Oracle® Database

PL/SQL Language Reference

11*g* Release 2 (11.2)

E17126-04

October 2010

ORACLE®

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

E17126-04

Copyright © 1996, 2010, 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, K. Rich, 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 xxvii

Audience xxvii
Documentation Accessibility xxvii
Related Documents xxviii
Conventions xxviii
Syntax Descriptions xxix

What's New in PL/SQL? xxxi

PL/SQL Feature for Oracle Database 11*g* Release 2 (11.2.0.2) xxxi
PL/SQL Features for Oracle Database 11*g* Release 2 (11.2.0.1) xxxi
PL/SQL Features for Oracle Database 11*g* Release 1 (11.1) xxxiii

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
Support for Object-Oriented Programming 1-3
Support for Developing Web Applications 1-3
Support for Developing Server Pages 1-4
Main Features of PL/SQL 1-4
Error Handling 1-4
Blocks 1-5
Variables and Constants 1-5
Subprograms 1-6
Packages 1-6
Triggers 1-6
Input and Output 1-6
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
Conditional Compilation 1-9
Processing a Query Result Set One Row at a Time 1-9
Architecture of PL/SQL 1-10
PL/SQL Engine 1-10
PL/SQL Units and Compilation Parameters 1-10

2 PL/SQL Language Fundamentals

Character Sets 2-1
Database Character Set 2-1
National Character Set 2-3
Lexical Units 2-3
Delimiters 2-3
Identifiers 2-4
Reserved Words and Keywords 2-5
Predefined Identifiers 2-5
User-Defined Identifiers 2-5
Literals 2-8
Comments 2-9
Single-Line Comments 2-9
Multiline Comments 2-10
Whitespace Characters Between Lexical Units 2-11
Declarations 2-11
Variable Declarations 2-12
Constant Declarations 2-12
Initial Values of Variables and Constants 2-13
NOT NULL Constraint 2-13
%TYPE Attribute 2-14
References to Identifiers 2-15
Scope and Visibility of Identifiers 2-16
Assigning Values to Variables 2-20
Assigning Values to Variables with the Assignment Statement 2-21
Assigning Values to Variables with the SELECT INTO Statement 2-22
Assigning Values to Variables as Parameters of a Subprogram 2-22
Assigning Values to BOOLEAN Variables 2-23
Expressions 2-23
Concatenation Operator 2-24
Operator Precedence 2-24
Logical Operators 2-27
Short-Circuit Evaluation 2-31
Comparison Operators 2-32
Relational Operators 2-32
IS [NOT] NULL Operator 2-34
LIKE Operator 2-34
BETWEEN Operator 2-35

IN Operator ... 2-36
BOOLEAN Expressions ... 2-37
CASE Expressions ... 2-37
Simple CASE Expression ... 2-38
Searched CASE Expression ... 2-39
SQL Functions in PL/SQL Expressions ... 2-40
Error-Reporting Functions ... 2-41
Pragmas ... 2-41
Conditional Compilation ... 2-41
How Conditional Compilation Works ... 2-42
Preprocessor Control Tokens ... 2-42
Selection Directives ... 2-42
Error Directives ... 2-43
Inquiry Directives ... 2-43
Static Expressions ... 2-46
Conditional Compilation Examples ... 2-49
Retrieving and Printing Post-Processed Source Text ... 2-51
Conditional Compilation Directive Restrictions ... 2-52

3 PL/SQL Data Types

SQL Data Types ... 3-2
Different Maximum Sizes ... 3-2
Additional PL/SQL Constants for BINARY_FLOAT and BINARY_DOUBLE ... 3-2
Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE ... 3-3
CHAR and VARCHAR2 Variables ... 3-3
Assigning or Inserting Too-Long Values ... 3-3
Declaring Variables for Multibyte Characters ... 3-4
Differences Between CHAR and VARCHAR2 Data Types ... 3-5
LONG and LONG RAW Variables ... 3-6
ROWID and UROWID Variables ... 3-7
BOOLEAN Data Type ... 3-7
PLS_INTEGER and BINARY_INTEGER Data Types ... 3-8
Preventing PLS_INTEGER Overflow ... 3-8
Predefined PLS_INTEGER Subtypes ... 3-9
SIMPLE_INTEGER Subtype of PLS_INTEGER ... 3-10
SIMPLE_INTEGER Overflow Semantics ... 3-10
Expressions with Both SIMPLE_INTEGER and Other Operands ... 3-11
Integer Literals in SIMPLE_INTEGER Range ... 3-11
User-Defined PL/SQL Subtypes ... 3-11
Unconstrained Subtypes ... 3-12
Constrained Subtypes ... 3-12
Subtypes with Base Types in Same Data Type Family ... 3-14

4 PL/SQL Control Statements

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 4-6
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
FOR LOOP Statement 4-13
FOR LOOP Index 4-15
Lower Bound and Upper Bound 4-17
EXIT WHEN or CONTINUE WHEN Statement in FOR LOOP Statement 4-18
WHILE LOOP Statement 4-20
Sequential Control Statements 4-20
GOTO Statement 4-21
NULL Statement 4-23

5 PL/SQL Collections and Records

Collection Types 5-2
Associative Arrays 5-4
NLS Parameter Values Affect Associative Arrays Indexed by String 5-6
Changing NLS Parameter Values After Populating Associative Arrays 5-6
Subscripts of Data Types Other Than VARCHAR2 5-6
Passing Associative Arrays to Remote Databases 5-7
Appropriate Uses for Associative Arrays 5-7
Varrays (Variable-Size Arrays) 5-8
Appropriate Uses for Varrays 5-9
Nested Tables 5-10
Important Differences Between Nested Tables and Arrays 5-12
Appropriate Uses for Nested Tables 5-12
Collection Constructors 5-13
Assigning Values to Collection Variables 5-14
Data Type Compatibility 5-14
Assigning Null Values to Varray or Nested Table Variables 5-15
Assigning Set Operation Results to Nested Table Variables 5-16
Multidimensional Collections 5-17
Collection Comparisons 5-18
Comparing Varray and Nested Table Variables to NULL 5-19
Comparing Nested Tables for Equality and Inequality 5-19
Comparing Nested Tables with SQL Multiset Conditions 5-20
Collection Methods 5-21
DELETE Collection Method 5-22
TRIM Collection Method 5-25
EXTEND Collection Method 5-26
EXISTS Collection Method 5-28

FIRST and LAST Collection Methods 5-28
FIRST and LAST Methods for Associative Array 5-29
FIRST and LAST Methods for Varray 5-30
FIRST and LAST Methods for Nested Table 5-31
COUNT Collection Method 5-32
COUNT Method for Varray 5-33
COUNT Method for Nested Table 5-33
LIMIT Collection Method 5-34
PRIOR and NEXT Collection Methods 5-35
Collection Types Defined in Package Specifications 5-37
Record Variables 5-38
Initial Values of Record Variables 5-39
RECORD Types 5-39
%ROWTYPE Attribute 5-42
Record Variable that Always Represents Full Row 5-42
Record Variable that Can Represent Partial Row 5-43
Assigning Values to Record Variables 5-44
Assigning One Record Variable to Another 5-45
Assigning Full or Partial Rows to Record Variables 5-47
SELECT INTO Statement for Assigning Row to Record Variable 5-47
FETCH Statement for Assigning Row to Record Variable 5-48
SQL Statements that Return Rows in PL/SQL Record Variables 5-49
Record Comparisons 5-49
Inserting Records into Tables 5-50
Updating Rows with Records 5-51
Restrictions on Record Inserts and Updates 5-52

6 PL/SQL Static SQL

Description of Static SQL 6-1
Statements 6-1
Pseudocolumns 6-3
CURRVAL and NEXTVAL in PL/SQL 6-3
Cursors 6-5
Implicit Cursors 6-5
SQL%ISOPEN Attribute: Is the Cursor Open? 6-6
SQL%FOUND Attribute: Were Any Rows Affected? 6-6
SQL%NOTFOUND Attribute: Were No Rows Affected? 6-7
SQL%ROWCOUNT Attribute: How Many Rows Were Affected? 6-7
Explicit Cursors 6-8
Declaring and Defining Explicit Cursors 6-8
Opening and Closing Explicit Cursors 6-9
Fetching Data with Explicit Cursors 6-10
Variables in Explicit Cursor Queries 6-12
When Explicit Cursor Queries Need Column Aliases 6-14
Explicit Cursors that Accept Parameters 6-15
Explicit Cursor Attributes 6-19
Query Result Set Processing 6-22

