer's Guide* for information about this clause.

#### *invoker_rights_clause*

Specifies the AUTHID property of the member functions and procedures of the ADT. For information about the AUTHID property, see "Invoker's Rights and Definer's Rights (AUTHID Property)" on page 8-19.

**Restrictions on *invoker_rights_clause*** This clause is subject to these restrictions:

- You can specify this clause only for an ADT, not for a nested table or varray type.
- You can specify this clause for clarity if you are creating a subtype. However, a subtype inherits the AUTHID property of its supertype, so you cannot specify a different value than was specified for the supertype.
- If the supertype was created with AUTHID DEFINER, then you must create the subtype in the same schema as the supertype.

#### AS OBJECT

Creates a schema-level ADT. Such ADTs are sometimes called **root** ADTs.

#### UNDER *supertype*

Creates a subtype of an existing type. The existing supertype must be an ADT. The subtype you create in this statement inherits the properties of its supertype. It must either override some of those properties or add properties to distinguish it from the supertype.

> **See Also:** "Subtype Example" on page 14-81 and "Type Hierarchy Example" on page 14-83

#### *sqlj_object_type*

Creates a **SQLJ object type**. With a SQLJ object type, you map a Java class to a SQL user-defined type. You can then define tables or columns of the SQLJ object type as you can with any other user-defined type.

You can map one Java class to multiple SQLJ object types. If there exists a subtype or supertype of a SQLJ object type, then it must also be a SQLJ object type. All types in the hierarchy must be SQLJ object types.

> **See Also:** *Oracle Database Object-Relational Developer's Guide* for more information about creating SQLJ object types

#### *java_ext_name*

The name of the Java class. If the class exists, then it must be public. The Java external name, including the schema, is validated.

Multiple SQLJ object types can be mapped to the same class. However:

- A subtype must be mapped to a class that is an immediate subclass of the class to which its supertype is mapped.
- Two subtypes of a common supertype cannot be mapped to the same class.

#### SQLData | CustomDatum | OraData

Specifies the mechanism for creating the Java instance of the type. `SQLData`, `CustomDatum`, and `OraData` are the interfaces that determine which mechanism to use.

> **See Also:** *Oracle Database JDBC Developer's Guide and Reference* for information about these three interfaces and "SQLJ Object Type Example" on page 14-81

#### *element_spec*

Specifies each attribute of the ADT.

#### *attribute*

The name of an ADT attribute. An ADT attribute is a data item with a name and a type specifier that forms the structure of the ADT. You must specify at least one attribute for each ADT. The name must be unique in the ADT, but can be used in other ADTs.

If you are creating a subtype, then the attribute name cannot be the same as any attribute or method name declared in the supertype chain.

#### *datatype*

The database built-in data type or user-defined type of the attribute.

##### Restrictions on *datatype*

- You cannot impose the `NOT NULL` constraint on an attribute.
- You cannot specify attributes of type `ROWID`, `LONG`, or `LONG RAW`.
- You cannot specify a data type of `UROWID` for an ADT.
- If you specify an object of type `REF`, then the target object must have an object identifier.
- If you are creating a collection type for use as a nested table or varray column of a table, then you cannot specify attributes of type `ANYTYPE`, `ANYDATA`, or `ANYDATASET`.

> **See Also:** Chapter 3, "PL/SQL Data Types," for a list of valid data types

### *sqlj_object_type_attr*

This clause is valid only if you have specified the *sqlj_object_type* clause to map a Java class to a SQLJ object type. Specify the external name of the Java field that corresponds to the attribute of the SQLJ object type. The Java *field_name* must exist in the class. You cannot map a Java *field_name* to multiple SQLJ object type attributes in the same type hierarchy.

This clause is optional when you create a SQLJ object type.

### *subprogram_spec*

Associates a procedure subprogram with the ADT.

### MEMBER

A function or procedure subprogram associated with the ADT that is referenced as an attribute. Typically, you invoke MEMBER methods in a selfish style, such as *object_expression.method*(). This class of method has an implicit first argument referenced as SELF in the method body, which represents the object on which the method was invoked.

**Restriction on MEMBER** You cannot specify a MEMBER method if you are mapping a Java class to a SQLJ object type.

> **See Also:** "Creating a Member Method: Example" on page 14-84

### STATIC

A function or procedure subprogram associated with the ADT. Unlike MEMBER methods, STATIC methods do not have any implicit parameters. You cannot reference SELF in their body. They are typically invoked as *type_name.method*().

#### Restrictions on STATIC

- You cannot map a MEMBER method in a Java class to a STATIC method in a SQLJ object type.
- For both MEMBER and STATIC methods, you must specify a corresponding method body in the type body for each procedure or function specification.

> **See Also:** "Creating a Static Method: Example" on page 14-84

### [NOT] FINAL, [NOT] INSTANTIABLE

At the schema level of the syntax, these clauses specify the inheritance attributes of the type.

Use the [NOT] FINAL clause to indicate whether any further subtypes can be created for this type:

- Specify FINAL if no further subtypes can be created for this type. This is the default.
- Specify NOT FINAL if further subtypes can be created under this type.

Use the [NOT] INSTANTIABLE clause to indicate whether any object instances of this type can be constructed:

- Specify INSTANTIABLE if object instances of this type can be constructed. This is the default.
- Specify NOT INSTANTIABLE if no default or user-defined constructor exists for this ADT. You must specify these keywords for any type with noninstantiable

methods and for any type that has no attributes, either inherited or specified in this statement.

### *inheritance_clauses*

Specify the relationship between supertypes and subtypes.

### OVERRIDING

Specifies that this method overrides a MEMBER method defined in the supertype. This keyword is required if the method redefines a supertype method. NOT OVERRIDING is the default.

**Restriction on OVERRIDING** The OVERRIDING clause is not valid for a STATIC method or for a SQLJ object type.

### FINAL

Specifies that this method cannot be overridden by any subtype of this type. The default is NOT FINAL.

### NOT INSTANTIABLE

Specifies that the type does not provide an implementation for this method. By default all methods are INSTANTIABLE.

**Restriction on NOT INSTANTIABLE** If you specify NOT INSTANTIABLE, then you cannot specify FINAL or STATIC.

> **See Also:** *constructor_spec* on page 14-79

### *procedure_spec* or *function_spec*

Specifies the parameters and data types of the procedure or function. If this subprogram does not include the declaration of the procedure or function, then you must issue a corresponding CREATE TYPE BODY statement.

**Restriction on *procedure_spec* or *function_spec*** If you are creating a subtype, then the name of the procedure or function cannot be the same as the name of any attribute, whether inherited or not, declared in the supertype chain.

### *return_clause*

The first form of the *return_clause* is valid only for a function. The syntax shown is an abbreviated form.

> **See Also:**
>
> - "Collection Method Call" on page 13-27 for information about method invocation and methods
> - "CREATE PROCEDURE Statement" on page 14-49 and "CREATE FUNCTION Statement" on page 14-32 for the full syntax with all possible clauses

### *sqlj_object_type_sig*

Use this form of the *return_clause* if you intend to create SQLJ object type functions or procedures.

- If you are mapping a Java class to a SQLJ object type and you specify EXTERNAL NAME, then the value of the Java method returned must be compatible with the

SQL returned value, and the Java method must be public. Also, the method signature (method name plus parameter types) must be unique in the type hierarchy.

- If you specify `EXTERNAL VARIABLE NAME`, then the type of the Java static field must be compatible with the return type.

#### *call_spec*

The call specification that maps a Java or C method name, parameter types, and return type to their SQL counterparts. If all the member methods in the type have been defined in this clause, then you need not issue a corresponding `CREATE TYPE BODY` statement.

The *`Java_declaration`* string identifies the Java implementation of the method.

> **See Also:**
>
> - *Oracle Database Java Developer's Guide* for an explanation of the parameters and semantics of the *`Java_declaration`*
> - *Oracle Database Advanced Application Developer's Guide* for information about calling external procedures

#### *pragma_clause*

Specifies a compiler directive. The `PRAGMA RESTRICT_REFERENCES` compiler directive denies member functions read/write access to database tables, packaged variables, or both, and thereby helps to avoid side effects.

> **Note:** Oracle recommends that you avoid using this clause unless you must do so for backward compatibility of your applications. This clause is deprecated, because the database now runs purity checks at run time.

#### *method*

The name of the `MEMBER` function or procedure to which the pragma is being applied.

#### DEFAULT

Causes the database to apply the pragma to all methods in the type for which a pragma has not been explicitly specified.

#### WNDS

Enforces the constraint **writes no database state**, which means that the method does not modify database tables.

#### WNPS

Enforces the constraint **writes no package state**, which means that the method does not modify packaged variables.

#### RNDS

Enforces the constraint **reads no database state**, which means that the method does not query database tables.

### RNPS

Enforces the constraint **reads no package state**, which means that the method does not reference package variables.

### TRUST

Specifies that the restrictions listed in the pragma are not to be enforced but are trusted to be true.

> **See Also:** "RESTRICT_REFERENCES Pragma" on page 13-109 for more information about this pragma

### *constructor_spec*

Creates a user-defined constructor, which is a function that returns an initialized instance of an ADT. You can declare multiple constructors for a single ADT, if the parameters of each constructor differ in number, order, or data type.

- User-defined constructor functions are always `FINAL` and `INSTANTIABLE`, so these keywords are optional.
- The parameter-passing mode of user-defined constructors is always `SELF IN OUT`. Therefore you need not specify this clause unless you want to do so for clarity.
- `RETURN SELF AS RESULT` specifies that the run-time type of the value returned by the constructor is run-time type of the `SELF` argument.

> **See Also:** *Oracle Database Object-Relational Developer's Guide* for more information about and examples of user-defined constructors and "Constructor Example" on page 14-83

### *map_order_function_spec*

You can define either one `MAP` method or one `ORDER` method in a type specification, regardless of how many `MEMBER` or `STATIC` methods you define. If you declare either method, then you can compare object instances in SQL.

You cannot define either `MAP` or `ORDER` methods for subtypes. However, a subtype can override a `MAP` method if the supertype defines a nonfinal `MAP` method. A subtype cannot override an `ORDER` method at all.

You can specify either `MAP` or `ORDER` when mapping a Java class to a SQL type. However, the `MAP` or `ORDER` methods must map to `MEMBER` functions in the Java class.

If neither a `MAP` nor an `ORDER` method is specified, then only comparisons for equality or inequality can be performed. Therefore object instances cannot be ordered. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal. No comparison method must be specified to determine the equality of two ADTs.

Use `MAP` if you are performing extensive sorting or hash join operations on object instances. `MAP` is applied once to map the objects to scalar values, and then the database uses the scalars during sorting and merging. A `MAP` method is more efficient than an `ORDER` method, which must invoke the method for each object comparison. You must use a `MAP` method for hash joins. You cannot use an `ORDER` method because the hash mechanism hashes on the object value.

> **See Also:** *Oracle Database Object-Relational Developer's Guide* for more information about object value comparisons

### MAP MEMBER

Specifies a MAP member function that returns the relative position of a given instance in the ordering of all instances of the object. A MAP method is called implicitly and induces an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

If the argument to the MAP method is null, then the MAP method returns null and the method is not invoked.

An object specification can contain only one MAP method, which must be a function. The result type must be a predefined SQL scalar type, and the MAP method can have no arguments other than the implicit SELF argument.

> **Note:** If *type_name* will be referenced in queries containing sorts (through an ORDER BY, GROUP BY, DISTINCT, or UNION clause) or containing joins, and you want those queries to be parallelized, then you must specify a MAP member function.

A subtype cannot define a new MAP method, but it can override an inherited MAP method.

### ORDER MEMBER

Specifies an ORDER member function that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative, zero, or positive integer. The negative, positive, or zero indicates that the implicit SELF argument is less than, equal to, or greater than the explicit argument.

If either argument to the ORDER method is null, then the ORDER method returns null and the method is not invoked.

When instances of the same ADT definition are compared in an ORDER BY clause, the ORDER method *map_order_function_spec* is invoked.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

A subtype can neither define nor override an ORDER method.

### *varray_type_def*

Creates the type as an ordered set of elements, each of which has the same data type.

**Restrictions on *varray_type_def*** You can create a VARRAY type of XMLType or of a LOB type for procedural purposes, for example, in PL/SQL or in view queries. However, database storage of such a varray is not supported, so you cannot create an object table or an database column of such a varray type.

**See Also:** "Varray Type Example" on page 14-83

### *nested_table_type_def*

Creates a named nested table of type *datatype*.

**See Also:**


- "Nested Table Type Example" on page 14-83
- "Nested Table Type Containing a Varray" on page 14-83

## Examples

**ADT Examples** This example shows how the sample type `customer_typ` was created for the sample Order Entry (`oe`) schema. A hypothetical name is given to the table so that you can duplicate this example in your test database:

```
CREATE TYPE customer_typ_demo AS OBJECT
    ( customer_id        NUMBER(6)
    , cust_first_name    VARCHAR2(20)
    , cust_last_name     VARCHAR2(20)
    , cust_address       CUST_ADDRESS_TYP
    , phone_numbers      PHONE_LIST_TYP
    , nls_language       VARCHAR2(3)
    , nls_territory      VARCHAR2(30)
    , credit_limit       NUMBER(9,2)
    , cust_email         VARCHAR2(30)
    , cust_orders        ORDER_LIST_TYP
    ) ;
/
```

In this example, the `data_typ1` ADT is created with one member function `prod`, which is implemented in the `CREATE TYPE BODY` statement:

```
CREATE TYPE data_typ1 AS OBJECT
   ( year NUMBER,
     MEMBER FUNCTION prod(invent NUMBER) RETURN NUMBER
   );
/

CREATE TYPE BODY data_typ1 IS
      MEMBER FUNCTION prod (invent NUMBER) RETURN NUMBER IS
         BEGIN
             RETURN (year + invent);
         END;
      END;
/
```

**Subtype Example** This statement shows how the subtype `corporate_customer_typ` in the sample `oe` schema was created. It is based on the `customer_typ` supertype created in the preceding example and adds the `account_mgr_id` attribute. A hypothetical name is given to the table so that you can duplicate this example in your test database:

```
CREATE TYPE corporate_customer_typ_demo UNDER customer_typ
    ( account_mgr_id     NUMBER(6)
    );
```

**SQLJ Object Type Example** These examples create a SQLJ object type and subtype. The `address_t` type maps to the Java class `Examples.Address`. The subtype `long_address_t` maps to the Java class `Examples.LongAddress`. The examples specify SQLData as the mechanism used to create the Java instance of these types. Each of the functions in these type specifications has a corresponding implementation in the Java class.

> **See Also:** *Oracle Database Object-Relational Developer's Guide* for the Java implementation of the functions in these type specifications

```
CREATE TYPE address_t AS OBJECT
```

```
  EXTERNAL NAME 'Examples.Address' LANGUAGE JAVA
  USING SQLData(
    street_attr varchar(250) EXTERNAL NAME 'street',
    city_attr varchar(50) EXTERNAL NAME 'city',
    state varchar(50) EXTERNAL NAME 'state',
    zip_code_attr number EXTERNAL NAME 'zipCode',
    STATIC FUNCTION recom_width RETURN NUMBER
      EXTERNAL VARIABLE NAME 'recommendedWidth',
    STATIC FUNCTION create_address RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION construct RETURN address_t
      EXTERNAL NAME 'create() return Examples.Address',
    STATIC FUNCTION create_address (street VARCHAR, city VARCHAR,
        state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME 'create (java.lang.String, java.lang.String, java.lang.String,
int) return Examples.Address',
    STATIC FUNCTION construct (street VARCHAR, city VARCHAR,
        state VARCHAR, zip NUMBER) RETURN address_t
      EXTERNAL NAME
        'create (java.lang.String, java.lang.String, java.lang.String, int) return
Examples.Address',
    MEMBER FUNCTION to_string RETURN VARCHAR
      EXTERNAL NAME 'tojava.lang.String() return java.lang.String',
    MEMBER FUNCTION strip RETURN SELF AS RESULT
      EXTERNAL NAME 'removeLeadingBlanks () return Examples.Address'
  ) NOT FINAL;
/

CREATE OR REPLACE TYPE long_address_t
UNDER address_t
EXTERNAL NAME 'Examples.LongAddress' LANGUAGE JAVA
USING SQLData(
    street2_attr VARCHAR(250) EXTERNAL NAME 'street2',
    country_attr VARCHAR (200) EXTERNAL NAME 'country',
    address_code_attr VARCHAR (50) EXTERNAL NAME 'addrCode',
    STATIC FUNCTION create_address RETURN long_address_t
      EXTERNAL NAME 'create() return Examples.LongAddress',
    STATIC FUNCTION  construct (street VARCHAR, city VARCHAR,
        state VARCHAR, country VARCHAR, addrs_cd VARCHAR)
      RETURN long_address_t
      EXTERNAL NAME
        'create(java.lang.String, java.lang.String,
        java.lang.String, java.lang.String, java.lang.String)
          return Examples.LongAddress',
    STATIC FUNCTION construct RETURN long_address_t
      EXTERNAL NAME 'Examples.LongAddress()
        return Examples.LongAddress',
    STATIC FUNCTION create_longaddress (
      street VARCHAR, city VARCHAR, state VARCHAR, country VARCHAR,
      addrs_cd VARCHAR) return long_address_t
      EXTERNAL NAME
        'Examples.LongAddress (java.lang.String, java.lang.String,
         java.lang.String, java.lang.String, java.lang.String)
           return Examples.LongAddress',
    MEMBER FUNCTION get_country RETURN VARCHAR
      EXTERNAL NAME 'country_with_code () return java.lang.String'
  );
/
```

**Type Hierarchy Example** These statements create a type hierarchy. Type employee_t inherits the name and ssn attributes from type person_t and in addition has department_id and salary attributes. Type part_time_emp_t inherits all of the attributes from employee_t and, through employee_t, those of person_t and in addition has a num_hrs attribute. Type part_time_emp_t is final by default, so no further subtypes can be created under it.

```
CREATE TYPE person_t AS OBJECT (name VARCHAR2(100), ssn NUMBER)
   NOT FINAL;
/

CREATE TYPE employee_t UNDER person_t
   (department_id NUMBER, salary NUMBER) NOT FINAL;
/

CREATE TYPE part_time_emp_t UNDER employee_t (num_hrs NUMBER);
/
```

You can use type hierarchies to create substitutable tables and tables with substitutable columns.

**Varray Type Example** This statement shows how the phone_list_typ varray type with five elements in the sample oe schema was created. A hypothetical name is given to the table so that you can duplicate this example in your test database:

```
CREATE TYPE phone_list_typ_demo AS VARRAY(5) OF VARCHAR2(25);
```

**Nested Table Type Example** This example from the sample schema pm creates the table type textdoc_tab of type textdoc_typ:

```
CREATE TYPE textdoc_typ AS OBJECT
    ( document_typ      VARCHAR2(32)
    , formatted_doc     BLOB
    ) ;

CREATE TYPE textdoc_tab AS TABLE OF textdoc_typ;
```

**Nested Table Type Containing a Varray** This example of multilevel collections is a variation of the sample table oe.customers. In this example, the cust_address object column becomes a nested table column with the phone_list_typ varray column embedded in it. The phone_list_typ type was created in "Varray Type Example" on page 14-83.

```
CREATE TYPE cust_address_typ2 AS OBJECT
       ( street_address     VARCHAR2(40)
       , postal_code        VARCHAR2(10)
       , city               VARCHAR2(30)
       , state_province     VARCHAR2(10)
       , country_id         CHAR(2)
       , phone              phone_list_typ_demo
       );

CREATE TYPE cust_nt_address_typ
   AS TABLE OF cust_address_typ2;
```

**Constructor Example** This example invokes the system-defined constructor to construct the demo_typ object and insert it into the demo_tab table:

```
CREATE TYPE demo_typ1 AS OBJECT (a1 NUMBER, a2 NUMBER);
```

```
CREATE TABLE demo_tab1 (b1 NUMBER, b2 demo_typ1);

INSERT INTO demo_tab1 VALUES (1, demo_typ1(2,3));
```

> **See Also:** *Oracle Database Object-Relational Developer's Guide* for more information about constructors

**Creating a Member Method: Example** This example invokes method constructor `col.get_square`. First the type is created:

```
CREATE TYPE demo_typ2 AS OBJECT (a1 NUMBER,
   MEMBER FUNCTION get_square RETURN NUMBER);
```

Next a table is created with an ADT column and some data is inserted into the table:

```
CREATE TABLE demo_tab2(col demo_typ2);

INSERT INTO demo_tab2 VALUES (demo_typ2(2));
```

The type body is created to define the member function, and the member method is invoked:

```
CREATE TYPE BODY demo_typ2 IS
   MEMBER FUNCTION get_square
   RETURN NUMBER
   IS x NUMBER;
   BEGIN
      SELECT c.col.a1*c.col.a1 INTO x
      FROM demo_tab2 c;
      RETURN (x);
   END;
END;
/

SELECT t.col.get_square() FROM demo_tab2 t;

T.COL.GET_SQUARE()
------------------
                 4
```

Unlike function invocations, method invocations require parentheses, even when the methods do not have additional arguments.

**Creating a Static Method: Example** This example changes the definition of the `employee_t` type to associate it with the `construct_emp` function. The example first creates an ADT `department_t` and then an ADT `employee_t` containing an attribute of type `department_t`:

```
CREATE OR REPLACE TYPE department_t AS OBJECT (
   deptno number(10),
   dname CHAR(30));

CREATE OR REPLACE TYPE employee_t AS OBJECT(
   empid RAW(16),
   ename CHAR(31),
   dept REF department_t,
      STATIC function construct_emp
      (name VARCHAR2, dept REF department_t)
      RETURN employee_t
);
```

This statement requires this type body statement.

```
CREATE OR REPLACE TYPE BODY employee_t IS
   STATIC FUNCTION construct_emp
   (name varchar2, dept REF department_t)
   RETURN employee_t IS
      BEGIN
         return employee_t(SYS_GUID(),name,dept);
      END;
END;
```

Next create an object table and insert into the table:

```
CREATE TABLE emptab OF employee_t;
INSERT INTO emptab
   VALUES (employee_t.construct_emp('John Smith', NULL));
```

## Related Topics

**In this chapter:**

- "ALTER TYPE Statement" on page 14-17
- "CREATE TYPE BODY Statement" on page 14-86
- "DROP TYPE Statement" on page 14-99

**In other chapters:**

- "Abstract Data Types" on page 1-8
- "Conditional Compilation Directive Restrictions" on page 2-53
- "Collection Type Definitions" on page 5-6
- "Collection" on page 13-23

> **See Also:** *Oracle Database Object-Relational Developer's Guide* for more information about objects, incomplete types, varrays, and nested tables

# CREATE TYPE BODY Statement

The CREATE TYPE BODY defines or implements the member methods defined in the type specification that was created with the "CREATE TYPE Statement" on page 14-69.

For each method specified in a type specification for which you did not specify the *call_spec*, you must specify a corresponding method body in the type body.

> **Note:** If you create a SQLJ object type, then specify it as a Java class.

Topics:

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

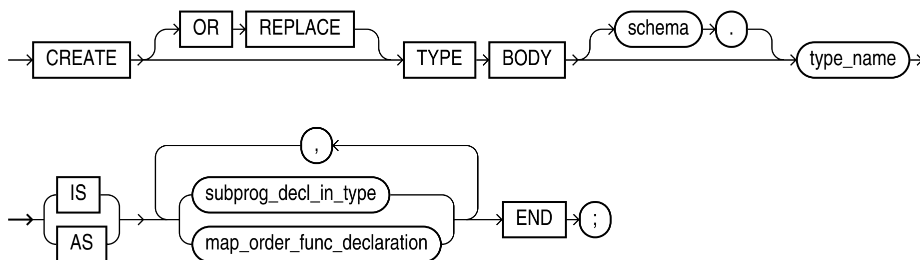
## Prerequisites

Every member declaration in the CREATE TYPE specification for an ADT must have a corresponding construct in the CREATE TYPE or CREATE TYPE BODY statement.

To create or replace a type body in your own schema, you must have the CREATE TYPE or the CREATE ANY TYPE system privilege. To create a type in another user's schema, you must have the CREATE ANY TYPE system privilege. To replace a type in another user's schema, you must have the DROP ANY TYPE system privilege.

## Syntax

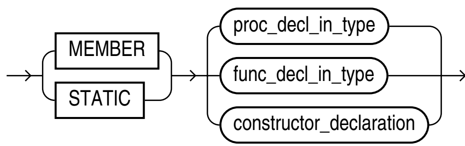
***create_type_body* ::=**



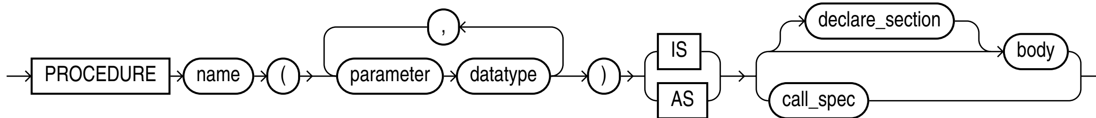

See:

- *map_order_func_declaration ::=* on page 14-88
- *subprog_decl_in_type ::=* on page 14-87

### *subprog_decl_in_type* ::=



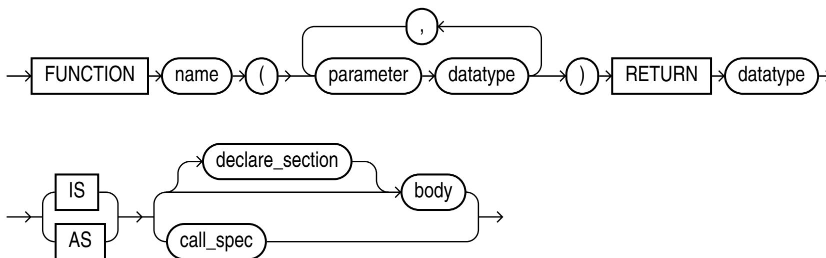

### *proc_decl_in_type* ::=




See:

- *body ::=* on page 13-11
- *call_spec ::=* on page 14-88
- *declare_section ::=* on page 13-8

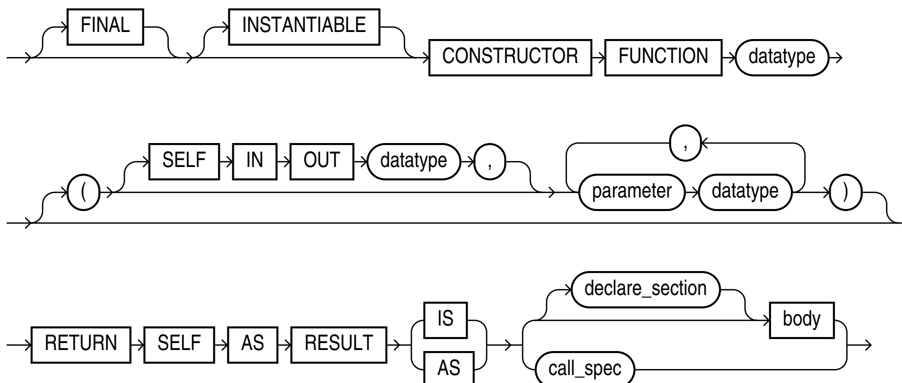
### *func_decl_in_type* ::=



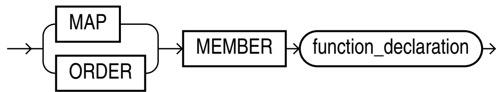

See:

- *body ::=* on page 13-11
- *call_spec ::=* on page 14-88
- *declare_section ::=* on page 13-8

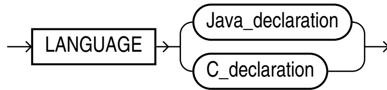
### *constructor_declaration* ::=



***map_order_func_declaration* ::=**

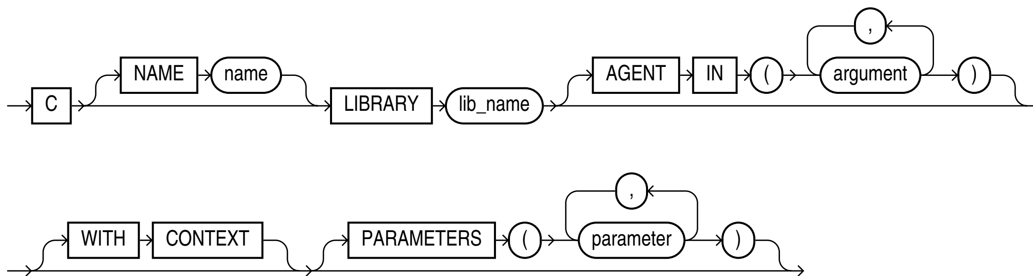



***call_spec* ::=**




***Java_declaration* ::=**

***C_declaration* ::=**




## Semantics

### OR REPLACE

Re-creates the type body if it exists, and recompiles it.

Users who were granted privileges on the type body before it was redefined can still access the type body without being regranted the privileges.

You can use this clause to add member subprogram definitions to specifications added with the `ALTER TYPE` ... `REPLACE` statement.

### *schema*

The name of the schema containing the type body. The default is your own schema.

### *type_name*

The name of an ADT.

### *subprog_decl_in_type*

The type of function or procedure subprogram associated with the type specification.

You must define a corresponding method name and optional parameter list in the type specification for each procedure or function declaration. For functions, you also must specify a return type.

### *proc_decl_in_type, func_decl_in_type*

A procedure or function subprogram declaration.

### *constructor_declaration*

A user-defined constructor subprogram declaration. The RETURN clause of a constructor function must be RETURN SELF AS RESULT. This setting indicates that the most specific type of the value returned by the constructor function is the most specific type of the SELF argument that was passed in to the constructor function.

**See Also:**

- "CREATE TYPE Statement" on page 14-69 for a list of restrictions on user-defined functions
- "Overloaded Subprograms" on page 8-13 for information about overloading subprogram names
- *Oracle Database Object-Relational Developer's Guide* for information about and examples of user-defined constructors

### *declare_section*

Declares items that are local to the procedure or function.

### *body*

Procedure or function statements.

### *call_spec*

The call specification that maps a Java or C method name, parameter types, and return type to their SQL counterparts.

The *Java_declaration* string identifies the Java implementation of the method.