Query Result Set Processing With SELECT INTO Statements 6-23
Single-Row Result Sets 6-23
Large Multiple-Row Result Sets 6-23
Query Result Set Processing With Cursor FOR LOOP Statements 6-24
Query Result Set Processing With Explicit Cursors, OPEN, FETCH, and CLOSE 6-26
Query Result Set Processing with Subqueries 6-26
Cursor Variables 6-28
Creating Cursor Variables 6-28
Opening and Closing Cursor Variables 6-30
Fetching Data with Cursor Variables 6-30
Assigning Values to Cursor Variables 6-32
Variables in Cursor Variable Queries 6-33
Cursor Variable Attributes 6-35
Cursor Variables as Subprogram Parameters 6-35
Cursor Variables as Host Variables 6-37
CURSOR Expressions 6-38
Transaction Processing and Control 6-39
COMMIT Statement 6-40
ROLLBACK Statement 6-41
SAVEPOINT Statement 6-43
Implicit Rollbacks 6-44
SET TRANSACTION Statement 6-45
Overriding Default Locking 6-46
LOCK TABLE Statement 6-46
SELECT FOR UPDATE and FOR UPDATE Cursors 6-46
Simulating CURRENT OF Clause with ROWID Pseudocolumn 6-48
Autonomous Transactions 6-49
Advantages of Autonomous Transactions 6-50
Transaction Context 6-50
Transaction Visibility 6-50
Declaring Autonomous Transactions 6-51
Controlling Autonomous Transactions 6-52
Entering and Exiting 6-52
Committing and Rolling Back 6-53
Savepoints 6-53
Avoiding Errors with Autonomous Transactions 6-53
Autonomous Triggers 6-53
Invoking Autonomous Functions from SQL 6-56

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

Reasons to Use Subprograms 8-1
Nested, Package, and Standalone Stored Subprograms 8-2
Subprogram Invocations 8-2
Subprogram Parts 8-3
Additional Parts for Functions 8-4
RETURN Statement 8-5
RETURN Statement in Function 8-6
RETURN Statement in Procedure 8-7
RETURN Statement in Anonymous Block 8-8
Forward Declaration 8-8
Subprogram Parameters 8-9
Formal and Actual Subprogram Parameters 8-9
Subprogram Parameter Passing Methods 8-11
Subprogram Parameter Modes 8-12
Subprogram Parameter Aliasing 8-15
Subprogram Parameter Aliasing with Parameters Passed by Reference 8-16
Subprogram Parameter Aliasing with Cursor Variable Parameters 8-18
Default Values for IN Subprogram Parameters 8-18
Positional, Named, and Mixed Notation for Actual Parameters 8-21
Subprogram Invocation Resolution 8-23
Overloaded Subprograms 8-25
Formal Parameters that Differ Only in Numeric Data Type 8-26
Subprograms that You Cannot Overload 8-27
Subprogram Overload Errors 8-28
Recursive Subprograms 8-29
Subprogram Side Effects 8-31
PL/SQL Function Result Cache 8-31
Enabling Result-Caching for a Function 8-32
Developing Applications with Result-Cached Functions 8-33
Restrictions on Result-Cached Functions 8-34
Examples of Result-Cached Functions 8-35
Result-Cached Application Configuration Parameters 8-35
Result-Cached Recursive Function 8-36

Advanced Result-Cached Function Topics 8-37
Rules for a Cache Hit 8-37
Result Cache Bypass 8-37
Making Result-Cached Functions Handle Session-Specific Settings 8-38
Making Result-Cached Functions Handle Session-Specific Application Contexts 8-39
Choosing Result-Caching Granularity 8-40
Result Caches in Oracle RAC Environment 8-41
Result Cache Management 8-41
Hot-Patching PL/SQL Units on Which Result-Cached Functions Depend 8-42
PL/SQL Functions that SQL Statements Can Invoke 8-43
Invoker's Rights and Definer's Rights (AUTHID Property) 8-43
Choosing AUTHID CURRENT_USER or AUTHID DEFINER 8-45
AUTHID and SQL Command SET ROLE 8-45
Need for Template Objects in IR Units 8-46
Overriding Default Name Resolution in IR Units 8-46
IR Subprograms, Views, and Database Triggers 8-46
IR Database Links 8-46
IR ADTs 8-47
IR Instance Methods 8-48
External Subprograms 8-49

9 PL/SQL Triggers

Overview of Triggers 9-1
Reasons to Use Triggers 9-2
DML Triggers 9-3
Conditional Predicates for Detecting Triggering DML Statement 9-4
Correlation Names and Pseudorecords 9-5
OBJECT_VALUE Pseudocolumn 9-9
INSTEAD OF Triggers 9-10
Compound DML Triggers 9-14
Compound DML Trigger Structure 9-15
Compound DML Trigger Restrictions 9-16
Performance Benefit of Compound DML Triggers 9-16
Using Compound DML Triggers with Bulk Insertion 9-16
Using Compound DML Triggers to Avoid Mutating-Table Error 9-18
Triggers for Ensuring Referential Integrity 9-19
Foreign Key Trigger for Child Table 9-20
UPDATE and DELETE RESTRICT Trigger for Parent Table 9-21
UPDATE and DELETE SET NULL Trigger for Parent Table 9-22
DELETE CASCADE Trigger for Parent Table 9-22
UPDATE CASCADE Trigger for Parent Table 9-23
Triggers for Complex Constraint Checking 9-24
Triggers for Complex Security Authorizations 9-25
Triggers for Transparent Event Logging 9-26
Triggers for Deriving Column Values 9-26
Triggers for Building Complex Updatable Views 9-27
Triggers for Fine-Grained Access Control 9-30

System Triggers ... 9-30
SCHEMA Triggers ... 9-31
DATABASE Triggers ... 9-31
Subprograms Invoked by Triggers ... 9-32
Trigger Compilation, Invalidation, and Recompilation ... 9-32
Exception Handling in Triggers ... 9-33
Trigger Design Guidelines ... 9-35
Trigger Restrictions ... 9-36
Trigger Size Restriction ... 9-36
Trigger LONG and LONG RAW Data Type Restrictions ... 9-37
Mutating-Table Restriction ... 9-37
Order in Which Triggers Fire ... 9-40
Trigger Enabling and Disabling ... 9-41
Trigger Changing and Debugging ... 9-42
Triggers and Oracle Database Data Transfer Utilities ... 9-42
Triggers for Publishing Events ... 9-43
Event Attribute Functions ... 9-44
Event Attribute Functions for Database Event Triggers ... 9-47
Event Attribute Functions for Client Event Triggers ... 9-48
Views for Information About Triggers ... 9-52

10 PL/SQL Packages

What is a Package? ... 10-1
Reasons to Use Packages ... 10-2
Package Specification ... 10-3
Appropriate Public Items ... 10-3
Creating Package Specifications ... 10-4
Package Body ... 10-5
Package Instantiation and Initialization ... 10-6
Package State ... 10-7
SERIALLY_REUSABLE Packages ... 10-7
Creating SERIALLY_REUSABLE Packages ... 10-8
SERIALLY_REUSABLE Package Work Unit ... 10-9
Explicit Cursors in SERIALLY_REUSABLE Packages ... 10-10
Package Writing Guidelines ... 10-11
Package Example ... 10-12
How STANDARD Package Defines the PL/SQL Environment ... 10-16

11 PL/SQL Error Handling

Compile-Time Warnings ... 11-2
DBMS_WARNING Package ... 11-3
Overview of Exception Handling ... 11-4
Exception Categories ... 11-5
Advantages of Exception Handlers ... 11-6
Guidelines for Avoiding and Handling Exceptions ... 11-8
Internally Defined Exceptions ... 11-9

Predefined Exceptions 11-10
User-Defined Exceptions 11-11
Redeclared Predefined Exceptions 11-12
Raising Exceptions Explicitly 11-13
RAISE Statement 11-13
Raising User-Defined Exception with RAISE Statement 11-13
Raising Internally Defined Exception with RAISE Statement 11-14
Reraising Current Exception with RAISE Statement 11-15
RAISE_APPLICATION_ERROR Procedure 11-16
Exception Propagation 11-17
Propagation of Exceptions Raised in Declarations 11-20
Propagation of Exceptions Raised in Exception Handlers 11-20
Unhandled Exceptions 11-24
Error Code and Error Message Retrieval 11-24
Continuing Execution After Handling Exceptions 11-25
Retrying Transactions After Handling Exceptions 11-27

12 PL/SQL Optimization and Tuning

PL/SQL Optimizer 12-1
Subprogram Inlining 12-1
Candidates for Tuning 12-4
Minimizing CPU Overhead 12-4
Tune SQL Statements 12-5
Tune Function Invocations in Queries 12-5
Tune Subprogram Invocations 12-7
Tune Loops 12-8
Tune Computation-Intensive PL/SQL Code 12-8
Use Data Types that Use Hardware Arithmetic 12-9
Avoid Constrained Subtypes in Performance-Critical Code 12-9
Minimize Implicit Data Type Conversion 12-9
Use SQL Character Functions 12-10
Put Least Expensive Conditional Tests First 12-10
Bulk SQL and Bulk Binding 12-11
FORALL Statement 12-11
FORALL Statements for Sparse Collections 12-14
Unhandled Exceptions in FORALL Statements 12-17
Handling FORALL Exceptions Immediately 12-18
Handling FORALL Exceptions After FORALL Statement Completes 12-19
Getting Number of Rows Affected by FORALL Statement 12-21
BULK COLLECT Clause 12-23
SELECT INTO Statement with BULK COLLECT Clause 12-24
FETCH Statement with BULK COLLECT Clause 12-31
RETURNING INTO Clause with BULK COLLECT Clause 12-35
Using FORALL Statement and BULK COLLECT Clause Together 12-36
Client Bulk-Binding of Host Arrays 12-38
Chaining Pipelined Table Functions for Multiple Transformations 12-38
Overview of Table Functions 12-39

Creating Pipelined Table Functions 12-40
Pipelined Table Functions as Transformation Functions 12-41
Chaining Pipelined Table Functions 12-43
Fetching from Results of Pipelined Table Functions 12-43
Passing CURSOR Expressions to Pipelined Table Functions 12-44
DML Statements on Pipelined Table Function Results 12-46
NO_DATA_NEEDED Exception 12-46
Updating Large Tables in Parallel 12-48
Collecting Data About User-Defined Identifiers 12-49
Profiling and Tracing PL/SQL Programs 12-49
Profiler API: Package DBMS_PROFILER 12-50
Trace API: Package DBMS_TRACE 12-50
Compiling PL/SQL Units for Native Execution 12-51
Determining Whether to Use PL/SQL Native Compilation 12-51
How PL/SQL Native Compilation Works 12-52
Dependencies, Invalidation, and Revalidation 12-52
Setting Up a New Database for PL/SQL Native Compilation 12-53
Compiling the Entire Database for PL/SQL Native or Interpreted Compilation 12-53

13 PL/SQL Language Elements

Assignment Statement 13-3
AUTONOMOUS_TRANSACTION Pragma 13-6
Basic LOOP Statement 13-7
Block 13-9
CASE Statement 13-20
CLOSE Statement 13-23
Collection Variable Declaration 13-25
Collection Method Invocation 13-31
Comment 13-34
Constant Declaration 13-36
CONTINUE Statement 13-38
Cursor FOR LOOP Statement 13-40
Cursor Variable Declaration 13-42
DELETE Statement Extension 13-45
EXCEPTION_INIT Pragma 13-46
Exception Declaration 13-48
Exception Handler 13-50
EXECUTE IMMEDIATE Statement 13-52
EXIT Statement 13-55
Explicit Cursor Declaration and Definition 13-57
Expression 13-61
FETCH Statement 13-71
FOR LOOP Statement 13-74
FORALL Statement 13-77
Formal Parameter Declaration 13-80
Function Declaration and Definition 13-83
GOTO Statement 13-87

IF Statement 13-89
Implicit Cursor Attribute 13-91
INLINE Pragma 13-94
INSERT Statement Extension 13-96
Named Cursor Attribute 13-98
NULL Statement 13-100
OPEN Statement 13-101
OPEN FOR Statement 13-103
PIPE ROW Statement 13-106
Procedure Declaration and Definition 13-108
RAISE Statement 13-110
Record Variable Declaration 13-111
RESTRICT_REFERENCES Pragma 13-114
RETURN Statement 13-116
RETURNING INTO Clause 13-118
%ROWTYPE Attribute 13-121
Scalar Variable Declaration 13-123
SELECT INTO Statement 13-125
SERIALLY_REUSABLE Pragma 13-130
SQLCODE Function 13-131
SQLERRM Function 13-132
%TYPE Attribute 13-134
UPDATE Statement Extensions 13-136
WHILE LOOP Statement 13-138

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 14-41
CREATE PACKAGE Statement 14-43
CREATE PACKAGE BODY Statement 14-46
CREATE PROCEDURE Statement 14-50
CREATE TRIGGER Statement 14-54
CREATE TYPE Statement 14-68
CREATE TYPE BODY Statement 14-85
DROP FUNCTION Statement 14-90
DROP LIBRARY Statement 14-92
DROP PACKAGE Statement 14-93
DROP PROCEDURE Statement 14-95
DROP TRIGGER Statement 14-97
DROP TYPE Statement 14-98
DROP TYPE BODY Statement 14-100

A PL/SQL Source Text Wrapping

PL/SQL Source Text Wrapping Limitations A-1
PL/SQL Source Text Wrapping Guidelines A-2
PL/SQL Source Text Wrapping with PL/SQL Wrapper Utility A-2
PL/SQL Source Text Wrapping with DBMS_DDL Subprograms A-8

B PL/SQL Name Resolution

Qualified Names and Dot Notation B-1
Column Name Precedence B-3
Differences Between PL/SQL and SQL Name Resolution Rules B-5
Resolution of Names in Static SQL Statements B-5
What is Capture? B-6
Outer Capture B-6
Same-Scope Capture B-6
Inner Capture B-7
Avoiding Inner Capture in SELECT and DML Statements B-7
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

E PL/SQL Predefined Data Types

Index

List of Examples

1–1 PL/SQL Block Structure 1-5
1–2 Processing Query Result Rows One at a Time 1-9
2–1 Valid Case-Insensitive Reference to Quoted User-Defined Identifier 2-6
2–2 Invalid Case-Insensitive Reference to Quoted User-Defined Identifier 2-7
2–3 Reserved Word as Quoted User-Defined Identifier 2-7
2–4 Neglecting Double Quotation Marks 2-7
2–5 Neglecting Case-Sensitivity 2-8
2–6 Single-Line Comments 2-10
2–7 Multiline Comments 2-10
2–8 Whitespace Characters Improving Source Text Readability 2-11
2–9 Scalar Variable Declarations 2-12
2–10 Constant Declarations 2-12
2–11 Variable and Constant Declarations with Initial Values 2-13
2–12 Variable Initialized to NULL by Default 2-13
2–13 Variable Declaration with NOT NULL Constraint 2-14
2–14 Variables Initialized to NULL Values 2-14
2–15 Declaring Variable of Same Type as Column 2-15
2–16 Declaring Variable of Same Type as Another Variable 2-15
2–17 Scope and Visibility of Identifiers 2-17
2–18 Qualifying Redeclared Global Identifier with Block Label 2-17
2–19 Qualifying Identifier with Subprogram Name 2-18
2–20 Duplicate Identifiers in Same Scope 2-18
2–21 Declaring Same Identifier in Different Units 2-19
2–22 Label and Subprogram with Same Name in Same Scope 2-19
2–23 Block with Multiple and Duplicate Labels 2-20
2–24 Assigning Values to Variables with Assignment Statement 2-21
2–25 Assigning Value to Variable with SELECT INTO Statement 2-22
2–26 Assigning Value to Variable as IN OUT Subprogram Parameter 2-22
2–27 Assigning Value to BOOLEAN Variable 2-23
2–28 Concatenation Operator 2-24
2–29 Concatenation Operator with NULL Operands 2-24
2–30 Controlling Evaluation Order with Parentheses 2-25
2–31 Expression with Nested Parentheses 2-25
2–32 Improving Readability with Parentheses 2-26
2–33 Operator Precedence 2-26
2–34 Procedure Prints BOOLEAN Variable 2-27
2–35 AND Operator 2-27
2–36 OR Operator 2-28
2–37 NOT Operator 2-29
2–38 NULL Value in Unequal Comparison 2-30
2–39 NULL Value in Equal Comparison 2-30
2–40 NOT NULL Equals NULL 2-30
2–41 Changing Evaluation Order of Logical Operators 2-31
2–42 Short-Circuit Evaluation 2-31
2–43 Relational Operators in Expressions 2-33
2–44 LIKE Operator in Expression 2-34
2–45 Escape Character in Pattern 2-35
2–46 BETWEEN Operator in Expressions 2-35
2–47 IN Operator in Expressions 2-36
2–48 IN Operator with Sets with NULL Values 2-36
2–49 Equivalent BOOLEAN Expressions 2-37
2–50 Simple CASE Expression 2-38
2–51 Simple CASE Expression with WHEN NULL 2-38
2–52 Searched CASE Expression 2-39

2–53 Searched CASE Expression with WHEN ... IS NULL 2-40
2–54 Predefined Inquiry Directives $$PLSQL_LINE and $$PLSQL_UNIT 2-44
2–55 Displaying Values of PL/SQL Compilation Parameters 2-45
2–56 PLSQL_CCFLAGS Assigns Value to Itself 2-46
2–57 Static Constants 2-48
2–58 Code for Checking Database Version 2-50
2–59 Compiling Different Code for Different Database Versions 2-50
2–60 Displaying Post-Processed Source Textsource text 2-51
3–1 CHAR and VARCHAR2 Blank-Padding Difference 3-6
3–2 Printing BOOLEAN Values 3-7
3–3 PLS_INTEGER Calculation Raises Overflow Exception 3-8
3–4 Preventing Example 3–3 Overflow 3-9
3–5 Violating Constraint of SIMPLE_INTEGER Subtype 3-10
3–6 User-Defined Unconstrained Subtypes Show Intended Use 3-12
3–7 User-Defined Constrained Subtype Detects Out-of-Range Values 3-13
3–8 Implicit Conversion Between Constrained Subtypes with Same Base Type 3-13
3–9 Implicit Conversion Between Subtypes with Base Types in Same Family 3-14
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 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 FOR LOOP Statements 4-13
4–15 Reverse FOR LOOP Statements 4-14
4–16 Simulating STEP Clause in FOR LOOP Statement 4-15
4–17 FOR LOOP Statement Tries to Change Index Value 4-15
4–18 Outside Statement References FOR LOOP Statement Index 4-16
4–19 FOR LOOP Statement Index with Same Name as Variable 4-16
4–20 FOR LOOP Statement References Variable with Same Name as Index 4-16
4–21 Nested FOR LOOP Statements with Same Index Name 4-17
4–22 FOR LOOP Statement Bounds 4-17
4–23 Specifying FOR LOOP Statement Bounds at Run Time 4-18
4–24 EXIT WHEN Statement in FOR LOOP Statement 4-18
4–25 EXIT WHEN Statement in Inner FOR LOOP Statement 4-19
4–26 CONTINUE WHEN Statement in Inner FOR LOOP Statement 4-19
4–27 WHILE LOOP Statements 4-20
4–28 GOTO Statement 4-21
4–29 Incorrect Label Placement 4-21
4–30 GOTO Statement Goes to Labeled NULL Statement 4-22
4–31 GOTO Statement Transfers Control to Enclosing Block 4-22
4–32 GOTO Statement Cannot Transfer Control into IF Statement 4-23
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 Associative Array Indexed by String 5-4
5–2 Function Returns Associative Array Indexed by Integer 5-5
5–3 Varray (Variable-Size Array) 5-8