**See Also:**

- *Oracle Database Java Developer's Guide* for an explanation of the parameters and semantics of the *Java_declaration*
- *Oracle Database Advanced Application Developer's Guide* for information about calling external procedures

### *map_order_func_declaration*

You can declare either one MAP method or one ORDER method, regardless of how many MEMBER or STATIC methods you declare. If you declare either a MAP or ORDER method, then you can compare object instances in SQL.

If you do not declare either method, then you can compare object instances only for equality or inequality. Instances of the same type definition are equal only if each pair of their corresponding attributes is equal.

### MAP MEMBER

Declares or implements a MAP member function that returns the relative position of a given instance in the ordering of all instances of the object. A MAP method is called implicitly and specifies an ordering of object instances by mapping them to values of a predefined scalar type. PL/SQL uses the ordering to evaluate Boolean expressions and to perform comparisons.

If the argument to the MAP method is null, then the MAP method returns null and the method is not invoked.

An type body can contain only one MAP method, which must be a function. The MAP function can have no arguments other than the implicit SELF argument.

#### ORDER MEMBER

Specifies an ORDER member function that takes an instance of an object as an explicit argument and the implicit SELF argument and returns either a negative integer, zero, or a positive integer, indicating that the implicit SELF argument is less than, equal to, or greater than the explicit argument, respectively.

If either argument to the ORDER method is null, then the ORDER method returns null and the method is not invoked.

When instances of the same ADT definition are compared in an ORDER BY clause, the database invokes the ORDER MEMBER *func_decl_in_type*.

An object specification can contain only one ORDER method, which must be a function having the return type NUMBER.

#### *func_decl_in_type*

A function subprogram declaration. See "CREATE PROCEDURE Statement" on page 14-49 and "CREATE FUNCTION Statement" on page 14-32 for the full syntax with all possible clauses.

#### AS EXTERNAL

An alternative way of declaring a C method. This clause is deprecated and is supported for backward compatibility only. Oracle recommends that you use the *call_spec* syntax with the *C_declaration*.

### Examples

Several examples of creating type bodies appear in the Examples section of "CREATE TYPE Statement" on page 14-69. For an example of re-creating a type body, see "Adding a Member Function: Example" on page 14-28.

### Related Topics

- "CREATE TYPE Statement" on page 14-69
- "DROP TYPE BODY Statement" on page 14-101
- "CREATE FUNCTION Statement" on page 14-32
- "CREATE PROCEDURE Statement" on page 14-49

# DROP FUNCTION Statement

The DROP FUNCTION statement drops a standalone stored function from the database.

**Note:** Do not use this statement to drop a function that is part of a package. Instead, either drop the entire package using the "DROP PACKAGE Statement" on page 14-94 or redefine the package without the function using the "CREATE PACKAGE Statement" on page 14-42 with the OR REPLACE clause.

Topics:

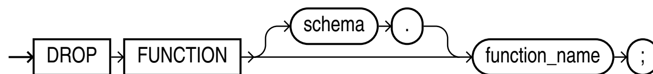
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

## Prerequisites

The function must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

## Syntax

***drop_function*::=**




## Semantics

***schema***

The name of the schema containing the function. The default is your own schema.

***function_name***

The name of the function to be dropped.

The database invalidates any local objects that depend on, or call, the dropped function. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error if you have not re-created the dropped function.

If any statistics types are associated with the function, then the database disassociates the statistics types with the FORCE option and drops any user-defined statistics collected with the statistics type.

**See Also:**

- *Oracle Database SQL Language Reference* for information about the `ASSOCIATE STATISTICS` statement
- *Oracle Database SQL Language Reference* for information about the `DISASSOCIATE STATISTICS` statement

## Example

**Dropping a Function: Example** This statement drops the function `SecondMax` in the sample schema `oe` and invalidates all objects that depend upon `SecondMax`:

```
DROP FUNCTION oe.SecondMax;
```

**See Also:** "Creating Aggregate Functions: Example" on page 14-38 for information about creating the `SecondMax` function

## Related Topics

- "ALTER FUNCTION Statement" on page 14-3
- "CREATE FUNCTION Statement" on page 14-32

# DROP LIBRARY Statement

The DROP LIBRARY statement drops an external procedure library from the database.

Topics:

- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

## Prerequisites

You must have the DROP ANY LIBRARY system privilege.

## Syntax

***drop_library*::=**

## Semantics

***library_name***

The name of the external procedure library being dropped.

## Example

**Dropping a Library: Example** The following statement drops the ext_lib library, which was created in "Creating a Library: Examples" on page 14-41:

```
DROP LIBRARY ext_lib;
```

## Related Topics

- "ALTER LIBRARY Statement" on page 14-6
- "CREATE LIBRARY Statement" on page 14-40

# DROP PACKAGE Statement

The DROP PACKAGE statement drops a stored package from the database. This statement drops the body and specification of a package.

**Note:** Do not use this statement to drop a single object from a package. Instead, re-create the package without the object using the "CREATE PACKAGE Statement" on page 14-42 and "CREATE PACKAGE BODY Statement" on page 14-45 with the OR REPLACE clause.

Topics:

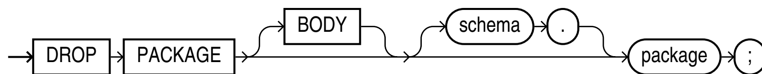
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

## Prerequisites

The package must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

## Syntax

***drop_package*::=**




## Semantics

### BODY

Drops only the body of the package. If you omit this clause, then the database drops both the body and specification of the package.

When you drop only the body of a package but not its specification, the database does not invalidate dependent objects. However, you cannot call a procedure or stored function declared in the package specification until you re-create the package body.

### *schema*

The name of the schema containing the package. The default is your own schema.

### *package*

The name of the package to be dropped.

The database invalidates any local objects that depend on the package specification. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error if you have not re-created the dropped package.

If any statistics types are associated with the package, then the database disassociates the statistics types with the FORCE clause and drops any user-defined statistics collected with the statistics types.

**See Also:**

- *Oracle Database SQL Language Reference* for information about the ASSOCIATE STATISTICS statement
- *Oracle Database SQL Language Reference* for information about the DISASSOCIATE STATISTICS statement

## Example

**Dropping a Package: Example** This statement drops the specification and body of the emp_mgmt package, which was created in "Creating a Package Body: Example" on page 14-46, invalidating all objects that depend on the specification:

```
DROP PACKAGE emp_mgmt;
```

## Related Topics

- "ALTER PACKAGE Statement" on page 14-8
- "CREATE PACKAGE Statement" on page 14-42
- "CREATE PACKAGE BODY Statement" on page 14-45

# DROP PROCEDURE Statement

The DROP PROCEDURE statement drops a standalone stored procedure from the database.

> **Note:** Do not use this statement to remove a procedure that is part of a package. Instead, either drop the entire package using the "DROP PACKAGE Statement" on page 14-94, or redefine the package without the procedure using the "CREATE PACKAGE Statement" on page 14-42 with the OR REPLACE clause.

Topics:

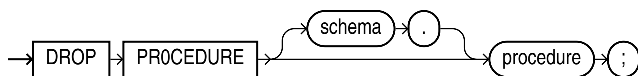
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

## Prerequisites

The procedure must be in your own schema or you must have the DROP ANY PROCEDURE system privilege.

## Syntax

***drop_procedure*::=**




## Semantics

***schema***

The name of the schema containing the procedure. The default is your own schema.

***procedure***

The name of the procedure to be dropped.

When you drop a procedure, the database invalidates any local objects that depend upon the dropped procedure. If you subsequently reference one of these objects, then the database tries to recompile the object and returns an error message if you have not re-created the dropped procedure.

## Example

**Dropping a Procedure: Example** This statement drops the procedure remove_emp owned by the user hr and invalidates all objects that depend upon remove_emp:

```
DROP PROCEDURE hr.remove_emp;
```

### Related Topics


- "ALTER PROCEDURE Statement" on page 14-11
- "CREATE PROCEDURE Statement" on page 14-49

# DROP TRIGGER Statement

The DROP TRIGGER statement drops a database trigger from the database.

Topics:

- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

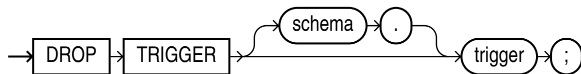
## Prerequisites

The trigger must be in your own schema or you must have the DROP ANY TRIGGER system privilege. To drop a trigger on DATABASE in another user's schema, you must also have the ADMINISTER DATABASE TRIGGER system privilege.

Topics:

## Syntax

***drop_trigger*::=**




## Semantics

***schema***

The name of the schema containing the trigger. The default is your own schema.

***trigger***

The name of the trigger to be dropped.

## Example

**Dropping a Trigger: Example** This statement drops the salary_check trigger in the schema hr:

```
DROP TRIGGER hr.salary_check;
```

## Related Topics

- "ALTER TRIGGER Statement" on page 14-14
- "CREATE TRIGGER Statement" on page 14-53

# DROP TYPE Statement

The DROP TYPE statement drops the specification and body of an ADT, varray type, or nested table type.

Topics:

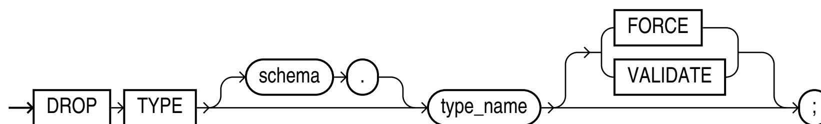
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

## Prerequisites

The ADT, varray type, or nested table type must be in your own schema or you must have the DROP ANY TYPE system privilege.

## Syntax

***drop_type*::=**




## Semantics

***schema***

The name of the schema containing the type. The default is your own schema.

***type_name***

The name of the object, varray, or nested table type to be dropped. You can drop only types with no type or table dependencies.

If *type_name* is a supertype, then this statement fails unless you also specify FORCE. If you specify FORCE, then the database invalidates all subtypes depending on this supertype.

If *type_name* is a statistics type, then this statement fails unless you also specify FORCE. If you specify FORCE, then the database first disassociates all objects that are associated with *type_name* and then drops *type_name*.

**See Also:**

- *Oracle Database SQL Language Reference* for information about the ASSOCIATE STATISTICS statement
- *Oracle Database SQL Language Reference* for information about the DISASSOCIATE STATISTICS statement

If *type_name* is an ADT that is associated with a statistics type, then the database first attempts to disassociate *type_name* from the statistics type and then drops *type_*

*name*. However, if statistics have been collected using the statistics type, then the database cannot disassociate *type_name* from the statistics type, and this statement fails.

If *type_name* is an implementation type for an index type, then the index type is marked INVALID.

If *type_name* has a public synonym defined on it, then the database also drops the synonym.

Unless you specify FORCE, you can drop only types that are standalone schema objects with no dependencies. This is the default behavior.

> **See Also:** *Oracle Database SQL Language Reference* for information about the CREATE INDEXTYPE statement

**FORCE**

Drops the type even if it has dependent database objects. The database marks UNUSED all columns dependent on the type to be dropped, and those columns become inaccessible.

> **Caution:** Oracle recommends against specifying FORCE to drop object types with dependencies. This operation is not recoverable and might make the data in the dependent tables or columns inaccessible.

**VALIDATE**

Causes the database to check for stored instances of this type in substitutable columns of any of its supertypes. If no such instances are found, then the database completes the drop operation.

This clause is meaningful only for subtypes. Oracle recommends the use of this option to safely drop subtypes that do not have any explicit type or table dependencies.

## Example

**Dropping an ADT: Example** This statement removes the ADT person_t. See "Type Hierarchy Example" on page 14-83 for the example that creates this ADT. Any columns that are dependent on person_t are marked UNUSED and become inaccessible.

```
DROP TYPE person_t FORCE;
```

## Related Topics

- "ALTER TYPE Statement" on page 14-17
- "CREATE TYPE Statement" on page 14-69
- "CREATE TYPE BODY Statement" on page 14-86

# DROP TYPE BODY Statement

The DROP TYPE BODY statement drops the body of an ADT, varray type, or nested table type. When you drop a type body, the type specification still exists, and you can re-create the type body. Prior to re-creating the body, you can still use the type, although you cannot call its member functions.

Topics:

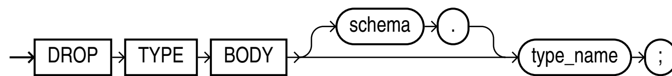
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

## Prerequisites

The type body must be in your own schema or you must have the DROP ANY TYPE system privilege.

## Syntax

***drop_type_body*::=**




## Semantics

***schema***

The name of the schema containing the type. The default is your own schema.

***type_name***

The name of the type body to be dropped.

**Restriction on *type_name*** You can drop a type body only if it has no type or table dependencies.

## Example

**Dropping an ADT Body: Example** This statement removes the ADT body data_typ1. See "ADT Examples" on page 14-81 for the example that creates this ADT.

```
DROP TYPE BODY data_typ1;
```

## Related Topics

- "ALTER TYPE Statement" on page 14-17
- "CREATE TYPE Statement" on page 14-69
- "CREATE TYPE BODY Statement" on page 14-86

# A

# PL/SQL Source Code Wrapping

This appendix explains what wrapping is, why you wrap PL/SQL code, and how to do it.

Topics:

- Overview of Wrapping
- Guidelines for Wrapping
- Limitations of Wrapping
- Wrapping PL/SQL Code with wrap Utility
- Wrapping PL/QL Code with DBMS_DDL Subprograms

## Overview of Wrapping

**Wrapping** is the process of hiding PL/SQL source code. Wrapping helps to protect your source code by making it more difficult for others to view it.

You can wrap PL/SQL source code with either the `wrap` utility or `DBMS_DDL` subprograms. The `wrap` utility wraps a single source file, such as a SQL*Plus script. The `DBMS_DDL` subprograms wrap a single dynamically generated PL/SQL unit, such as a single `CREATE PROCEDURE` statement.

Wrapped source files can be moved, backed up, and processed by SQL*Plus and the Import and Export utilities, but they are not visible through the static data dictionary views `*_SOURCE`.

---

**Note:** Wrapping a file that is already wrapped has no effect on the file.

---

## Guidelines for Wrapping

- Wrap only the body of a package or ADT, not the specification.

  This allows other developers to see the information they must use the package or type, but prevents them from seeing its implementation.

- Wrap code only after you have finished editing it.

  You cannot edit PL/SQL source code inside wrapped files. Either wrap your code after it is ready to ship to users or include the wrapping operation as part of your build environment.

To change wrapped PL/SQL code, edit the original source file and then wrap it again.

- Before distributing a wrapped file, view it in a text editor to be sure that all important parts are wrapped.

## Limitations of Wrapping

- Wrapping is not a secure method for hiding passwords or table names.

  Wrapping a PL/SQL unit helps prevent most users from examining the source code, but might not stop all of them.

- Wrapping does not hide the source code for triggers.

  To hide the workings of a trigger, write a one-line trigger that invokes a wrapped subprogram.

- Wrapping does not detect syntax or semantic errors.

  Wrapping detects only tokenization errors (for example, runaway strings), not syntax or semantic errors (for example, nonexistent tables or views). Syntax or semantic errors are detected during PL/SQL compilation or when running the output file in SQL*Plus.

- Wrapped PL/SQL units are not downward-compatible.

  Wrapped PL/SQL units are upward-compatible between Oracle Database releases, but are not downward-compatible. For example, you can load files processed by the V8.1.5 `wrap` utility into a V8.1.6 Oracle Database, but you cannot load files processed by the V8.1.6 `wrap` utility into a V8.1.5 Oracle Database.

**See Also:**


- "Limitations of the wrap Utility" on page A-4
- "Limitation of the DBMS_DDL.WRAP Function" on page A-6


## Wrapping PL/SQL Code with wrap Utility

The `wrap` utility processes an input SQL file and wraps only the PL/SQL units in the file, such as a package specification, package body, function, procedure, type specification, or type body. It does not wrap PL/SQL content in anonymous blocks or triggers or non-PL/SQL code.

The `wrap` utility does not need to connect to Oracle Database (in fact, it cannot connect to Oracle Database).

To run the wrap utility, enter the `wrap` command at your operating system prompt using this syntax (with no spaces around the equal signs):

```
wrap iname=input_file [ oname=output_file ]
```

*`input_file`* is the name of a file containing SQL statements, which you typically run using SQL*Plus. If you omit the file extension, `.sql` is assumed. For example, these commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql
```

You can specify a different file extension. For example:

```
wrap iname=/mydir/myfile.src
```

*output_file* is the name of the wrapped file to be created. If you omit the oname option, *output_file* has the same name as *input_file*, but with the extension .plb. For example, these commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql oname=/mydir/myfile.plb
```

You can specify a different output file name and extension. For example:

```
wrap iname=/mydir/myfile oname=/yourdir/yourfile.out
```

---

**Note:** If *input_file* is already wrapped, *output_file* is identical to *input_file*.

---

Topics:

- Input and Output Files for the PL/SQL wrap Utility
- Running the wrap Utility
- Limitations of the wrap Utility

## Input and Output Files for the PL/SQL wrap Utility

The input file can contain any combination of SQL statements. Most statements are passed through unchanged. CREATE statements that define subprograms, packages, or ADTs are wrapped; their bodies are replaced by a scrambled form that the PL/SQL compiler understands.

These CREATE statements are wrapped:

```
CREATE [OR REPLACE] FUNCTION function_name
CREATE [OR REPLACE] PROCEDURE procedure_name
CREATE [OR REPLACE] PACKAGE package_name
CREATE [OR REPLACE] PACKAGE BODY package_name
CREATE [OR REPLACE] TYPE type_name AS OBJECT
CREATE [OR REPLACE] TYPE type_name UNDER type_name
CREATE [OR REPLACE] TYPE BODY type_name
```

The CREATE [OR REPLACE] TRIGGER statement, and [DECLARE] BEGIN END anonymous blocks, are not wrapped. All other SQL statements are passed unchanged to the output file.

All comment lines in the unit being wrapped are deleted, except for those in a CREATE OR REPLACE header and C-style comments (delimited by /* */).

The output file is a text file, which you can run as a script in SQL*Plus to set up your PL/SQL subprograms and packages. Run a wrapped file as follows:

```
SQL> @wrapped_file_name.plb;
```

## Running the wrap Utility

For example, assume that the wraptest.sql file contains:

```
CREATE PROCEDURE wraptest IS
  TYPE emp_tab IS TABLE OF employees%ROWTYPE INDEX BY PLS_INTEGER;
  all_emps  emp_tab;
BEGIN
  SELECT * BULK COLLECT INTO all_emps FROM employees;
```

```
  FOR i IN 1..10 LOOP
    DBMS_OUTPUT.PUT_LINE('Emp Id: ' || all_emps(i).employee_id);
  END LOOP;
END;
/
```

To wrap the file, run this command from the operating system prompt:

```
wrap iname=wraptest.sql
```

The output of the `wrap` utility is similar to:

```
PL/SQL Wrapper: Release 10.2.0.0.0 on Tue Apr 26 16:47:39 2005
Copyright (c) 1993, 2005, Oracle.  All rights reserved.
Processing wraptest.sql to wraptest.plb
```

If you view the contents of the `wraptest.plb` text file, the first line is `CREATE PROCEDURE wraptest wrapped` and the rest of the file contents is hidden.

You can run `wraptest.plb` in SQL*Plus to run the SQL statements in the file:

```
SQL> @wraptest.plb
```

After the `wraptest.plb` is run, you can run the procedure that was created:

```
SQL> CALL wraptest();
```

## Limitations of the wrap Utility

- The PL/SQL code to be wrapped cannot include substitution variables using the SQL*Plus `DEFINE` notation.

  Wrapped source code is parsed by the PL/SQL compiler, not by SQL*Plus.
- The wrap utility removes most comments from wrapped files.

  See "Input and Output Files for the PL/SQL wrap Utility" on page A-3.

# Wrapping PL/QL Code with DBMS_DDL Subprograms

The `DBMS_DDL` package contains procedures for wrapping a single PL/SQL unit, such as a package specification, package body, function, procedure, type specification, or type body. These overloaded subprograms provide a mechanism for wrapping dynamically generated PL/SQL units that are created in a database.

The `DBMS_DDL` package contains the `WRAP` functions and the `CREATE_WRAPPED` procedures. The `CREATE_WRAPPED` both wraps the text and creates the PL/SQL unit. When invoking the wrap procedures, use the fully qualified package name, `SYS.DBMS_DDL`, to avoid any naming conflicts and the possibility that someone might create a local package called `DBMS_DDL` or define the `DBMS_DDL` public synonym. The input `CREATE OR REPLACE` statement runs with the privileges of the user who invokes `DBMS_DDL.WRAP` or `DBMS_DDL.CREATE_WRAPPED`.

The `DBMS_DDL` package also provides the `MALFORMED_WRAP_INPUT` exception (ORA-24230) which is raised if the input to the wrap procedures is not a valid PL/SQL unit.

**Note:** Wrapping a PL/SQL unit that is already wrapped has no effect on the unit.

Topics:

- DBMS_DDL.CREATE_WRAPPED Procedure
- Limitation of the DBMS_DDL.WRAP Function

**See Also:** *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_DDL package

## DBMS_DDL.CREATE_WRAPPED Procedure

In Example A–1 CREATE_WRAPPED is used to dynamically create and wrap a package specification and a package body in a database.

***Example A–1 Wrapping Package with DBMS_DDL.CREATE_WRAPPED Procedure***

```
DECLARE
  package_text  VARCHAR2(32767); -- text for creating package spec & body

  FUNCTION generate_spec (pkgname VARCHAR2) RETURN VARCHAR2 AS
  BEGIN
    RETURN 'CREATE PACKAGE ' || pkgname || ' AS
      PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
      PROCEDURE fire_employee (emp_id NUMBER);
      END ' || pkgname || ';';
  END generate_spec;

  FUNCTION generate_body (pkgname VARCHAR2) RETURN VARCHAR2 AS
  BEGIN
    RETURN 'CREATE PACKAGE BODY ' || pkgname || ' AS
      PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS
      BEGIN
        UPDATE employees
          SET salary = salary + amount WHERE employee_id = emp_id;
      END raise_salary;
      PROCEDURE fire_employee (emp_id NUMBER) IS
      BEGIN
        DELETE FROM employees WHERE employee_id = emp_id;
      END fire_employee;
    END ' || pkgname || ';';
  END generate_body;

BEGIN
  -- Generate package spec
  package_text := generate_spec('emp_actions');

  -- Create wrapped package spec
  DBMS_DDL.CREATE_WRAPPED(package_text);

  -- Generate package body
  package_text := generate_body('emp_actions');

  -- Create wrapped package body
  DBMS_DDL.CREATE_WRAPPED(package_text);
END;
/

-- Invoke procedure from wrapped package
CALL emp_actions.raise_salary(120, 100);
```

When you check the static data dictionary views *_SOURCE, the source is wrapped, or hidden, so that others cannot view the code details. For example:

```
SELECT text FROM USER_SOURCE WHERE name = 'EMP_ACTIONS';
```

Result is similar to:

```
TEXT
--------------------------------------------------------------------
PACKAGE emp_actions WRAPPED
a000000
1f
abcd
...
2 rows selected.
```

## Limitation of the DBMS_DDL.WRAP Function

If you invoke DBMS_SQL.PARSE (when using an overload where the statement formal has data type VARCHAR2A or VARCHAR2S for text which exceeds 32767 bytes) on the output of DBMS_DDL.WRAP, then you must set the LFFLG parameter to FALSE. Otherwise DBMS_SQL.PARSE adds lines to the wrapped unit which corrupts the unit.

# B

# PL/SQL Name Resolution

This appendix explains how PL/SQL resolves ambiguous references to identifiers.

Topics:


- What is Name Resolution?
- Examples of Qualified Names and Dot Notation
- How Name Resolution Differs in PL/SQL and SQL
- What is Capture?
- Avoiding Inner Capture in DML Statements



**See Also:** "Resolution of Names in Static SQL Statements" on page 6-3


## What is Name Resolution?

During compilation, the PL/SQL compiler determines which objects are associated with each name in a PL/SQL subprogram. A name might refer to a local variable, a table, a package, a subprogram, a schema, and so on. When a subprogram is recompiled, that association might change if objects were created or deleted.

A declaration or definition in an inner scope can hide another in an outer scope. In Example B–1, the declaration of variable `client` hides the definition of data type `Client` because PL/SQL names are not case-sensitive. In the inner block, the reference to the data type must be qualified with the label of outer block.

***Example B–1 Resolving Global and Local Variable Names***

```
BEGIN
  <<block1>>
  DECLARE
    TYPE Client IS RECORD (
      first_name VARCHAR2(20),
      last_name VARCHAR2(25)
    );
    TYPE Customer IS RECORD (
      first_name VARCHAR2(20),
      last_name VARCHAR2(25)
    );
  BEGIN
    DECLARE
      client  Customer;         -- declaration of variable client
      lead      block1.Client;  -- qualified reference to type Client
    BEGIN
```

```
      NULL;
    END;
  END;
END;
/
```

You can refer to data type `Client` by qualifying the reference with block label `block1`.

In these `CREATE TYPE` statements, the second statement generates a warning. Creating an attribute named `manager` hides the type named `manager`, so the declaration of the second attribute does not run correctly:

```
CREATE OR REPLACE TYPE manager AS OBJECT (dept NUMBER);
/
CREATE OR REPLACE TYPE person AS OBJECT (manager NUMBER, mgr manager)
    -- raises a warning;
/
```

## Name Resolution

In ambiguous SQL statements, the names of database columns take precedence over the names of local variables and formal parameters. For example, if a variable and a column with the same name are used in a `WHERE` clause, SQL considers both names to refer to the column.

> **Caution:** When a variable name is interpreted as a column name, data can be deleted unintentionally, as Example B–2 shows. Example B–2 also shows two ways to avoid this error.

***Example B–2 Block Label for Name Resolution***

```
DROP TABLE employees2;
CREATE TABLE employees2 AS
  SELECT last_name FROM employees;

-- Deletes everyone, because both LAST_NAMEs refer to the column:

BEGIN
  DELETE FROM employees2
  WHERE LAST_NAME = LAST_NAME;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
END;
/
```

Result:

```
Deleted 107 rows.
```

Undo deletion:

```
ROLLBACK;
```

Avoid error by giving column and variable different names:

```
DECLARE
  last_name    VARCHAR2(10) := 'King';
  v_last_name  VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2
```

```
  WHERE last_name = v_last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
END;
/
```

Result:

```
Deleted 2 rows.
```

Undo deletion:

```
ROLLBACK;
```

Avoid error by qualifying variable with block name:

```
<<main>> -- Label block for future reference
DECLARE
  last_name    VARCHAR2(10) := 'King';
  v_last_name  VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2
  WHERE last_name = main.last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
END;
/
```

Result:

```
Deleted 2 rows.
```

Undo deletion:

```
ROLLBACK;
```

You can use a subprogram name to qualify references to local variables and formal parameters, as in Example B–3.

#### *Example B–3 Subprogram Name for Name Resolution*

```
DECLARE
  FUNCTION dept_name (department_id IN NUMBER)
    RETURN departments.department_name%TYPE
  IS
    department_name  departments.department_name%TYPE;
  BEGIN
    SELECT department_name INTO dept_name.department_name
      --   ^column              ^local variable
    FROM departments
    WHERE department_id = dept_name.department_id;
    --    ^column         ^formal parameter
    RETURN department_name;
  END dept_name;
BEGIN
 FOR item IN (SELECT department_id FROM departments)
   LOOP
     DBMS_OUTPUT.PUT_LINE
       ('Department: ' || dept_name(item.department_id));
   END LOOP;
END;
/
```

Result:

```
Department: Administration
Department: Marketing
Department: Purchasing
Department: Human Resources
Department: Shipping
Department: IT
Department: Public Relations
Department: Sales
Department: Executive
Department: Finance
Department: Accounting
Department: Treasury
Department: Corporate Tax
Department: Control And Credit
Department: Shareholder Services
Department: Benefits
Department: Manufacturing
Department: Construction
Department: Contracting
Department: Operations
Department: IT Support
Department: NOC
Department: IT Helpdesk
Department: Government Sales
Department: Retail Sales
Department: Recruiting
Department: Payroll
```

## Examples of Qualified Names and Dot Notation

During name resolution, the compiler can encounter various forms of references—unqualified names, qualified names, indexed components of collections, and so on. Example B–4 shows some of these.

***Example B–4 Dot Notation for Qualifying Names***

```
CREATE OR REPLACE PACKAGE pkg1 AS
  m NUMBER;
  TYPE t1 IS RECORD (a NUMBER);
  v1 t1;
  TYPE t2 IS TABLE OF t1 INDEX BY PLS_INTEGER;
  v2 t2;
  FUNCTION f1 (p1 NUMBER) RETURN t1;
  FUNCTION f2 (q1 NUMBER) RETURN t2;
END pkg1;
/

CREATE OR REPLACE PACKAGE BODY pkg1 AS
  FUNCTION f1 (p1 NUMBER) RETURN t1 IS
    n NUMBER;
  BEGIN
     n := m;             -- Unqualified variable name
     n := pkg1.m;        -- Variable name qualified by package name
     n := pkg1.f1.p1;    -- Parameter name qualified by function name,
                         --  which is qualified by package name
     n := v1.a;          -- Variable name followed by component name
     n := pkg1.v1.a;     -- Variable name qualified by package name
                         --  and followed by component name
```

```
    n := v2(10).a;      -- Indexed name followed by component name
    n := f1(10).a;      -- Function call followed by component name
    n := f2(10)(10).a;  -- Function call followed by indexed name
                        --  and followed by component name
    n := hr.pkg1.f2(10)(10).a;  -- Schema name, package name, function call,
                                --  index, component name
    v1.a := p1;
    RETURN v1;
  END f1;

  FUNCTION f2 (q1 NUMBER) RETURN t2 IS
    v_t1 t1;
    v_t2 t2;
  BEGIN
    v_t1.a := q1;
    v_t2(1) := v_t1;
    RETURN v_t2;
  END f2;
END pkg1;
/
```

An outside reference to a private variable declared in a function body is not legal. For example, an outside reference to the variable `n` declared in function `f1`, such as `hr.pkg1.f1.n` from function `f2`, raises an exception. See "Private and Public Items in Packages" on page 10-15.