5–4 Nested Table of Local Type ... 5-10
5–5 Nested Table of Standalone Stored Type ... 5-11
5–6 Initializing Collection (Varray) Variable to Empty ... 5-13
5–7 Data Type Compatibility for Collection Assignment ... 5-14
5–8 Assigning Null Value to Nested Table Variable ... 5-15
5–9 Assigning Set Operation Results to Nested Table Variable ... 5-16
5–10 Two-Dimensional Varray (Varray of Varrays) ... 5-17
5–11 Nested Tables of Nested Tables and Varrays of Integers ... 5-17
5–12 Nested Tables of Associative Arrays and Varrays of Strings ... 5-18
5–13 Comparing Varray and Nested Table Variables to NULL ... 5-19
5–14 Comparing Nested Tables for Equality and Inequality ... 5-20
5–15 Comparing Nested Tables with SQL Multiset Conditions ... 5-20
5–16 DELETE Method with Nested Table ... 5-23
5–17 DELETE Method with Associative Array Indexed by String ... 5-24
5–18 TRIM Method with Nested Table ... 5-26
5–19 EXTEND Method with Nested Table ... 5-27
5–20 EXISTS Method with Nested Table ... 5-28
5–21 FIRST and LAST Values for Associative Array Indexed by Integer ... 5-29
5–22 FIRST and LAST Values for Associative Array Indexed by String ... 5-29
5–23 Printing Varray with FIRST and LAST in FOR LOOP ... 5-30
5–24 Printing Nested Table with FIRST and LAST in FOR LOOP ... 5-31
5–25 COUNT and LAST Values for Varray ... 5-33
5–26 COUNT and LAST Values for Nested Table ... 5-33
5–27 LIMIT and COUNT Values for Different Collection Types ... 5-34
5–28 PRIOR and NEXT Methods ... 5-35
5–29 Printing Elements of Sparse Nested Table ... 5-36
5–30 Identically Defined Package and Local Collection Types ... 5-37
5–31 Identically Defined Package and Standalone Stored Collection Types ... 5-38
5–32 RECORD Type Definition and Variable Declaration ... 5-39
5–33 RECORD Type with RECORD Field (Nested Record) ... 5-40
5–34 RECORD Type with Varray Field ... 5-40
5–35 Identically Defined Package and Local RECORD Types ... 5-41
5–36 %ROWTYPE Variable Represents Full Database Table Row ... 5-42
5–37 %ROWTYPE Variable Does Not Inherit Initial Values or Constraints ... 5-43
5–38 %ROWTYPE Variable Represents Partial Database Table Row ... 5-44
5–39 %ROWTYPE Variable Represents Join Row ... 5-44
5–40 Assigning Record to Another Record of Same RECORD Type ... 5-45
5–41 Assigning %ROWTYPE Record to RECORD Type Record ... 5-45
5–42 Assigning Nested Record to Another Record of Same RECORD Type ... 5-46
5–43 SELECT INTO Assigns Values to Record Variable ... 5-47
5–44 FETCH Assigns Values to Record that Function Returns ... 5-48
5–45 UPDATE Statement Assigns Values to Record Variable ... 5-49
5–46 Initializing Table by Inserting Record of Default Values ... 5-50
5–47 Updating Rows with Record ... 5-51
6–1 Static SQL Statements ... 6-2
6–2 CURRVAL and NEXTVAL Pseudocolumns ... 6-4
6–3 SQL%FOUND Implicit Cursor Attribute ... 6-6
6–4 SQL%ROWCOUNT Implicit Cursor Attribute ... 6-7
6–5 Explicit Cursor Declaration and Definition ... 6-9
6–6 FETCH Statements Inside LOOP Statements ... 6-10
6–7 Fetching Same Explicit Cursor into Different Variables ... 6-11
6–8 Variable in Explicit Cursor Query—No Result Set Change ... 6-12
6–9 Variable in Explicit Cursor Query—Result Set Change ... 6-13
6–10 Explicit Cursor with Calculated Column that Needs Alias ... 6-14
6–11 Explicit Cursor that Accepts Parameters ... 6-15

6–12 Cursor Parameters with Default Values 6-16
6–13 Adding Formal Parameter to Existing Cursor 6-18
6–14 %ISOPEN Explicit Cursor Attribute 6-20
6–15 %FOUND Explicit Cursor Attribute 6-20
6–16 %NOTFOUND Explicit Cursor Attribute 6-21
6–17 %ROWCOUNT Explicit Cursor Attribute 6-22
6–18 Implicit Cursor FOR LOOP Statement 6-24
6–19 Explicit Cursor FOR LOOP Statement 6-25
6–20 Passing Parameters to Explicit Cursor FOR LOOP Statement 6-25
6–21 Cursor FOR Loop References Calculated Columns 6-26
6–22 Subquery in FROM Clause of Parent Query 6-27
6–23 Correlated Subquery 6-27
6–24 Cursor Variable Declarations 6-29
6–25 Cursor Variable with User-Defined Return Type 6-29
6–26 Fetching Data with Cursor Variables 6-31
6–27 Fetching from Cursor Variable into Collections 6-32
6–28 Variable in Cursor Variable Query—No Result Set Change 6-33
6–29 Variable in Cursor Variable Query—Result Set Change 6-33
6–30 Procedure to Open Cursor Variable for One Query 6-35
6–31 Opening Cursor Variable for Chosen Query (Same Return Type) 6-36
6–32 Opening Cursor Variable for Chosen Query (Different Return Types) 6-36
6–33 Cursor Variable as Host Variable in Pro*C Client Program 6-37
6–34 CURSOR Expression 6-39
6–35 COMMIT Statement with COMMENT and WRITE Clauses 6-40
6–36 ROLLBACK Statement 6-42
6–37 SAVEPOINT and ROLLBACK Statements 6-43
6–38 Reusing SAVEPOINT with ROLLBACK 6-44
6–39 SET TRANSACTION Statement in Read-Only Transaction 6-45
6–40 FOR UPDATE Cursor in CURRENT OF Clause of UPDATE Statement 6-47
6–41 SELECT FOR UPDATE Statement for Multiple Tables 6-47
6–42 FETCH with FOR UPDATE Cursor After COMMIT Statement 6-48
6–43 Simulating CURRENT OF Clause with ROWID Pseudocolumn 6-49
6–44 Declaring Autonomous Function in Package 6-51
6–45 Declaring Autonomous Standalone Procedure 6-51
6–46 Declaring Autonomous PL/SQL Block 6-52
6–47 Autonomous Trigger Logs INSERT Statements 6-54
6–48 Autonomous Trigger Uses Native Dynamic SQL for DDL 6-55
6–49 Invoking Autonomous Function 6-56
7–1 Invoking Subprogram from Dynamic PL/SQL Block 7-3
7–2 Unsupported Data Type in Native Dynamic SQL 7-4
7–3 Uninitialized Variable Represents 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 Execution Resumes After RETURN Statement in Function 8-6

8–4 Function Where Not Every Execution Path Leads to RETURN Statement 8-6
8–5 Function Where Every Execution Path Leads to RETURN Statement 8-7
8–6 Execution Resumes After RETURN Statement in Procedure 8-7
8–7 Execution Resumes After RETURN Statement in Anonymous Block 8-8
8–8 Nested Subprograms Invoke Each Other 8-8
8–9 Formal Parameters and Actual Parameters 8-9
8–10 Parameter Inherits Only NOT NULL from Subtype 8-10
8–11 Avoiding Implicit Conversion of Actual Parameters 8-11
8–12 Parameter Values Before, During, and After Procedure Invocation 8-13
8–13 OUT and IN OUT Parameter Values After Unhandled Exception 8-15
8–14 Aliasing from Global Variable as Actual Parameter 8-16
8–15 Aliasing from Same Actual Parameter for Multiple Formal Parameters 8-17
8–16 Aliasing from Cursor Variable Subprogram Parameters 8-18
8–17 Procedure with Default Parameter Values 8-19
8–18 Function Provides Default Parameter Value 8-19
8–19 Adding Subprogram Parameter Without Changing Existing Invocations 8-20
8–20 Equivalent Invocations with Different Notations in Anonymous Block 8-22
8–21 Equivalent Invocations with Different Notations in SELECT Statements 8-22
8–22 Resolving PL/SQL Procedure Names 8-24
8–23 Overloaded Subprogram 8-26
8–24 Overload Error Causes Compile-Time Error 8-28
8–25 Overload Error Compiles Successfully 8-28
8–26 Invoking Subprogram in Example 8–25 Causes Compile-Time Error 8-28
8–27 Correcting Overload Error in Example 8–25 8-29
8–28 Invoking Subprogram in Example 8–27 8-29
8–29 Package Specification Without Overload Errors 8-29
8–30 Improper Invocation of Properly Overloaded Subprogram 8-29
8–31 Recursive Function Returns n Factorial (n!) 8-30
8–32 Recursive Function Returns nth Fibonacci Number 8-30
8–33 Declaring and Defining Result-Cached Function 8-32
8–34 Result-Cached Function Returns Configuration Parameter Setting 8-36
8–35 Result-Cached Function Handles Session-Specific Settings 8-38
8–36 Result-Cached Function Handles Session-Specific Application Context 8-39
8–37 Caching One Name at a Time (Finer Granularity) 8-40
8–38 Caching Translated Names One Language at a Time (Coarser Granularity) 8-41
8–39 ADT for Use in Any Schema 8-47
8–40 Invoking IR Instance Method 8-48
8–41 PL/SQL Anonymous Block Invokes External Procedure 8-50
8–42 PL/SQL Standalone Stored Procedure Invokes External Procedure 8-50
9–1 Trigger Uses Conditional Predicates to Detect Triggering Statement 9-5
9–2 Trigger Logs Changes to EMPLOYEES.SALARY 9-6
9–3 Conditional Trigger Prints Salary Change Information 9-7
9–4 Trigger Modifies LOB Columns 9-8
9–5 Trigger with REFERENCING Clause 9-9
9–6 Trigger References OBJECT_VALUE Pseudocolumn 9-9
9–7 INSTEAD OF Trigger 9-11
9–8 INSTEAD OF Trigger on Nested Table Column of View 9-12
9–9 Compound Trigger Logs Changes to One Table in Another Table 9-17
9–10 Compound Trigger Avoids Mutating-Table Error 9-18
9–11 Foreign Key Trigger for Child Table 9-20
9–12 UPDATE and DELETE RESTRICT Trigger for Parent Table 9-21
9–13 UPDATE and DELETE SET NULL Trigger for Parent Table 9-22
9–14 DELETE CASCADE Trigger for Parent Table 9-22
9–15 UPDATE CASCADE Trigger for Parent Table 9-23
9–16 Trigger Checks Complex Constraints 9-24

9–17 Trigger Enforces Security ... 9-25
9–18 Trigger Derives New Column Values ... 9-27
9–19 BEFORE Statement Trigger on Sample Schema HR ... 9-31
9–20 AFTER Statement Trigger on Database ... 9-31
9–21 Trigger Monitors Logons ... 9-31
9–22 Trigger Invokes Java Subprogram ... 9-32
9–23 Trigger Cannot Handle Exception if Remote Database is Unavailable ... 9-34
9–24 Workaround for Example 9–23 ... 9-34
9–25 Trigger Causes Mutating-Table Error ... 9-37
9–26 Update Cascade ... 9-39
9–27 Viewing Information About Triggers ... 9-52
10–1 Simple Package Specification ... 10-4
10–2 Passing Associative Array to Standalone Subprogram ... 10-4
10–3 Matching Package Specification and Body ... 10-5
10–4 Creating SERIALLY_REUSABLE Packages ... 10-8
10–5 Effect of SERIALLY_REUSABLE Pragma ... 10-9
10–6 Cursor in SERIALLY_REUSABLE Package Open at Call Boundary ... 10-10
10–7 Separating Cursor Declaration and Definition in Package ... 10-12
10–8 Creating emp_admin Package ... 10-12
11–1 Setting Value of PLSQL_WARNINGS Compilation Parameter ... 11-3
11–2 Displaying and Setting PLSQL_WARNINGS with DBMS_WARNING Subprograms 11-3
11–3 Single Exception Handler for Multiple Exceptions ... 11-6
11–4 Locator Variables for Statements that Share Exception Handler ... 11-7
11–5 Naming Internally Defined Exception ... 11-10
11–6 Anonymous Block Handles ZERO_DIVIDE ... 11-11
11–7 Anonymous Block Avoids ZERO_DIVIDE ... 11-11
11–8 Redeclared Predefined Identifier ... 11-12
11–9 Declaring, Raising, and Handling User-Defined Exception ... 11-13
11–10 Explicitly Raising Predefined Exception ... 11-14
11–11 Reraising Exception ... 11-15
11–12 Raising User-Defined Exception with RAISE_APPLICATION_ERROR ... 11-16
11–13 Exception that Propagates Beyond Scope is Handled ... 11-19
11–14 Exception that Propagates Beyond Scope is Not Handled ... 11-19
11–15 Exception Raised in Declaration is Not Handled ... 11-20
11–16 Exception Raised in Declaration is Handled by Enclosing Block ... 11-20
11–17 Exception Raised in Exception Handler is Not Handled ... 11-21
11–18 Exception Raised in Exception Handler is Handled by Invoker ... 11-21
11–19 Exception Raised in Exception Handler is Handled by Enclosing Block ... 11-22
11–20 Exception Raised in Exception Handler is Not Handled ... 11-22
11–21 Exception Raised in Exception Handler is Handled by Enclosing Block ... 11-23
11–22 Displaying SQLCODE and SQLERRM Values ... 11-25
11–23 Exception Handler Runs and Execution Ends ... 11-25
11–24 Exception Handler Runs and Execution Continues ... 11-26
11–25 Retrying Transaction After Handling Exception ... 11-27
12–1 Specifying that Subprogram Is To Be Inlined ... 12-3
12–2 Specifying that Overloaded Subprogram Is To Be Inlined ... 12-3
12–3 Specifying that Subprogram Is Not To Be Inlined ... 12-3
12–4 PRAGMA INLINE ... 'NO' Overrides PRAGMA INLINE ... 'YES' ... 12-3
12–5 Nested Query Improves Performance ... 12-6
12–6 NOCOPY Subprogram Parameters ... 12-7
12–7 DELETE Statement in FOR LOOP Statement ... 12-12
12–8 DELETE Statement in FORALL Statement ... 12-12
12–9 Time Difference for INSERT Statement in FOR LOOP and FORALL Statements ... 12-12
12–10 FORALL Statement for Subset of Collection ... 12-13
12–11 FORALL Statements for Sparse Collection and Its Subsets ... 12-14

12–12 Handling FORALL Exceptions Immediately 12-18
12–13 Handling FORALL Exceptions After FORALL Statement Completes 12-20
12–14 Showing Number of Rows Affected by Each DELETE in FORALL 12-21
12–15 Showing Number of Rows Affected by Each INSERT SELECT in FORALL 12-22
12–16 Bulk-Selecting Two Database Columns into Two Nested Tables 12-24
12–17 Bulk-Selecting into Nested Table of Records 12-25
12–18 SELECT BULK COLLECT INTO Statement with Unexpected Results 12-26
12–19 Cursor Workaround for Example 12–18 12-27
12–20 Second Collection Workaround for Example 12–18 12-28
12–21 Limiting Bulk Selection with ROWNUM and SAMPLE 12-30
12–22 Bulk-Fetching into Two Nested Tables 12-31
12–23 Bulk-Fetching into Nested Table of Records 12-33
12–24 Limiting Bulk FETCH with LIMIT 12-34
12–25 Returning Deleted Rows in Two Nested Tables 12-36
12–26 DELETE with RETURN BULK COLLECT INTO in FORALL Statement 12-36
12–27 DELETE with RETURN BULK COLLECT INTO in FOR LOOP Statement 12-37
12–28 Anonymous Block Bulk-Binds Input Host Array 12-38
12–29 Creating and Invoking Pipelined Table Function 12-41
12–30 Pipelined Table Function Transforms Each Row to Two Rows 12-42
12–31 Fetching from Results of Pipelined Table Functions 12-43
12–32 Pipelined Table Function with Two Cursor Variable Parameters 12-44
12–33 Pipelined Table Function as Aggregate Function 12-45
12–34 Pipelined Table Function Does Not Handle NO_DATA_NEEDED 12-47
12–35 Pipelined Table Function Handles NO_DATA_NEEDED 12-48
A–1 SQL File with Two Wrappable PL/SQL Units A-3
A–2 Wrapping File with PL/SQL Wrapper Utility A-4
A–3 Running Wrapped File and Viewing Wrapped PL/SQL Units A-5
A–4 Creating Wrapped Package Body with CREATE_WRAPPED Procedure A-8
A–5 Viewing Package with Wrapped Body and Invoking Package Procedure A-9
B–1 Qualified Names B-2
B–2 Variable Name Interpreted as Column Name Causes Unintended Result B-3
B–3 Fixing Example B–2 with Different Variable Name B-3
B–4 Fixing Example B–2 with Block Label B-4
B–5 Subprogram Name for Name Resolution B-4
B–6 Inner Capture of Column Reference B-7
B–7 Inner Capture of Attribute Reference B-8
B–8 Qualifying ADT Attribute References B-9
B–9 Qualifying References to Row Expressions B-9

List of Figures

1–1 PL/SQL Engine 1-10
5–1 Varray of Maximum Size 10 with 7 Elements 5-8
5–2 Array and Nested Table 5-12
6–1 Transaction Control Flow 6-49
8–1 How PL/SQL Compiler Resolves Invocations 8-24
11–1 Exception Does Not Propagate 11-17
11–2 Exception Propagates from Inner Block to Outer Block 11-18
11–3 PL/SQL Returns Unhandled Exception Error to Host Environment 11-18

List of Tables

1–1 PL/SQL I/O-Processing Packages 1-6
1–2 PL/SQL Compilation Parameters 1-11
2–1 Punctuation Characters in Every Database Character Set 2-2
2–2 PL/SQL Delimiters 2-3
2–3 Operator Precedence 2-25
2–4 Logical Truth Table 2-27
2–5 Relational Operators 2-32
3–1 Data Types with Different Maximum Sizes in PL/SQL and SQL 3-2
3–2 Predefined PL/SQL BINARY_FLOAT and BINARY_DOUBLE Constants 3-3
3–3 Predefined Subtypes of PLS_INTEGER Data Type 3-9
5–1 PL/SQL Collection Types 5-3
5–2 Collection Methods 5-21
8–1 PL/SQL Subprogram Parameter Modes 8-12
8–2 PL/SQL Actual Parameter Notations 8-21
8–3 Finer and Coarser Caching Granularity 8-40
9–1 OLD and NEW Pseudorecord Field Values 9-6
9–2 Compound Trigger Timing-Point Sections 9-15
9–3 Constraints and Triggers for Ensuring Referential Integrity 9-19
9–4 System-Defined Event Attributes 9-45
9–5 Database Event Triggers 9-48
9–6 Client Event Triggers 9-49
11–1 Compile-Time Warning Categories 11-2
11–2 Exception Categories 11-6
11–3 PL/SQL Predefined Exceptions 11-10
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.