Dot notation is used for identifying record fields, object attributes, and items inside packages or other schemas. When you combine these items, you might need to use expressions with multiple levels of dots, where it is not always clear what each dot refers to. Some of the combinations are:

- Field or attribute of a function return value, for example:

  ```
  func_name().field_name
  func_name().attribute_name
  ```

- Schema object owned by another schema, for example:

  ```
  schema_name.table_name
  schema_name.procedure_name()
  schema_name.type_name.member_name()
  ```

- Package object owned by another user, for example:

  ```
  schema_name.package_name.procedure_name()
  schema_name.package_name.record_name.field_name
  ```

- Record containing an ADT, for example:

  ```
  record_name.field_name.attribute_name
  record_name.field_name.member_name()
  ```

## How Name Resolution Differs in PL/SQL and SQL

The name resolution rules for PL/SQL and SQL are similar. You can avoid the few differences if you follow the capture avoidance rules. For compatibility, the SQL rules are more permissive than the PL/SQL rules. SQL rules, which are mostly context sensitive, recognize as legal more situations and DML statements than the PL/SQL rules.

- PL/SQL uses the same name-resolution rules as SQL when the PL/SQL compiler processes a SQL statement, such as a DML statement. For example, for a name such as HR.JOBS, SQL matches objects in the HR schema first, then packages, types, tables, and views in the current schema.
- PL/SQL uses a different order to resolve names in PL/SQL statements such as assignments and subprogram calls. In the case of a name HR.JOBS, PL/SQL searches first for packages, types, tables, and views named HR in the current schema, then for objects in the HR schema.

For information about SQL naming rules, see *Oracle Database SQL Language Reference*.

# What is Capture?

When a declaration or type definition in another scope prevents the compiler from resolving a reference correctly, that declaration or definition is said to capture the reference. Capture is usually the result of migration or schema evolution. There are three kinds of capture: inner, same-scope, and outer. Inner and same-scope capture apply only in SQL scope.

Topics:

- Inner Capture
- Same-Scope Capture
- Outer Capture

## Inner Capture

An inner capture occurs when a name in an inner scope no longer refers to an entity in an outer scope:

- The name might now resolve to an entity in an inner scope.
- The program might cause an error, if some part of the identifier is captured in an inner scope and the complete reference cannot be resolved.

If the reference points to a different but valid name, you might not know why the program is acting differently.

In the following example, the reference to col2 in the inner SELECT statement binds to column col2 in table tab1 because table tab2 has no column named col2:

```
DROP TABLE tab1;
CREATE TABLE tab1 (col1 NUMBER, col2 NUMBER);
INSERT INTO tab1 (col1, col2) VALUES (100, 10);

DROP TABLE tab2;
CREATE TABLE tab2 (col1 NUMBER);
INSERT INTO tab2 (col1) VALUES (100);

CREATE OR REPLACE PROCEDURE proc AS
  CURSOR c1 IS
    SELECT * FROM tab1
    WHERE EXISTS (SELECT * FROM tab2 WHERE col2 = 10);
BEGIN
  NULL;
END;
/
```

Add a column named col2 to table tab2:

```
ALTER TABLE tab2 ADD (col2 NUMBER);
```

Now procedure proc is invalid and will be recompiled automatically upon next use. However, upon recompilation, the col2 in the inner SELECT statement binds to column col2 in table tab2 because tab2 is in the inner scope. Thus, the reference to col2 is captured by the addition of column col2 to table tab2.

Collections and ADTs are also vulnerable to inner capture. In the following example, the reference to hr.tab2.a resolves to attribute a of column tab2 in table tab1 through table alias hr, which is visible in the outer scope of the query:

```
CREATE OR REPLACE TYPE type1 AS OBJECT (a NUMBER);
/
DROP TABLE tab1;
CREATE TABLE tab1 (tab2 type1);
INSERT INTO tab1 (tab2) VALUES (type1(10));

DROP TABLE tab2;
CREATE TABLE tab2 (x NUMBER);
INSERT INTO tab2 (x) VALUES (10);

/* Alias tab1 with same name as schema name,
   a bad practice used here for illustration purpose.
   Note lack of alias in second SELECT statement. */

SELECT * FROM tab1 hr
WHERE EXISTS (SELECT * FROM hr.tab2 WHERE x = hr.tab2.a);
```

Result:

```
TAB2(A)
---------------

TYPE1(10)

1 row selected.
```

Suppose that you add a column named a to table hr.tab2, which appears in the inner subquery. When the query is processed, an inner capture occurs because the reference to hr.tab2.a resolves to column a of table tab2 in schema hr.

To avoid inner captures, follow the rules in "Avoiding Inner Capture in DML Statements" on page B-8. According to those rules, revise the preceding query as follows:

```
SELECT * FROM hr.tab1 p1
WHERE EXISTS (SELECT * FROM hr.tab2 p2 WHERE p2.x = p1.tab2.a);
```

## Same-Scope Capture

In SQL scope, a same-scope capture occurs when a column is added to one of two tables used in a join, so that the same column name exists in both tables. Previously, you could refer to that column name in a join query. To avoid an error, now you must qualify the column name with the table name.

## Outer Capture

An outer capture occurs when a name in an inner scope, which once resolved to an entity in an inner scope, is resolved to an entity in an outer scope. SQL and PL/SQL

are designed to prevent outer captures. You need not take any action to avoid this condition.

# Avoiding Inner Capture in DML Statements

You can avoid inner capture in DML statements by following these rules:

- Specify an alias for each table in the DML statement.
- Keep table aliases unique throughout the DML statement.
- Avoid table aliases that match schema names used in the query.
- Qualify each column reference with the table alias.

Qualifying a reference with `schema_name.table_name` does not prevent inner capture if the statement refers to tables with columns of an Abstract Data Type (ADT). Such columns are vulnerable to additional inner capture situations. To minimize problems, the name-resolution algorithm includes these rules for the use of table aliases.

Topics:

- Qualifying References to Attributes and Methods
- Qualifying References to Row Expressions

## Qualifying References to Attributes and Methods

To reference an attribute or method of a table, you must give the table an alias and use the alias to qualify the reference to the attribute or method.

```
CREATE OR REPLACE TYPE t1 AS OBJECT (x NUMBER);
/
DROP TABLE tb1;
CREATE TABLE tb1 (col1 t1);
```

The references in the following `INSERT` statements do not need aliases, because they have no column lists:

```
BEGIN
  INSERT INTO tb1 VALUES ( t1(10) );
  INSERT INTO tb1 VALUES ( t1(20) );
  INSERT INTO tb1 VALUES ( t1(30) );
END;
/
```

The following references cause error ORA-00904:

```
UPDATE tb1 SET col1.x = 10 WHERE col1.x = 20;

UPDATE tb1 SET tb1.col1.x = 10 WHERE tb1.col1.x = 20;

UPDATE hr.tb1 SET hr.tb1.col1.x = 10 WHERE hr.tb1.col1.x = 20;

DELETE FROM tb1 WHERE tb1.col1.x = 10;
```

The following references cause error ORA-00904:

```
UPDATE hr.tb1 t set t.col1.x = 10 WHERE t.col1.x = 20;

DECLARE
  y NUMBER;
```

```
BEGIN
  SELECT t.col1.x INTO y FROM tb1 t WHERE t.col1.x = 30;
END;
/

DELETE FROM tb1 t WHERE t.col1.x = 10;
```

## Qualifying References to Row Expressions

Row expressions must resolve as references to table aliases. You can pass row expressions to the operators `REF` and `VALUE`, and you can use row expressions in the `SET` clause of an `UPDATE` statement. For example:

```
CREATE OR REPLACE TYPE t1 AS OBJECT (x number);
/
DROP TABLE ot1;
CREATE TABLE ot1 OF t1;

BEGIN
  INSERT INTO ot1 VALUES (t1(10));
  INSERT INTO ot1 VALUES (20);
  INSERT INTO ot1 VALUES (30);
END;
/
```

The following references cause error ORA-00904:

```
UPDATE ot1 SET VALUE(ot1.x) = t1(20) WHERE VALUE(ot1.x) = t1(10);

DELETE FROM ot1 WHERE VALUE(ot1) = (t1(10));
```

The following references, with table aliases, are correct:

```
UPDATE ot1 o SET o = (t1(20)) WHERE o.x = 10;

DECLARE
  n_ref  REF t1;
BEGIN
  SELECT REF(o) INTO n_ref FROM ot1 o WHERE VALUE(o) = t1(30);
END;
/

DECLARE
  n t1;
BEGIN
  SELECT VALUE(o) INTO n FROM ot1 o WHERE VALUE(o) = t1(30);
END;
/

DECLARE
  n NUMBER;
BEGIN
  SELECT o.x INTO n FROM ot1 o WHERE o.x = 30;
END;
/

DELETE FROM ot1 o WHERE VALUE(o) = (t1(20));
```

# C

# PL/SQL Program Limits

This appendix describes the program limits that are imposed by the PL/SQL language. PL/SQL is based on the programming language Ada. As a result, PL/SQL uses a variant of Descriptive Intermediate Attributed Notation for Ada (DIANA), a tree-structured intermediate language. It is defined using a metanotation called Interface Definition Language (IDL). DIANA is used internally by compilers and other tools.

At compile time, PL/SQL source code is translated into system code. Both the DIANA and system code for a subprogram or package are stored in the database. At run time, they are loaded into the shared memory pool. The DIANA is used to compile dependent subprograms; the system code simply runs.

In the shared memory pool, a package specification, ADT specification, standalone subprogram, or anonymous block is limited to 67108864 (2**26) DIANA nodes which correspond to tokens such as identifiers, keywords, operators, and so on. This allows for ~6,000,000 lines of code unless you exceed limits imposed by the PL/SQL compiler, some of which are given in Table C–1.

***Table C–1 PL/SQL Compiler Limits***

| Item | Limit |
|---|---|
| bind variables passed to a program unit | 32768 |
| exception handlers in a program unit | 65536 |
| fields in a record | 65536 |
| levels of block nesting | 255 |
| levels of record nesting | 32 |
| levels of subquery nesting | 254 |
| levels of label nesting | 98 |
| levels of nested collections | no predefined limit |
| magnitude of a PLS_INTEGER or BINARY_ INTEGERvalue | -2147483648..2147483647 |
| number of formal parameters in an explicit cursor, function, or procedure | 65536 |
| objects referenced by a program unit | 65536 |
| precision of a FLOAT value (binary digits) | 126 |
| precision of a NUMBER value (decimal digits) | 38 |
| precision of a REAL value (binary digits) | 63 |

*Table C–1 (Cont.) PL/SQL Compiler Limits*

| Item | Limit |
|---|---|
| size of an identifier (characters) | 30 |
| size of a string literal (bytes) | 32767 |
| size of a CHAR value (bytes) | 32767 |
| size of a LONG value (bytes) | 32760 |
| size of a LONG RAW value (bytes) | 32760 |
| size of a RAW value (bytes) | 32767 |
| size of a VARCHAR2 value (bytes) | 32767 |
| size of an NCHAR value (bytes) | 32767 |
| size of an NVARCHAR2 value (bytes) | 32767 |
| size of a BFILE value (bytes) | 4G * value of DB_BLOCK_SIZE parameter |
| size of a BLOB value (bytes) | 4G * value of DB_BLOCK_SIZE parameter |
| size of a CLOB value (bytes) | 4G * value of DB_BLOCK_SIZE parameter |
| size of an NCLOB value (bytes) | 4G * value of DB_BLOCK_SIZE parameter |

To estimate how much memory a program unit requires, you can query the static data dictionary view `USER_OBJECT_SIZE`. The column `PARSED_SIZE` returns the size (in bytes) of the "flattened" DIANA. For example:

```
CREATE OR REPLACE PACKAGE pkg1 AS
  TYPE numset_t IS TABLE OF NUMBER;
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED;
END pkg1;
/

CREATE PACKAGE BODY pkg1 AS
  -- FUNCTION f1 returns a collection of elements (1,2,3,... x)
  FUNCTION f1(x NUMBER) RETURN numset_t PIPELINED IS
  BEGIN
    FOR i IN 1..x LOOP
      PIPE ROW(i);
    END LOOP;
    RETURN;
  END f1;
END pkg1;
/
```

SQL*Plus commands for formatting results of next query:

```
COLUMN name FORMAT A4
COLUMN type FORMAT A12
COLUMN source_size FORMAT 999
COLUMN parsed_size FORMAT 999
COLUMN code_size FORMAT 999
COLUMN error_size FORMAT 999
```

Query:

```
SELECT * FROM user_object_size WHERE name = 'PKG1';
```

Result:

```
NAME TYPE         SOURCE_SIZE PARSED_SIZE CODE_SIZE ERROR_SIZE
---- ------------ ----------- ----------- --------- ----------
PKG1 PACKAGE BODY         233         103       314          0
PKG1 PACKAGE              112         464       262          0
```

Unfortunately, you cannot estimate the number of DIANA nodes from the parsed size. Two program units with the same parsed size might require 1500 and 2000 DIANA nodes, respectively because, for example, the second unit contains more complex SQL statements.

When a PL/SQL block, subprogram, package, or schema-level user-defined type exceeds a size limit, you get an error such as `PLS-00123: program too large`. Typically, this problem occurs with packages or anonymous blocks. With a package, the best solution is to divide it into smaller packages. With an anonymous block, the best solution is to redefine it as a group of subprograms, which can be stored in the database.

For more information about the limits on data types, see Chapter 3, "PL/SQL Data Types." For limits on collection subscripts, see "Collection Element References" on page 5-12.

# D

# PL/SQL Reserved Words and Keywords

**Reserved words** and **keywords** are identifiers that have special meaning in PL/SQL.

You cannot use reserved words as ordinary user-defined identifiers. You can use keywords as ordinary user-defined identifiers, and you can use reserved words as quoted user-defined identifiers, but neither practice is recommended.

Like all PL/SQL identifiers, reserved words and keywords are case-insensitive.

Table D–1 lists the PL/SQL reserved words.

Table D–2 lists the PL/SQL keywords.

Some of the words in this appendix are also reserved by SQL. You can display them with the dynamic performance view `V$RESERVED_WORDS`. For information about this view, see*Oracle Database Reference*.