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.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/support/contact.html or visit http://www.oracle.com/accessibility/support.html if you are hearing impaired.

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.
`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*.

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 Feature for Oracle Database 11g Release 2 (11.2.0.2)
- PL/SQL Features for Oracle Database 11g Release 2 (11.2.0.1)
- PL/SQL Features for Oracle Database 11g Release 1 (11.1)

PL/SQL Feature for Oracle Database 11*g* Release 2 (11.2.0.2)

Package Treated as Stateless if State is Constant for Life of Session

Before Release 11.2.0.2, if a session recompiled the body of a stateful package, and then another session that had instantiated that package referenced it, the latter session got the severely disruptive error ORA-04068 (“existing state of packages … has been discarded”). Therefore, "hot patching" packages was likely to disrupt their users.

As of Release 11.2.0.2, Oracle Database treats a package as stateless if its state is constant for the life of a session (or longer). This is the case for a package whose items are all compile-time constants. Therefore, "hot patching" packages (especially noneditioned packages) is much less likely to disrupt sessions that are using them.

For more information, see "Package State" on page 10-7.

PL/SQL Features for Oracle Database 11*g* Release 2 (11.2.0.1)

The PL/SQL features for Oracle Database 11*g* Release 2 (11.2.0.1) 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.

Oracle recommends this technique 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 12-48.

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 14-68.

Crossedition Triggers

Crossedition triggers are intended to fire when database manipulation language (DML) statements change database tables while an online application that uses the tables is being patched or upgraded with edition-based redefinition. The body of a crossedition trigger is designed to handle these 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 14-54.

> **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 8-31.

Result Caches in Oracle RAC Environment Are No Longer Private

For Oracle Database 11*g* Release 1 (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 8-41.

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

The PL/SQL features for Release 11.1 are:

- Enhancements to Regular Expression 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 DML Triggers
- More Control Over Triggers
- Automatic Subprogram Inlining
- PL/Scope
- PL/SQL Hierarchical Profiler
- PL/SQL Native Compiler Generates Native Code Directly

Enhancements to Regular Expression SQL Functions

The regular expression SQL functions `REGEXP_INSTR` and `REGEXP_SUBSTR` have increased functionality. A new regular expression SQL 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 3-10
- "Additional PL/SQL Constants for BINARY_FLOAT and BINARY_DOUBLE" on page 3-2
- "Use Data Types that Use Hardware Arithmetic" on page 12-9

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 13-38

Sequences in PL/SQL Expressions

The pseudocolumns `CURRVAL` and `NEXTVAL` make writing PL/SQL source text 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 in PL/SQL" 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 13-52 and "OPEN FOR Statement" on page 13-103.

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 in Example 8–21 on page 8-22.

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 resides 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 8-31
- "Function Declaration and Definition" on page 13-83

Compound DML Triggers

A compound DML trigger created on a table or editioning view can fire at multiple timing points. Each timing point section has its own executable part and optional exception-handling part, but all of these parts can access a common PL/SQL state. The common state arises when the triggering statement starts and disappears 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 help program an approach where you want the actions you implement for the various timing points to share common data.

For more information, see "Compound DML 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 9-40
- "Trigger Enabling and Disabling" on page 9-41

Automatic Subprogram Inlining

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

You can use `PRAGMA INLINE` to specify that individual subprogram invocations 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:

- "Subprogram Inlining" on page 12-1
- "INLINE Pragma" on page 13-94

> **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 text. 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 text browsers that increase PL/SQL developer productivity by minimizing time spent browsing and understanding source text.

For more information, see "Collecting Data About User-Defined Identifiers" on page 12-49.

> **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 invocations. 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 invocations 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 12-49.

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 12-51.

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
- 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. For example:

- 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 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 1-9).

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, 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 B-5). 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 subprograms are stored in executable form, which can be invoked repeatedly. Because stored subprograms run in the database server, a single invocation 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 invocation overhead. For more information about subprograms, see "Subprograms" on page 1-6.

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 1-6.

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 1-6.

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 statements, 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. For information about packages, see "Packages" on page 1-6.

> **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 text. 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

- Error Handling
- Blocks
- Variables and Constants
- Subprograms
- Packages
- Triggers
- Input and Output
- Data Abstraction
- Control Statements
- Conditional Compilation
- Processing a Query Result Set One Row at a Time

Error Handling

PL/SQL makes it easy to detect and handle errors. When an error occurs, PL/SQL raises an exception. Normal execution stops and control transfers to the exception-handling part of the PL/SQL block. You do not have to check every

operation to ensure that it succeeded, as in a C program. For more information, see Chapter 11, "PL/SQL Error Handling".

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 divide 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 13-9.

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

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

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

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 2-11 and "Assigning Values to Variables" on page 2-20.

Subprograms

A PL/SQL **subprogram** is a named PL/SQL block that can be invoked repeatedly. If the subprogram has parameters, their values can differ for each invocation. PL/SQL has two types of subprograms, procedures and functions. A function returns a result. 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 8-49.

Packages

A **package** is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents. You can think of a package as an application.

You can write your own packages—for details, see Chapter 10, "PL/SQL Packages." You can also use the many product-specific packages that Oracle Database supplies. For information about these, see *Oracle Database PL/SQL Packages and Types Reference*.

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 for each event or 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."

Input and Output

Most PL/SQL input and output (I/O) is done with SQL statements that store data in database tables or query those tables. For information about SQL statements, see *Oracle Database SQL Language Reference*.

All other PL/SQL I/O is done with PL/SQL packages that Oracle Database supplies, which Table 1–1 summarizes.

Table 1–1 PL/SQL I/O-Processing Packages

Package	Description	More Information
`DBMS_OUTPUT`	Lets PL/SQL blocks, subprograms, packages, and triggers display output. Especially useful for displaying PL/SQL debugging information.	*Oracle Database PL/SQL Packages and Types Reference*
`HTF`	Has hypertext functions that generate HTML tags (for example, the `HTF.ANCHOR` function generates the HTML anchor tag `<A>`).	*Oracle Database PL/SQL Packages and Types Reference*
`HTP`	Has hypertext procedures that generate HTML tags.	*Oracle Database PL/SQL Packages and Types Reference*
`DBMS_PIPE`	Lets two or more sessions in the same instance communicate.	*Oracle Database PL/SQL Packages and Types Reference*
`UTL_FILE`	Lets PL/SQL programs read and write operating system files.	*Oracle Database PL/SQL Packages and Types Reference*

Table 1–1 (Cont.) PL/SQL I/O-Processing Packages

Package	Description	More Information
UTL_HTTP	Lets PL/SQL programs make Hypertext Transfer Protocol (HTTP) callouts, and access data on the Internet over HTTP.	*Oracle Database PL/SQL Packages and Types Reference*
UTL_SMTP	Sends electronic mails (emails) over Simple Mail Transfer Protocol (SMTP) as specified by RFC821.	*Oracle Database PL/SQL Packages and Types Reference*

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.

Some subprograms in the packages in Table 1–1 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 SET SERVEROUTPUT ON
- *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

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 6-5.

Composite Variables

A **composite variable** has internal components, which you can access individually. You can 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 either a full or partial row of a database table or view. For every column of the full or partial row, the record has a field with the same name and data type. If the structure of the row changes, then the structure of the record changes accordingly. For more information about `%ROWTYPE`, see "%ROWTYPE Attribute" on page 5-42.

%TYPE Attribute

The `%TYPE` attribute lets you declare a data item of the same data type as a previously declared variable or column (without knowing what that type is). If the declaration of the referenced item changes, then 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 2-14.

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 14-68.

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.

Conditional Compilation

Conditional compilation lets you customize the functionality in a PL/SQL application without removing source text. 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 2-41.