***Table D–1 PL/SQL Reserved Words***

| Begins with: | Reserved Words |
|---|---|
| A | ALL, ALTER, AND, ANY, AS, ASC, AT |
| B | BEGIN, BETWEEN, BY |
| C | CASE, CHECK, CLUSTERS, CLUSTER, COLAUTH, COLUMNS, COMPRESS, CONNECT, CRASH, CREATE, CURRENT |
| D | DECLARE, DEFAULT, DELETE, DESC, DISTINCT, DROP |
| E | ELSE, END, EXCEPTION, EXCLUSIVE, EXISTS |
| F | FETCH, FOR, FROM |
| G | GOTO, GRANT, GROUP |
| H | HAVING |
| I | IDENTIFIED, IF, IN, INDEXES, INDEX, INSERT, INTERSECT, INTO, IS |
| L | LIKE, LOCK |
| M | MINUS, MODE |
| N | NOCOMPRESS, NOT, NOWAIT, NULL |
| O | OF, ON, OPTION, OR, ORDER, OVERLAPS |
| P | PRIOR, PROCEDURE, PUBLIC |
| R | RESOURCE, REVOKE |
| S | SELECT, SHARE, SIZE, SQL, START |
| T | TABAUTH, TABLE, THEN, TO |
| U | UNION, UNIQUE, UPDATE, USE |
| V | VALUES, VIEW, VIEWS |
| W | WHEN, WHERE, WITH |

*Table D–2 PL/SQL Keywords*

| Begins with: | Keywords |
|---|---|
| A | A, ADD, AGENT, AGGREGATE, ARRAY, ATTRIBUTE, AUTHID, AVG |
| B | BFILE_BASE, BINARY, BLOB_BASE, BLOCK, BODY, BOTH, BOUND, BULK, BYTE |
| C | C, CALL, CALLING, CASCADE, CHAR, CHAR_BASE, CHARACTER, CHARSETFORM, CHARSETID, CHARSET, CLOB_BASE, CLOSE, COLLECT, COMMENT, COMMIT, COMMITTED, COMPILED, CONSTANT, CONSTRUCTOR, CONTEXT, CONTINUE, CONVERT, COUNT, CURSOR, CUSTOMDATUM |
| D | DANGLING, DATA, DATE, DATE_BASE, DAY, DEFINE, DETERMINISTIC, DOUBLE, DURATION |
| E | ELEMENT, ELSIF, EMPTY, ESCAPE, EXCEPT, EXCEPTIONS, EXECUTE, EXIT, EXTERNAL |
| F | FINAL, FIXED, FLOAT, FORALL, FORCE, FUNCTION |
| G | GENERAL |
| H | HASH, HEAP, HIDDEN, HOUR |
| I | IMMEDIATE, INCLUDING, INDICATOR, INDICES, INFINITE, INSTANTIABLE, INT, INTERFACE, INTERVAL, INVALIDATE, ISOLATION |
| J | JAVA |
| L | LANGUAGE, LARGE, LEADING, LENGTH, LEVEL, LIBRARY, LIKE2, LIKE4, LIKEC, LIMIT, LIMITED, LOCAL, LONG, LOOP |
| M | MAP, MAX, MAXLEN, MEMBER, MERGE, MIN, MINUTE, MOD, MODIFY, MONTH, MULTISET |
| N | NAME, NAN, NATIONAL, NATIVE, NCHAR, NEW, NOCOPY, NUMBER_BASE |
| O | OBJECT, OCICOLL, OCIDATETIME, OCIDATE, OCIDURATION, OCIINTERVAL, OCILOBLOCATOR, OCINUMBER, OCIRAW, OCIREFCURSOR, OCIREF, OCIROWID, OCISTRING, OCITYPE, ONLY, OPAQUE, OPEN, OPERATOR, ORACLE, ORADATA, ORGANIZATION, ORLANY, ORLVARY, OTHERS, OUT, OVERRIDING |
| P | PACKAGE, PARALLEL_ENABLE, PARAMETER, PARAMETERS, PARTITION, PASCAL, PIPE, PIPELINED, PRAGMA, PRECISION, PRIVATE |
| R | RAISE, RANGE, RAW, READ, RECORD, REF, REFERENCE, RELIES_ON, REM, REMAINDER, RENAME, RESULT, RESULT_CACHE, RETURN, RETURNING, REVERSE, ROLLBACK, ROW |
| S | SAMPLE, SAVE, SAVEPOINT, SB1, SB2, SB4, SECOND, SEGMENT, SELF, SEPARATE, SEQUENCE, SERIALIZABLE, SET, SHORT, SIZE_T, SOME, SPARSE, SQLCODE, SQLDATA, SQLNAME, SQLSTATE, STANDARD, STATIC, STDDEV, STORED, STRING, STRUCT, STYLE, SUBMULTISET, SUBPARTITION, SUBSTITUTABLE, SUBTYPE, SUM, SYNONYM |
| T | TDO, THE, TIME, TIMESTAMP, TIMEZONE_ABBR, TIMEZONE_HOUR, TIMEZONE_MINUTE, TIMEZONE_REGION, TRAILING, TRANSACTION, TRANSACTIONAL, TRUSTED, TYPE |
| U | UB1, UB2, UB4, UNDER, UNSIGNED, UNTRUSTED, USE, USING |
| V | VALIST, VALUE, VARIABLE, VARIANCE, VARRAY, VARYING, VOID |
| W | WHILE, WORK, WRAPPED, WRITE |
| Y | YEAR |
| Z | ZONE |

# Index

## Symbols


$$PLSQL_LINE inquiry directive, 2-46
$$PLSQL_UNIT inquiry directive, 2-46
% wildcard character, 2-36
%BULK_EXCEPTIONS cursor attribute, 12-17
%BULK_ROWCOUNT cursor attribute, 12-19
%FOUND cursor attribute
  for implicit cursor, 6-7
  for named cursor, 6-14
%ISOPEN cursor attribute
  for implicit cursor, 6-7
  for named cursor, 6-14
%NOTFOUND cursor attribute
  for implicit cursor, 6-8
  for named cursor, 6-15
%ROWCOUNT cursor attribute
  for implicit cursor, 6-8
  for named cursor, 6-16
%ROWTYPE attribute
  column alias and, 6-17
  explicit cursor and, 6-17
  initial value and, 2-17
  overview of, 2-16
  syntax of, 13-116
%TYPE attribute
  initial value and, 2-15
  NOT NULL constraint and, 2-15
  overview of, 2-15
  syntax of, 13-127
_ wildcard character, 2-36


## A


Abstract Data Type (ADT)
  creating, 14-69
  editioned, 14-21
  for use in any schema, 8-22
  what it is, 1-8
accent-insensitive comparison, 2-36
ACCESS_INTO_NULL exception, 11-7
actual parameter
  of explicit cursor, 6-18
  of subprogram
    notation for, 8-12
    passed by reference, 8-7
    passed by value, 8-7
    what it is, 8-6
ADT
  *See* Abstract Data Type (ADT)
AFTER trigger
  what it is, 9-2
  when to use, 9-7
AL16UTF16 encoding, 2-3
alerting an application, 10-16
aliasing, subprogram parameter
  bulk binds and, 12-29
  NOCOPY hint and, 8-27
ALTER FUNCTION statement, 14-3
ALTER LIBRARY statement, 14-6
ALTER PACKAGE statement, 14-8
ALTER PROCEDURE statement, 14-11
ALTER TRIGGER statement, 14-14
ALTER TYPE statement, 14-17
AND operator, 2-29
anonymous block
  AUTHID property and, 8-20
  what it is, 1-6
apostrophe inside string literal, 2-9
architecture of PL/SQL, 1-10
array
  associative
    *See* associative array
  variable-size
    *See* variable-size array
assigment of value
  to variable, 2-22
assignment of value
  to collection, 5-13
  to collection of records, 5-44
  to record, 5-37
assignment statement
  in general, 2-23
  syntax of, 13-3
associative array
  characteristics of, 5-2
  in general, 5-3
  NLS parameters and, 5-4
  subscript range for, 5-12
  syntax of, 13-23
atomic (lexical) unit, 2-3
attribute

%ROWTYPE
column alias and, 6-17
explicit cursor and, 6-17
initial value and, 2-17
overview of, 2-16
syntax of, 13-116
%TYPE
initial value and, 2-15
NOT NULL constraint and, 2-15
overview of, 2-15
syntax of, 13-127
of cursor variable, 6-14
of explicit cursor, 6-14
of implicit cursor
in general, 6-6
native dynamic SQL and, 7-2
AUTHID property, 8-19
autonomous transaction, 6-47
autonomous trigger, 6-51
AUTONOMOUS_TRANSACTION pragma
in compound trigger, 9-17
syntax of, 13-6

## B

bag data structure
*See* nested table
base type, 3-1
basic LOOP statement, 13-16
BEFORE trigger
package variables and, 9-8
what it is, 9-2
when to use, 9-7
BETWEEN operator, 2-37
BFILE data type, 3-22
binary string
LONG RAW, 3-14
RAW, 3-12
BINARY_DOUBLE data type
for computation-intensive programs, 12-33
in general, 3-5
tuning code and, 12-7
BINARY_FLOAT data type
for computation-intensive programs, 12-33
in general, 3-5
tuning code and, 12-7
BINARY_INTEGER data type
*See* PLS_INTEGER data type
bind argument
avoiding SQL injection with, 7-15
placeholder for
*See* placeholder for bind argument
binding variables, 12-10
blank-padding
CHAR and VARCHAR2 strings, 3-10
NCHAR strings, 3-13
BLOB data type, 3-22
block syntax, 13-8
BOOLEAN data type, 3-15
BOOLEAN expression, 2-39
BOOLEAN literal, 2-10
BOOLEAN static expression, 2-49
BOOLEAN variable, 2-25
bounded collection, 5-2
built-in function
*See* SQL function
BULK COLLECT clause
in general, 6-20
tuning and, 12-21
bulk SQL for tuning loop, 12-10
byte string
LONG RAW, 3-14
RAW, 3-12
BYTE unit, 3-9

## C

C subprogram, 8-25
cache, function result, 8-29
call stack, AUTHID property and, 8-20
capture, B-6
cascading triggers, 9-20
CASE expression
searched, 2-41
simple, 2-40
case sensitivity
in character comparison, 2-36
in character literal, 2-9
in database character set, 2-1
in identifier, 2-6
in keyword, D-1
in reserved word, D-1
in string literal, 2-9
LIKE operator and, 2-36
CASE statement
overview of, 4-2
searched, 4-7
simple
compared to IF THEN ELSIF statement, 4-6
in general, 4-7
syntax of, 13-18
CASE_NOT_FOUND exception, 11-7
case-insensitive comparison, 2-36
CHAR data type
compared to VARCHAR2, 3-9
in general, 3-8
CHAR literal, 2-8
CHAR unit, 3-9
character data types, 3-8
character set
database, 2-1
national, 2-3
character string
fixed-length
CHAR, 3-8
VARCHAR2, 3-8
national, 3-12
variable-length
LONG, 3-14
NVARCHAR2, 3-13

RAW, 3-12
VARCHAR2, 3-8
CHARACTER subtype, 3-9
CLOB data type
comparison operator and, 2-34
in general, 3-23
CLOSE statement, 13-21
collating sequence, 2-35
collection
constructor for, 5-10
cursor variable and, 6-35
retrieving query results into, 12-21
subscript range for, 5-12
syntax of, 13-23
what it is, 5-1
collection method
for parameter, 5-22
invalid subscript for, 5-33
syntax of, 13-27
what it is, 5-22
COLLECTION_IS_NULL exception, 11-7
column alias
in cursor FOR LOOP, 6-22
in explicit cursor, 6-17
comment
in general, 2-10
nested, 2-12
syntax of, 13-30
COMMIT statement
FOR UPDATE cursor and, 6-45
in autonomous transaction, 6-51
in general, 6-38
comparison
of collections, 5-18
of records, 5-39
comparison operator
cursor variable and, 6-34
in general, 2-34
compatible data type, 3-27
compilation
conditional, 2-43
for native execution, 12-36
interpreted, 12-39
compilation parameter
displaying value of, 2-47
in general, 1-12
predefined inquiry directive for, 2-46
compiler directive
*See* pragma
compile-time warning
in general, 11-1
tuning code and, 12-8
composite variable, 5-1
compound trigger, 9-14
computation-intensive program, 12-33
concatenation operator (||), 2-26
concurrent transactions, 6-51
conditional compilation, 2-43
conditional compilation directive
error, 2-45
in general, 2-44
inquiry, 2-45
restrictions on, 2-53
selection, 2-44
conditional predicate, 9-24
conditional selection statement, 4-1
conditional trigger, 9-14
Connection Pool, 10-18
constant
declaration
overview of, 2-13
syntax of, 13-32
initial value of, 2-13
predefined, 3-5
static
in DBMS_DB_VERSION package, 2-51
in general, 2-50
constrained subtype, 3-2
constraint, NOT NULL
*See* NOT NULL constraint
constructor
for collection, 5-10
for nested table, 5-10
for variable-size array, 5-11
context of transaction, 6-48
CONTINUE statement
in general, 4-11
syntax of, 13-35
CONTINUE WHEN statement
in general, 4-12
syntax of, 13-35
control statement, 4-1
control token, 2-44
conversion
*See* data type conversion
correlated subquery, 6-25
COUNT method, 5-23
counter, loop, 4-15
CPU overhead, avoiding, 12-4
CREATE FUNCTION statement, 14-32
CREATE LIBRARY statement, 14-40
CREATE PACKAGE statement, 14-42
CREATE TRIGGER statement, 14-53
CREATE TYPE BODY statement, 14-86
CREATE TYPE statement, 14-69
CURRENT OF clause
FOR UPDATE cursor and, 6-45
ROWID pseudocolumn instead of, 6-46
what it is, 6-45
CURRENT_USER, 8-21
CURRVAL pseudocolumn, 6-3
cursor
explicit
*See* explicit cursor
FOR UPDATE
unusable after COMMIT or ROLLBACK, 6-45
what it is, 6-45
implicit
*See* implicit cursor
named, 6-14

nested, 6-35
what it is, 6-6
cursor attribute
DBMS_SQL package and, 7-7
for implicit cursor
in general, 6-6
native dynamic SQL and, 7-2
SQL%BULK_EXCEPTIONS, 12-17
SQL%BULK_ROWCOUNT, 12-19
SQL%FOUND, 6-7
SQL%ISOPEN, 6-7
SQL%NOTFOUND, 6-8
SQL%ROWCOUNT, 6-8
for named cursor
%FOUND, 6-14
%ISOPEN, 6-14
%NOTFOUND, 6-15
%ROWCOUNT, 6-16
in general, 6-14
cursor expression, 6-35
cursor FOR LOOP statement
in general, 6-21
syntax of, 13-37
cursor parameter, 6-18
cursor variable
attributes of, 6-14
declaration syntax of, 13-39
in general, 6-25
syntax of attribute of, 13-93
tuning and, 12-34
CURSOR_ALREADY_OPEN exception, 11-7

## D

d or D in numeric literal, 2-8
data abstraction, 1-7
data type
compatible, 3-27
object
*See* Abstract Data Type (ADT)
of character literal, 2-9
of string literal, 2-9
user-defined
*See* Abstract Data Type (ADT)
what it determines, 3-1
data type conversion
explicit, 3-27
implicit
in general, 3-27
minimizing, 12-6
in general, 3-26
SQL injection and, 7-13
database character set, 2-1
database link, IR, 8-22
Database Resident Connection Pool, 10-18
DATE data type, 3-16
datetime arithmetic, 3-20
datetime data types, 3-15
datetime literal, 2-10
DAY field, 3-15
DBMS_ALERT package, 10-16
DBMS_ASSERT package, 7-16
DBMS_CONNECTION_POOL package, 10-18
DBMS_DB_VERSION package, 2-51
DBMS_OUTPUT package, 10-17
DBMS_PARALLEL_EXECUTE package, 12-50
DBMS_PIPE package, 10-17
DBMS_PREPROCESSOR package, 2-53
DBMS_PROFILE package, 12-9
DBMS_SQL package
in general, 7-6
switching to native dynamic SQL from, 7-7
DBMS_SQL.TO_NUMBER function, 7-8
DBMS_SQL.TO_REFCURSOR function, 7-7
DBMS_TRACE package, 12-10
DBMS_WARNING package, 11-3
dbmsupgin.sql script, 12-38
dbmsupgnv.sql script, 12-38
DDL statement in trigger, 6-51
deadlock
autonomous transaction and, 6-51
implicit rollback and, 6-42
DEC subtype, 3-7
DECIMAL subtype, 3-7
declaration, 2-12
DEFINE
binding category, 12-11
wrap utility and, A-4
DEFINER, 8-21
definer's rights unit
*See* DR unit
DELETE method, 5-30
DELETE statement
PL/SQL extension to, 13-42
*See also* DML statement
DELETING conditional predicate, 9-24
delimiter
in general, 2-4
user-defined, 2-9
dense collection, 5-2
Descriptive Intermediate Attributed Notation for Ada (DIANA), C-1
digit, 2-1
directive
compiler
*See* pragma
error, 2-45
inquiry, 2-45
selection, 2-44
*See also* conditional compilation directive
disabled trigger, 9-3
disabling compile-time warnings, 11-2
DML statement
FORALL statement and, 12-11
implicit cursor for, 6-6
inner capture in, B-8
inside pipelined table function, 12-49
on pipelined table function, 12-49
repeating efficiently, 12-11
what it is, xxxi

dot notation
for collection method, 5-22
for pseudocolumn, 6-4
for qualified name, 2-18
for record field, 5-1
name resolution and, B-5
DOUBLE PRECISION subtype, 3-7
double quotation mark ("), 2-2
DR unit
call stack and, 8-20
dynamic SQL and, 8-20
name resolution and, 8-20
privilege checking and, 8-20
SET ROLE command and, 8-21
static SQL and, 8-20
*See also* AUTHID property
DROP FUNCTION statement, 14-91
DROP LIBRARY statement, 14-93
DROP PACKAGE statement, 14-94
DROP PROCEDURE statement, 14-96
DROP TRIGGER statement, 14-98
DROP TYPE BODY statement, 14-101
DUP_VAL_ON_INDEX exception, 11-8
dynamic SQL
AUTHID property and, 8-20
native
in general, 7-2
switching to DBMS_SQL package from, 7-7
placeholder for bind argument in
EXECUTE IMMEDIATE statement and, 7-2
repeated, 7-5
tuning, 12-34
what it is, 7-1


## E


e or E in numeric literal, 2-8
editioned Abstract Data Type (ADT), 14-21
electronic mail, 10-18
element of collection
reference to, 5-12
what it is, 5-1
e-mail, 10-18
embedded SQL
*See* static SQL
enabled trigger, 9-3
enabling compile-time warnings, 11-2
error directive, 2-45
error handling, 11-1
error-reporting function
SQLCODE, 13-124
SQLERRM, 13-125
escape character, 2-37
escaped identifier, 6-3
evaluation order, 2-27
evolution of type, 14-17
exception
avoiding, 11-6
declaration syntax of, 13-45
for collection, 5-31
handling
guidelines for, 11-6
overview of, 11-4
in FORALL statement
effect on rollback, 12-16
unhandled, 12-16
predefined
in general, 11-7
redefining, 11-12
progpagation of, 11-13
raised in trigger body, 9-24
raising, 11-13
reraising, 11-15
unhandled
in FORALL statement, 12-16
in general, 11-22
user-defined, 11-9
exception handler
for FORALL statement, 12-17
in general, 11-16
in pipelined table function, 12-50
syntax of, 13-47
EXCEPTION_INIT pragma, 13-43
EXECUTE IMMEDIATE statement
in general, 7-2
syntax of, 13-49
tuning and, 12-34
EXISTS method, 5-23
EXIT statement
in general, 4-10
syntax of, 13-52
EXIT WHEN statement
in general, 4-10
syntax of, 13-52
exiting a loop, 4-9
explicit cursor
calculated column in, 6-17
declaration syntax of, 13-54
expression in, 6-17
in cursor FOR LOOP statement, 6-22
in general, 6-6
in package, 10-19
syntax of attribute of, 13-93
explicit data type conversion, 3-27
explicit format model, 7-17
expression
as initial parameter value, 8-11
BOOLEAN, 2-39
cursor, 6-35
in explicit cursor, 6-17
in general, 2-25
SQL function in PL/SQL, 2-42
static, 2-48
syntax of, 13-58
EXTEND method, 5-27
external subprogram, 8-25


## F


f or F in numeric literal, 2-8

FETCH statement
across COMMIT, 6-46
LIMIT clause of, 12-25
syntax of, 13-67
field
of datetime or interval data type, 3-15
of record, 5-1
FIRST method, 5-24
fixed-length character string
CHAR, 3-8
NCHAR, 3-13
FLOAT subtype, 3-7
FOR EACH ROW trigger, 9-13
FOR LOOP statement
in general, 4-14
index of, 4-15
syntax of, 13-70
FOR UPDATE cursor
unusable after COMMIT or ROLLBACK, 6-45
what it is, 6-45
FORALL statement
counting rows affected by, 12-19
exception in
effect on rollback, 12-16
handling, 12-17
unhandled, 12-16
in general, 12-11
syntax of, 13-73
tuning code and, 12-11
formal parameter
of cursor, 6-18
of subprogram, 8-6
format model, 7-17
forward declaration of subprogram
for nested subprogram, 8-6
in general, 8-1
function
built-in
*See* SQL function
declaration syntax of, 13-76
error-reporting
SQLCODE, 13-124
SQLERRM, 13-125
invoking, 8-2
options for, 8-4
pipelined table, 12-40
SQL
*See* SQL function
structure of, 8-4
tuning, 12-4
function call, 8-2
function result cache, 8-29

## G

global identifier, 2-18
GOTO statement
exception handler and, 11-18
in general, 4-21
restrictions on, 13-80
syntax of, 13-80

## H

hash table
*See* associative array
hiding PL/SQL source code
*See* wrapping PL/SQL source code
host variable as cursor variable, 6-33
HOUR field, 3-15
HTF package, 10-18
HTML tag generation, 10-18
HTP package, 10-18
HTTP callouts, 10-18

## I

identifier
escaped, 6-3
global, 2-18
in general, 2-5
in static SQL, 6-2
local, 2-18
predefined, 2-6
reference to, 2-17
scope of, 2-18
user-defined
collecting data about, 12-8
ordinary, 2-7
quoted, 2-7
visibility of, 2-18
IDL, C-1
IF statement
IF THEN ELSE form, 4-3
IF THEN ELSIF form
compared to nested IF THEN ELSE statements, 4-6
compared to simple CASE statement, 4-6
in general, 4-5
IF THEN form, 4-2
nested
compared to IF THEN ELSIF form, 4-6
example of, 4-4
overview of, 4-1
syntax of, 13-82
IMP statement, 9-6
implicit cursor
cursor expression with, 6-36
declaration syntax of, 13-84
dynamic SQL and, 7-6
in cursor FOR LOOP statement, 6-21
in general, 6-6
implicit data type conversion
in general, 3-27
minimizing, 12-6
implicit rollback, 6-42
IN operator, 2-38
IN OUT parameter mode, 8-9
IN parameter mode, 8-8
in-bind, 12-11

independent (autonomous) transaction, 6-47
index of FOR LOOP statement, 4-15
index-by table
  *See* associative array
infinite loop, 4-10
INFORMATIONAL compile-time warning, 11-2
initial value
  %ROWTYPE attribute and, 2-17
  %TYPE attribute and, 2-15
  NOT NULL constraint and, 2-14
  of collection, 5-10
  of constant, 2-13
  of cursor parameter, 6-18
  of subprogram parameter, 8-10
  of variable, 2-13
initialization parameter, 1-12
inline LOB locator, 3-21
INLINE pragma, 13-87
inlining, subprogram, 12-2
input, 1-6
inquiry directive, 2-45
INSERT statement
  inserting record with
    in general, 5-40
    restrictions on, 5-43
  PL/SQL extension to, 13-89
  *See also* DML statement
INSERTING conditional predicate, 9-24
instance method, IR, 8-24
INSTEAD OF trigger, 9-8
INT subtype, 3-7
integer literal, 2-8
INTEGER subtype, 3-7
Interface Definition Language (IDL), C-1
interpreted compilation, 12-39
interval arithmetic, 3-20
interval data types, 3-15
INTERVAL DAY TO SECOND data type, 3-20
interval literal, 2-10
INTERVAL YEAR TO MONTH data type, 3-19
INVALID_CURSOR exception
  handling, 6-33
  in general, 11-8
INVALID_NUMBER exception, 11-8
invoker's rights unit
  *See* IR unit
IR unit
  Abstract Data Type (ADT), 8-22
  call stack and, 8-20
  database link, 8-22
  dynamic SQL and, 8-20
  instance method, 8-24
  name resolution and, 8-20
  overriding name resolution in, 8-21
  privilege checking and, 8-20
  run by trigger, 8-22
  run in view, 8-22
  static SQL and, 8-20
  template objects in, 8-21
  *See also* AUTHID property
IS NOT NULL operator, 2-36
IS NULL operator, 2-36
isolation level of transaction, 6-49


## J


Java subprogram, 8-25


## K


key-value pair, 5-3
keywords, D-1


## L


labeled LOOP statements, 4-11
large object (LOB) data types, 3-21
LAST method, 5-24
leading blanks in string literals, 2-9
LENGTHB function, 2-3
letter, 2-1
LEVEL pseudocolumn, 6-5
lexical unit, 2-3
LIKE operator, 2-36
LIMIT clause of bulk FETCH statement, 12-25
LIMIT method, 5-24
literal
  syntax of, 13-91
  types of, 2-8
LOB data types, 3-21
LOB locator, 3-21
local identifier, 2-18
lock mode, 6-44
LOCK TABLE statement, 6-44
locking
  overriding default, 6-44
  result set row, 6-44
  table, 6-44
logical operator, 2-29
logical rowid, 3-14
LOGIN_DENIED exception, 11-8
LONG data type, 3-14
LONG RAW data type, 3-14
loop counter, 4-15
LOOP statement
  exiting, 4-9
  kinds of, 4-9
  labeling, 4-9
LOOP statements
  bulk SQL for, 12-10
  labeled, 4-11
  nested, 4-11
  optimizing, 12-5
LOOP UNTIL structure, 4-14


## M


manageability, 1-3
membership test, 2-38
memory overhead, avoiding, 12-8
Method 4, 7-7

method, collection
    *See* collection method
MINUTE field, 3-15
mixed parameter notation, 8-12
mode
    lock, 6-44
    parameter, 8-8
MONTH field, 3-15
multidimensional collection, 5-20
multiline comment, 2-11
multiple data transformations, 12-40
mutating table, 9-28

## N

name
    qualified, 2-18
    qualified remote, 2-18
    remote, 2-18
    simple, 2-17
    *See also* identifier
name resolution
    AUTHID property and, 8-20
    in static SQL, 6-3
    overriding in IR unit, 8-21
    PL/SQL and SQL differences, B-5
    what it is, B-1
named cursor
    syntax of attribute of, 13-93
    what it is, 6-14
    *See also* explicit cursor *and* cursor variable
named parameter notation, 8-12
national character set, 2-3
national character string, 3-12
native dynamic SQL
    in general, 7-2
    switching to DBMS_SQL package from, 7-7
native execution, compilation for, 12-36
NATURAL subtype, 3-3
NATURALN subtype, 3-3
NCHAR data type, 3-13
NCLOB data type, 3-23
nested comment, 2-12
nested cursor, 6-35
nested IF statement
    compared to IF THEN ELSIF form, 4-6
    example of, 4-4
nested LOOP statements, 4-11
nested record, 5-35
nested subprogram
    forward declaration for, 8-6
    what it is, 8-1
nested table
    characteristics of, 5-2
    constructor for, 5-10
    in general, 5-5
    INSTEAD OF trigger and, 9-9
    subscript range for, 5-12
    syntax of, 13-23
nested transaction, 6-48
new features, xxxiii
NEW pseudorecord, 9-2
NEXT method, 5-25
NEXTVAL pseudocolumn, 6-3
NLS parameters
    associative array and, 5-4
    SQL injection and, 7-13
NLS_LENGTH_SEMANTICS parameter, 3-9
NLS_SORT parameter, 2-36
NO_DATA_FOUND exception, 11-8
NOCOPY hint
    effect on OUT parameter, 8-9
    for tuning subprogram call, 12-34
    in general, 8-7
    restrictions on, 12-36
    subprogram parameter aliasing and, 8-27
NOT NULL constraint
    %TYPE attribute and, 2-15
    EXTEND method and, 5-27
    on collection element, 5-10
    on constant, 2-14
    on cursor parameter, 6-10
    on PL/SQL variable, 2-14
NOT operator, 2-31
NOT_LOGGED_ON exception, 11-8
NULL statement
    syntax of, 13-96
    uses for, 4-23
null string, 2-9
NULL value
    comparison operator and, 2-34
    concatenation operator and, 2-26
    for $$PLSQL_UNIT inquiry directive, 2-46
    for unresolvable inquiry directive, 2-48
    in control statement, 2-32
    IN operator and, 2-38
    in set, 2-38
    simple CASE expression and, 2-40
    simple CASE statement and, 4-7
NUMBER data type, 3-6
numeric data types, 3-2
numeric literal, 2-8
NUMERIC subtype, 3-7
NVARCHAR2 data type, 3-13

## O

obfuscating PL/SQL source code
    *See* wrapping PL/SQL source code
object type
    *See* Abstract Data Type (ADT)
OBJECT_VALUE pseudocolumn, 9-25
OCI
    associative array and, 5-4
    cursor variable and, 6-34
OLD pseudorecord, 9-2
OPEN FOR statement, 13-98
OPEN statement, 13-97
operating system text file I/O, 10-18
operation, 2-27

operator
  comparison
    cursor variable and, 6-34
    in general, 2-34
  logical, 2-29
  relational, 2-34
operator precedence, 2-27
optimizer, 12-1
OR operator, 2-30
Oracle Call Interface (OCI)
  associative array and, 5-4
  cursor variable and, 6-34
Oracle RAC environment, result caches in, 8-38
OUT parameter mode, 8-8
out-bind, 12-11
out-of-line LOB locator, 3-21
output, 1-6
overhead, avoiding
  CPU, 12-4
  memory, 12-8
overloaded subprogram
  in general, 8-13
  PLS_INTEGER and, 3-4
  SIMPLE_INTEGER and, 3-4


## P


package
  advantages of, 10-3
  body of, 10-10
  contents of, 10-2
  guidelines for writing, 10-18
  initialization of, 10-2
  of static constants, 2-50
  pinning in shared memory pool, 12-8
  private items in, 10-15
  product-specific, 10-16
  public items in, 10-15
  referencing contents of, 10-9
  serially reusable, 10-3
  specification of, 10-8
  STANDARD, 10-16
  state of, 10-4
  tuning code and, 12-8
  what it is, 10-1
packaged subprogram, 8-1
parallel updates for large tables, 12-50
parameter
  compilation
    displaying value of, 2-47
    in general, 1-12
    predefined inquiry directive for, 2-46
  cursor, 6-18
  initialization, 1-12
  subprogram
    *See* subprogram parameter
parameter mode, 8-8
parentheses
  nested, 2-27
  to control evaluation order, 2-27
  to improve readability, 2-28
pattern matching, 2-36
percent sign (%) wildcard character, 2-36
PERFORMANCE compile-time warning, 11-2
physical rowid, 3-14
pipe, 10-17
PIPE ROW statement, 12-44
pipelined table function, 12-40
placeholder for bind argument
  in conditional compilation directive, 2-54
  in dynamic SQL
    EXECUTE IMMEDIATE statement and, 7-2
    repeated, 7-5
  in static SQL
    in general, 6-2
    OPEN FOR statement and, 6-30
PLS_INTEGER data type
  for computation-intensive programs, 12-33
  in general, 3-3
  tuning code and, 12-7
PLS_INTEGER static expression, 2-48
PL/Scope tool, 12-8
PL/SQL architecture, 1-10
PL/SQL block syntax, 13-8
PL/SQL engine, 1-11
PL/SQL function result cache, 8-29
PL/SQL language
  advantages of, 1-1
  character sets of, 2-1
  high performance of, 1-2
  high productivity with, 1-2
  lexical units of, 2-3
  limits of, C-1
  main features of, 1-4
  manageability and, 1-3
  new features of, xxxiii
  portability of, 1-3
  program limits of, C-1
  scalability of, 1-3
  security and, 1-3
  SQL integration in, 1-1
  syntax of elements of, 13-1
PL/SQL unit
  stored
    SQL statements for, 14-1
    what it is, 1-13
  what it is, 1-11
PLSQL_CCFLAGS compilation parameter, 2-47
PLSQL_OPTIMIZE_LEVEL compilation parameter, 12-1
PLSQL_WARNINGS compilation parameter
  displaying value of, 11-3
  setting value of, 11-2
portability, 1-3
positional parameter notation, 8-12
POSITIVE subtype, 3-3
POSITIVEN subtype, 3-3
post-processed source text, 2-53
pragma
  AUTONOMOUS_TRANSACTION

in compound trigger, 9-17
syntax of, 13-6
EXCEPTION_INIT, 13-43
INLINE, 13-87
RESTRICT_REFERENCES, 13-109
SERIALLY_REUSABLE, 13-122
what it is, 2-43
precedence, operator, 2-27
precision in NUMBER data type, 3-7
predefined constant, 3-5
predefined data type, 3-1
predefined exception
in general, 11-7
redefining, 11-12
predefined identifier, 2-6
predefined inquiry directive, 2-46
preprocessor control token, 2-44
PRIOR method, 5-25
privilege checking and AUTHID property, 8-20
procedure
declaration syntax of, 13-103
invoking, 8-2
structure of, 8-4
product-specific package, 10-16
Profiler API, 12-9
profiling and tracing programs, 12-9
program limits, C-1
PROGRAM_ERROR exception, 11-8
pseudocolumn
OBJECT_VALUE, 9-25
SQL, 6-3
pseudoinstruction
*See* pragma
pseudorecord, 9-2
punctuation character between identifiers, 2-6
purity rules, 8-27

## Q

qualified name, 2-18
qualified remote name, 2-18
query processing
in general, 6-19
multiple-row dynamic query, 7-4
quotation mark, single or double, 2-2
quoted identifier, 2-7

## R

RAISE statement, 13-105
RAISE_APPLICATION_ERROR procedure, 11-11
range test, 2-37
RAW data type, 3-12
read-only transaction, 6-43
read-write transaction, 6-43
real literal, 2-8
REAL subtype, 3-7
record
definition and declaration of, 5-34
nested, 5-35
what it is, 5-1
recursive subprogram
in general, 8-25
result-cached, 8-34
recursive trigger, 9-4
REF CURSOR
*See* cursor variable
REF CURSOR type, 6-26
relational operator, 2-34
RELEASE constant, 2-51
remote name, 2-18
REPEAT UNTIL structure, 4-14
reserved preprocessor control token, 2-44
reserved words, D-1
Resident Connection Pool, 10-18
RESTRICT_REFERENCES pragma, 13-109
result cache, 8-29
RESULT_CACHE clause, 8-30
RETURN clause, 8-4
RETURN statement
in general, 8-4
syntax of, 13-111
RETURNING INTO clause, 13-113
REUSE SETTINGS clause, 1-13
rollback
deadlock and, 6-42
FORALL statement and, 12-16
ROLLBACK statement
FOR UPDATE cursor and, 6-45
in autonomous transaction, 6-51
in general, 6-39
row trigger
body of, 9-21
column values accessible to, 9-22
rowid, 3-14
ROWID pseudocolumn
in static SQL, 6-5
instead of CURRENT OF clause, 6-46
what it is, 3-14
ROWIDTOCHAR function, 6-6
row-level trigger, 9-13
pseudorecords and, 9-2
what it is, 9-2
ROWNUM pseudocolumn, 6-6
ROWTYPE_MISMATCH exception, 11-8
run-time error
*See* exception

## S

SAVEPOINT statement
in autonomous transaction, 6-51
in general, 6-41
scalability, 1-3
scalar data types, 3-2
scale in NUMBER data type, 3-7
scientific notation, 2-8
scope of identifier, 2-18
searched CASE expression, 2-41
searched CASE statement

explanation of, 4-7
syntax of, 13-18
SECOND field, 3-15
security, 1-3
SELECT FOR UPDATE statement, 6-44
SELECT INTO statement
implicit cursor for, 6-6
introduction to, 2-23
SQL%NOTFOUND attribute and, 6-8
SQL%ROWCOUNT attribute and, 6-9
syntax of, 13-118
selection directive, 2-44
selector
in simple CASE expression, 2-40
in simple CASE statement, 4-7
SELF_IS_NULL exception, 11-8
sending electronic mail, 10-18
sequence, 6-3
sequential control statement, 4-20
serially reusable package, 10-3
SERIALLY_REUSABLE pragma, 13-122
session cursor
what it is, 6-6
set data structure
*See* nested table
set membership test, 2-38
SET ROLE command and AUTHID property, 8-21
SET TRANSACTION statement, 6-43
SEVERE compile-time warning, 11-2
short-circuit evaluation
how it works, 2-33
tuning code and, 12-6
side effects of subprogram, 8-27
SIGNTYPE subtype, 3-3
simple CASE expression, 2-40
simple CASE statement
compared to IF THEN ELSIF statement, 4-6
explanation of, 4-7
syntax of, 13-18
Simple Mail Transfer Protocol (SMTP), 10-18
simple name, 2-17
simple trigger, 9-2
SIMPLE_DOUBLE subtype
in general, 3-6
tuning code and, 12-7
SIMPLE_FLOAT subtype
in general, 3-6
tuning code and, 12-7
SIMPLE_INTEGER subtype
in general, 3-3
tuning code and, 12-7
single quotation mark
inside string literal, 2-9
symbol, 2-2
single-line comment, 2-10
SMALLINT subtype, 3-7
SMTP (Simple Mail Transfer Protocol), 10-18
space character between identifiers, 2-6
sparse collection, 5-2
SQL
dynamic
*See* dynamic SQL
static
*See* static SQL
SQL cursor
*See* implicit cursor
SQL function
highly optimized, 12-6
in PL/SQL expression, 2-42
SQL injection, 7-9
SQL integration in PL/SQL, 1-1
SQL pseudocolumn, 6-3
SQL statement
for stored PL/SQL units, 14-1
running in trigger, 9-3
tuning, 12-4
*See also* anonymous block
SQL%BULK_EXCEPTIONS cursor attribute, 12-17
SQL%BULK_ROWCOUNT cursor attribute, 12-19
SQL%FOUND cursor attribute, 6-7
SQL%NOTFOUND cursor attribute, 6-8
SQL%ROWCOUNT cursor attribute, 6-8
SQL*Loader and triggers, 9-6
SQLCODE function, 13-124
SQLERRM function, 13-125
SQLJ object type, creating, 14-69
standalone stored subprogram, 8-1
STANDARD package, 10-16
state
of package, 10-4
of trigger, 9-3
statement injection, 7-11
statement modification, 7-10
statement-level trigger, 9-2, 9-13
static constant
in DBMS_DB_VERSION package, 2-51
in general, 2-50
static expression, 2-48
static SQL
AUTHID property and, 8-20
name resolution in, 6-3
placeholder for bind argument in
in general, 6-2
OPEN FOR statement and, 6-30
PL/SQL identifier in, 6-2
what it is, 6-1
STORAGE_ERROR exception
in general, 11-8
recursion and, 8-25
store table, 5-6
stored PL/SQL unit, 1-13
string data type
binary or byte
LONG RAW, 3-14
RAW, 3-12
character
*See* character string
national character, 3-12
string literal
in general, 2-9

null, 2-9
syntax of, 13-91
zero-length, 2-9
STRING subtype, 3-9
strong REF CURSOR type, 6-26
subnormal range, 3-6
subprogram
C, 8-25
external, 8-25
inlining, 12-2
invoking
*See* subprogram call
Java, 8-25
nested
forward declaration for, 8-6
what it is, 8-1
overloaded
in general, 8-13
PLS_INTEGER and, 3-4
SIMPLE_INTEGER and, 3-4
overview of, 8-1
parameter
*See* subprogram parameter
parts of, 8-3
purity rules for, 8-27
reasons to use, 8-2
recursive
in general, 8-25
result-cached, 8-34
side effects of, 8-27
subprogram call
optimization of, 12-2
resolution of, 8-17
syntax of, 8-2
tuning, 12-34
subprogram parameter
actual
notation for, 8-12
passed by reference, 8-7
passed by value, 8-7
what it is, 8-6
aliasing
bulk binds and, 12-29
NOCOPY hint and, 8-27
cursor expression as, 6-36
cursor variable as, 6-28
formal, 8-6
initial value of, 8-10
mode of, 8-8
record as, 5-36
set of rows as, 6-36
subquery
correlated, 6-25
in cursor declaration, 6-23
in FROM clause, 6-24
in general, 6-23
SUBSCRIPT_BEYOND_COUNT exception, 11-8
SUBSCRIPT_OUTSIDE_LIMIT exception, 11-8
subtype
constrained, 3-2
of BINARY_DOUBLE data type, 3-6
of BINARY_FLOAT data type, 3-6
of datetime date type, 3-21
of interval data type, 3-21
of NUMBER data type, 3-7
of PLS_INTEGER data type, 3-3
unconstrained, 3-2
user-defined, 3-23
what it is, 3-1
symbols in database character set, 2-1
synonym, 2-18
syntax of PL/SQL language elements, 13-1
SYS_INVALID_ROWID exception, 11-8
SYS_REFCURSOR type, 6-26
system trigger
restrictions on, 9-30
what it is, 9-2


## T


table
hash
*See* associative array
index-by
*See* associative array
mutating, 9-28
nested
characteristics of, 5-2
constructor for, 5-10
in general, 5-5
INSTEAD OF trigger and, 9-9
subscript range for, 5-12
store, 5-6
unordered
*See* associative array
updating large in parallel, 12-50
table function, pipelined, 12-40
TCL statement
in general, 6-37
in trigger, 6-51
TIMEOUT_ON_RESOURCE exception, 11-8
TIMESTAMP data type, 3-16
TIMESTAMP WITH LOCAL TIME ZONE data type, 3-18
TIMESTAMP WITH TIME ZONE data type,, 3-18
TIMEZONE_ABBR field, 3-16
TIMEZONE_HOUR field, 3-15
TIMEZONE_MINUTE field, 3-15
TIMEZONE_REGION field, 3-15
timing point of trigger, 9-2
TO_NUMBER function, 7-8
TO_REFCURSOR function, 7-7
TOO_MANY_ROWS exception, 11-9
Trace API, 12-10
trace file, 11-1
tracing and profiling programs, 12-9
trailing blanks in string literals, 2-9
transaction
autonomous, 6-47
ending

with COMMIT statement, 6-38
with ROLLBACK statement, 6-39
independent (autonomous), 6-47
nested, 6-48
read-only, 6-43
read-write, 6-43
SQL%ROWCOUNT cusor attribute and, 6-9
Transaction Control Language
*See* TCL statement
TRANSACTIONS initialization parameter, 6-51
trigger
as security mechanism, 9-4
AUTHID property and, 8-20
autonomous, 6-51
body of, 9-21
cascading, 9-20
compilation of, 9-31
compound, 9-14
conditional, 9-14
creating, 9-5
DDL statement in, 6-51
debugging, 9-32
designing, 9-4
disabled, 9-3
disabling, 9-32
enabled, 9-3
enabling, 9-32
firing order, 9-20
for publishing database event, 9-50
IR subprogram run by, 8-22
level of, 9-13
overview of, 9-1
pseudorecords and, 9-2
recursive, 9-4
replacing, 9-32
state of, 9-3
TCL statement in, 6-51
timing point of, 9-2
types of, 9-2
uses of, 9-3
views for information about, 9-33
triggering statement
in general, 9-6
of compound trigger, 9-16
TRIM method, 5-28
tri-state logic, 2-29
tuning PL/SQL code, 12-3
type
*See* data type


## U


unbounded collection, 5-2
unconstrained subtype, 3-2
underscore (_) wildcard character, 2-36
unhandled exception
in FORALL statement, 12-16
in general, 11-22
Unicode, 2-3
universal rowid, 3-14
unordered table
*See* associative array
updatable view, 9-8
UPDATE statement
PL/SQL extensions to, 13-129
trigger on, 9-7
with values in record
in general, 5-42
restrictions on, 5-43
*See also* DML statement
UPDATING conditional predicate, 9-24
updating large tables in parallel, 12-50
UROWID data type
in static SQL, 6-5
what it is, 3-14
USER_DUMP_DEST initialization parameter, 11-1
user-defined exception, 11-9
user-defined identifier
collecting data about, 12-8
ordinary, 2-7
quoted, 2-7
user-defined subtype, 3-23
user-defined type
*See* Abstract Data Type (ADT)
UTF8 encoding, 2-3
UTL_FILE package, 10-18
UTL_HTTP package, 10-18
UTL_SMTP package, 10-18
utlrp.sql script, 12-40


## V


V$RESERVED_WORDS view, D-1
validation check for avoiding SQL injection, 7-16
VALUE_ERROR exception, 11-9
VARCHAR subtype, 3-9
VARCHAR2 data type
compared to CHAR, 3-9
in general, 3-8
tuning code and, 12-8
VARCHAR2 static expression, 2-49
variable
assigning value to, 2-22
binding of, 12-10
BOOLEAN, 2-25
composite, 5-1
cursor
*See* cursor variable
declaration of
overview of, 2-12
syntax of, 13-131
host, cursor variable as, 6-33
initial value of, 2-13
with undefined value, 6-2
variable-length character string
LONG, 3-14
NVARCHAR2, 3-13
RAW, 3-12
VARCHAR2, 3-8
variable-size array

characteristics of, 5-2
constructor for, 5-11
in general, 5-6
subscript range for, 5-12
syntax of, 13-23
varray
*See* variable-size array
VERSION constant, 2-51
view
AUTHID property and, 8-20
IR subprogram run in, 8-22
updatable, 9-8
visibility
of identifier, 2-18
of transaction, 6-48

## W

warning, compile-time
in general, 11-1
tuning code and, 12-8
weak REF CURSOR type, 6-26
WHILE LOOP statement
in general, 4-13
syntax of, 13-133
whitespace character
between lexical units, 2-4
in database character set, 2-1
inside lexical unit, 2-3
wildcard character, 2-36
wrap utility, A-3
wrapping PL/SQL source code
in general, A-1
inquiry directives and, 2-48

## Y

YEAR field, 3-15

## Z

ZERO_DIVIDE exception, 11-9
zero-length string, 2-9