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 basic loop, as in Example 1–2, or you can control the process precisely by using individual statements to run the query, retrieve the results, 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
    ORDER BY employee_id
  )
  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

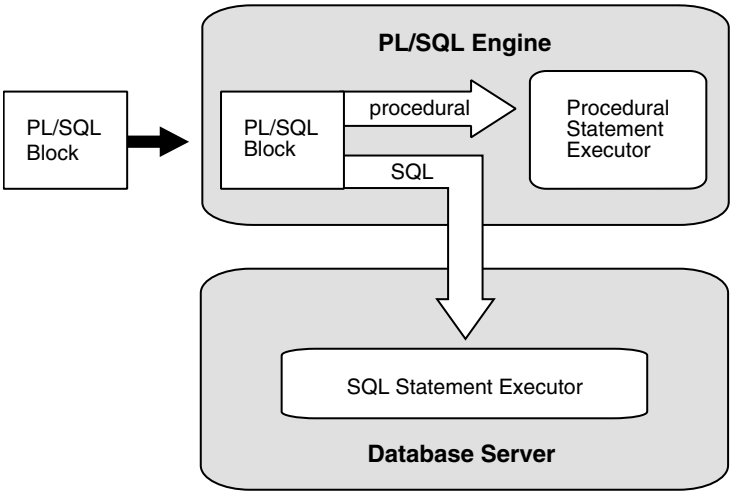
- PL/SQL Engine
- PL/SQL Units and Compilation Parameters

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 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 1–2 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 1–2 PL/SQL Compilation Parameters

Parameter	Description
PLSCOPE_SETTINGS	Controls the compile-time collection, cross-reference, and storage of PL/SQL source text identifier data. Used by the PL/Scope tool (see *Oracle Database Advanced Application Developer's Guide*). For more information about PLSCOPE_SETTINGS, see *Oracle Database Reference*.
PLSQL_CCFLAGS	Enables you to control conditional compilation of each PL/SQL unit independently. For more information about PLSQL_CCFLAGS, see "How Conditional Compilation Works" on page 2-42 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 12-51. If the optimization level (set by PLSQL_OPTIMIZE_LEVEL) is less than 2: - The compiler generates interpreted code, regardless of PLSQL_CODE_TYPE. - If you specify NATIVE, the compiler warns you that NATIVE was ignored. 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). PLSQL_OPTIMIZE_LEVEL=1 instructs the PL/SQL compiler to generate and store code for use by the PL/SQL debugger. For more information about PLSQL_OPTIMIZE_LEVEL, see "PL/SQL Optimizer" on page 12-1 and *Oracle Database Reference*.

Table 1–2 (Cont.) PL/SQL Compilation Parameters

Parameter	Description
PLSQL_WARNINGS	Enables or disables the reporting of warning messages by the PL/SQL compiler, and specifies which warning messages to show as errors. For more information about PLSQL_WARNINGS, see "Compile-Time Warnings" on page 11-2 and *Oracle Database Reference*.
NLS_LENGTH_SEMANTICS	Enables you to create CHAR and VARCHAR2 columns using either byte-length or character-length semantics. For more information about byte and character length semantics, see "CHAR and VARCHAR2 Variables" on page 3-3. 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 1–2 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

Any character data to be processed by PL/SQL or stored in a database must be represented as a sequence of bytes. The byte representation of a single character is called a **character code**. A set of character codes is called a **character set**.

Every Oracle database supports a database character set and a national character set. PL/SQL also supports these character sets. This document explains how PL/SQL uses the database character set and national character set.

Topics

- Database Character Set
- National Character Set

> **See Also:** *Oracle Database Globalization Support Guide* for general information about character sets

Database Character Set

PL/SQL uses the **database character set** to represent:

- Stored source text of PL/SQL units

For information about PL/SQL units, see "PL/SQL Units and Compilation Parameters" on page 1-10.

- Character values of data types `CHAR`, `VARCHAR2`, `CLOB`, and `LONG`

  For information about these data types, see "SQL Data Types" on page 3-2.

The database character set can be either single-byte, mapping each supported character to one particular byte, or multibyte-varying-width, mapping each supported character to a sequence of one, two, three, or four bytes. The maximum number of bytes in a character code depends on the particular character set.

Every database character set includes these basic characters:

- **Latin letters:** *A* through *Z* and *a* through *z*
- **Decimal digits:** *0* through *9*
- **Punctuation characters** in Table 2–1
- **Whitespace characters:** *space*, *tab*, *new line*, and *carriage return*

PL/SQL source text that uses only the basic characters can be stored and compiled in any database. PL/SQL source text that uses nonbasic characters can be stored and compiled only in databases whose database character sets support those nonbasic characters.

Table 2–1 Punctuation Characters in Every Database Character Set

Symbol	Name
(	Left parenthesis
)	Right parenthesis
<	Left angle bracket
>	Right angle bracket
+	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
@	At sign
%	Percent sign
#	Number sign
$	Dollar sign

Table 2–1 (Cont.) Punctuation Characters in Every Database Character Set

Symbol	Name
_	Underscore
|	Vertical bar

See Also: *Oracle Database Globalization Support Guide* for more information about the database character set

National Character Set

PL/SQL uses the **national character set** to represent character values of data types `NCHAR`, `NVARCHAR2` and `NCLOB`. For information about these data types, see "SQL Data Types" on page 3-2.

See Also: *Oracle Database Globalization Support Guide* for more information about the national character set

Lexical Units

The **lexical units** of PL/SQL are its smallest individual components—delimiters, identifiers, literals, and comments.

Topics

- Delimiters
- Identifiers
- Literals
- Comments
- Whitespace Characters Between Lexical Units

Delimiters

A **delimiter** is a character, or character combination, that has a special meaning in PL/SQL. Do not embed any others characters (including whitespace characters) inside a delimiter.

Table 2–2 summarizes the PL/SQL delimiters.

Table 2–2 PL/SQL Delimiters

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

Table 2–2 (Cont.) PL/SQL Delimiters

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

Identifiers

Identifiers name PL/SQL elements, which include:

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

- Variables

Every character in an identifier, alphabetic or not, is significant. For example, the identifiers `lastname` and `last_name` are different.

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

Except as explained in "Quoted User-Defined Identifiers" on page 2-6, PL/SQL is case-insensitive for identifiers. For example, the identifiers `lastname`, `LastName`, and `LASTNAME` are the same.

Topics

- Reserved Words and Keywords
- Predefined Identifiers
- User-Defined 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 them as quoted user-defined identifiers, but it is not recommended. For more information, see "Quoted User-Defined Identifiers" on page 2-6.

You can use keywords as ordinary user-defined identifiers, but it is not recommended.

For lists of PL/SQL reserved words and keywords, see Table D–1 and Table 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`.

For a list of predefined identifiers, use this query:

```
SELECT TYPE_NAME FROM ALL_TYPES WHERE PREDEFINED='YES';
```

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 2-16).

User-Defined Identifiers

A **user-defined identifier** is:

- Composed of characters from the database character set
- Either ordinary or quoted

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

Ordinary User-Defined Identifiers An ordinary user-defined identifier:

- Begins with a letter
- Can include letters, digits, and these symbols:
  - Dollar sign ($)

  - Number sign (#)
  - Underscore (_)
- Is not a reserved word (listed in Table D–1).

The database character set defines which characters are classified as letters and digits. The representation of the identifier in the database character set cannot exceed 30 bytes.

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
```

Quoted User-Defined Identifiers A quoted user-defined identifier is enclosed in double quotation marks. Between the double quotation marks, any characters from the database character set are allowed except double quotation marks, new line characters, and null characters. For example, these identifiers are acceptable:

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

The representation of the quoted identifier in the database character set cannot exceed 30 bytes (excluding the double quotation marks).

A quoted user-defined identifier is case-sensitive, with one exception: If a quoted user-defined identifier, without its enclosing double quotation marks, is a valid *ordinary* user-defined identifier, then the double quotation marks are optional in references to the identifier, and if you omit them, then the identifier is case-insensitive.

In Example 2–1, the quoted user-defined identifier `"HELLO"`, without its enclosing double quotation marks, is a valid ordinary user-defined identifier. Therefore, the reference `Hello` is valid.

Example 2–1 Valid Case-Insensitive Reference to Quoted User-Defined Identifier

```
DECLARE
  "HELLO" varchar2(10) := 'hello';
BEGIN
  DBMS_Output.Put_Line(Hello);
END;
/
```

Result:

```
hello
```

In Example 2–2, the reference `"Hello"` is invalid, because the double quotation marks make the identifier case-sensitive.

Example 2–2 Invalid Case-Insensitive Reference to Quoted User-Defined Identifier

```
DECLARE
  "HELLO" varchar2(10) := 'hello';
BEGIN
  DBMS_Output.Put_Line("Hello");
END;
/
```

Result:

```
  DBMS_Output.Put_Line("Hello");
                       *
ERROR at line 4:
ORA-06550: line 4, column 25:
PLS-00201: identifier 'Hello' must be declared
ORA-06550: line 4, column 3:
PL/SQL: Statement ignored
```

It is not recommended, but you can use a reserved word as a quoted user-defined identifier. Because a reserved word is not a valid ordinary user-defined identifier, you must always enclose the identifier in double quotation marks, and it is always case-sensitive.

Example 2–3 declares quoted user-defined identifiers `"BEGIN"`, `"Begin"`, and `"begin"`. Although `BEGIN`, `Begin`, and `begin` represent the same reserved word, `"BEGIN"`, `"Begin"`, and `"begin"` represent different identifiers.

Example 2–3 Reserved Word as Quoted User-Defined Identifier

```
DECLARE
  "BEGIN" varchar2(15) := 'UPPERCASE';
  "Begin" varchar2(15) := 'Initial Capital';
  "begin" varchar2(15) := 'lowercase';
BEGIN
  DBMS_Output.Put_Line("BEGIN");
  DBMS_Output.Put_Line("Begin");
  DBMS_Output.Put_Line("begin");
END;
/
```

Result:

```
UPPERCASE
Initial Capital
lowercase

PL/SQL procedure successfully completed.
```

Example 2–4 references a quoted user-defined identifier that is a reserved word, neglecting to enclose it in double quotation marks.

Example 2–4 Neglecting Double Quotation Marks

```
DECLARE
  "HELLO" varchar2(10) := 'hello';  -- HELLO is not a reserved word
```

```
  "BEGIN" varchar2(10) := 'begin';  -- BEGIN is a reserved word
BEGIN
  DBMS_Output.Put_Line(Hello);      -- Double quotation marks are optional
  DBMS_Output.Put_Line(BEGIN);      -- Double quotation marks are required
end;
/
```

Result:

```
  DBMS_Output.Put_Line(BEGIN);      -- Double quotation marks are required
                      *
ERROR at line 6:
ORA-06550: line 6, column 24:
PLS-00103: Encountered the symbol "BEGIN" when expecting one of the following:
( ) - + case mod new not null <an identifier>
<a double-quoted delimited-identifier> <a bind variable>
table continue avg count current exists max min prior sql
stddev sum variance execute multiset the both leading
trailing forall merge year month day hour minute second
timezone_hour timezone_minute timezone_region timezone_abbr
time timestamp interval date
<a string literal with character set specificat
```

Example 2–5 references a quoted user-defined identifier that is a reserved word, neglecting its case-sensitivity.

Example 2–5 Neglecting Case-Sensitivity

```
DECLARE
  "HELLO" varchar2(10) := 'hello';  -- HELLO is not a reserved word
  "BEGIN" varchar2(10) := 'begin';  -- BEGIN is a reserved word
BEGIN
  DBMS_Output.Put_Line(Hello);      -- Identifier is case-insensitive
  DBMS_Output.Put_Line("Begin");    -- Identifier is case-sensitive
END;
/
```

Result:

```
  DBMS_Output.Put_Line("Begin");    -- Identifier is case-sensitive
                       *
ERROR at line 6:
ORA-06550: line 6, column 25:
PLS-00201: identifier 'Begin' must be declared
ORA-06550: line 6, column 3:
PL/SQL: Statement ignored
```

Literals

A **literal** is a value that is neither represented by an identifier nor calculated from other values. For example, `123` is an integer literal and `'abc'` is a character literal, but `1+2` is not a literal.

PL/SQL literals include all SQL literals (described in *Oracle Database SQL Language Reference*) and `BOOLEAN` literals (which SQL does not have). A`BOOLEAN` literal is the predefined logical value `TRUE`, `FALSE`, or `NULL`. `NULL` represents an unknown value.

> **Note:** Like *Oracle Database SQL Language Reference*, this document uses the terms *character literal* and *string* interchangeably.

When using character literals in PL/SQL, remember:

- Character literals are case-sensitive.

  For example, `'Z'` and `'z'` are different.

- Whitespace characters are significant.

  For example, these literals are different:

  ```
  'abc'
  ' abc'
  'abc '
  ' abc '
  'a b c'
  ```

- `'0'` through `'9'` are not equivalent to the integer literals 0 through 9.

  However, because PL/SQL converts them to integers, you can use them in arithmetic expressions.

- A character literal with zero characters has the value `NULL` and is called a **null string**.

  However, this `NULL` value is not the `BOOLEAN` value `NULL`.

- An **ordinary character literal** is composed of characters in the **database character set**.

  For information about the database character set, see *Oracle Database Globalization Support Guide*.

- A **national character literal** is composed of characters in the **national character set**.

  For information about the national character set, see *Oracle Database Globalization Support Guide*.

Comments

The PL/SQL compiler ignores comments. Their purpose is to help other application developers understand your source text. 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 13-34

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 2–6 has three single-line comments.

Example 2–6 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 2–7 has two multiline comments. (The SQL function `TO_CHAR` returns the character equivalent of its argument. For more information about `TO_CHAR`, see *Oracle Database SQL Language Reference*.)

Example 2–7 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;
*/
```

Whitespace Characters Between Lexical Units

You can put whitespace characters between lexical units, which often makes your source text easier to read, as Example 2–8 shows.

Example 2–8 Whitespace Characters Improving Source Text 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;
/
```

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

For information about declaring objects other than variables and constants, see the syntax of *declare_section* in "Block" on page 13-9.

Variable Declarations

A variable declaration always specifies the name and data type of the variable. For most data types, a variable declaration can also specify an initial value.

The variable name must be a valid user-defined identifier (see "User-Defined Identifiers" on page 2-5).

The data type can be any PL/SQL data type. The PL/SQL data types include the SQL data types. A data type is either scalar (without internal components) or composite (with internal components).

Example 2–9 declares several variables with scalar data types.

Example 2–9 Scalar 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;
/
```

See Also:

- "Scalar Variable Declaration" on page 13-123 for scalar variable declaration syntax
- Chapter 3, "PL/SQL Data Types" for information about scalar data types
- Chapter 5, "PL/SQL Collections and Records," for information about composite data types and variables

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.)

Example 2–10 declares three constants with scalar data types.

Example 2–10 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;
/
```

See Also: "Constant Declaration" on page 13-36 for constant declaration syntax

Initial Values of Variables and Constants

In a variable declaration, the initial value is optional unless you specify the NOT NULL constraint (for details, see "NOT NULL Constraint" on page 2-13). In a constant declaration, the initial value is required (and the constant can never have a different value).

If the declaration is in a block or subprogram, the initial value is assigned to the variable or constant every time control passes to the block or subprogram. If the declaration is in a package specification, the initial value is assigned to the variable or constant 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 2–11 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 2–11 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 2–12, the variable counter has the initial value NULL, by default. As the example shows (using the "IS [NOT] NULL Operator" on page 2-34) NULL is different from zero.

Example 2–12 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

You can impose the NOT NULL constraint on a scalar variable or constant (or scalar component of a composite variable or constant). The NOT NULL constraint prevents

assigning a null value to the item. The item can acquire this constraint either implicitly (from its data type) or explicitly.

A scalar variable declaration that specifies NOT NULL, either implicitly or explicitly, must assign an initial value to the variable (because the default initial value for a scalar variable is NULL).

In Example 2–13, the variable acct_id acquires the NOT NULL constraint explicitly, and the variables a, b, and c acquire it from their data types.

Example 2–13 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 2–14, all variables are initialized to NULL.

Example 2–14 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 2-34.

%TYPE Attribute

The %TYPE attribute lets you declare a data item of the same data type as a previously declared variable or column (without knowing what that type is). If the declaration of the referenced item changes, then the declaration of the referencing item changes accordingly.

The syntax of the declaration is:

```
referencing_item referenced_item%TYPE;
```

For the kinds of items that can be referencing and referenced items, see "%TYPE Attribute" on page 13-134.

The referencing item inherits the following from the referenced item:

- Data type and size
- Constraints (unless the referenced item is a 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 column is:

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

In Example 2–15, the variable surname inherits the data type and size of the 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 2–15 Declaring Variable of Same Type as Column

```
DECLARE
  surname  employees.last_name%TYPE;
BEGIN
  DBMS_OUTPUT.PUT_LINE('surname=' || surname);
END;
/
```

Result:

```
surname=
```

In Example 2–16, the variable surname inherits the data type, size, and NOT NULL constraint of the variable name. Because surname does not inherit the initial value of name, its declaration needs an initial value (which cannot exceed 25 characters).

Example 2–16 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
```

> **See Also:** "%ROWTYPE Attribute" on page 5-42, which lets you declare a record variable that represents either a full or partial row of a database table or view

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 2-16).

If the identifier names an object 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 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`, explained in *Oracle Database SQL Language Reference*.

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

Note: You can reference identifiers declared in the packages `STANDARD` and `DBMS_STANDARD` without qualifying them with the package names, unless you have declared a local identifier with the same name (see "Scope and Visibility of Identifiers" on page 2-16).

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 2–17 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 2–17 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 2–18 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 2–18 Qualifying Redeclared Global Identifier with 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 2–19, 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 2–19 Qualifying Identifier with Subprogram Name

```
CREATE OR REPLACE PROCEDURE check_credit (credit_limit NUMBER) AS
  rating NUMBER := 3;

  FUNCTION check_rating RETURN BOOLEAN IS
    rating  NUMBER := 1;
    over_limit  BOOLEAN;
  BEGIN
    IF check_credit.rating <= credit_limit THEN  -- reference global variable
      over_limit := FALSE;
    ELSE
      over_limit := TRUE;
      rating := credit_limit;                    -- reference 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(credit_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 2–20 shows.

Example 2–20 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 2–21 shows.

Example 2–21 Declaring Same Identifier in 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 2–22, `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 2–22 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 2–23 has two labels for the outer block, `compute_ratio` and `another_label`. The second label appears 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 2–23 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 =
3.142857142857142857142857142857142857142857
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.

The variable and the value must have compatible data types. One data type is **compatible** with another data type if it can be implicitly converted to that type. For information about implicit data conversion, see *Oracle Database SQL Language Reference*.

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

See Also:

- "Assigning Values to Collection Variables" on page 5-14
- "Assigning Values to Record Variables" on page 5-44
- "FETCH Statement" on page 13-71

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 13-61.

Example 2–24 declares several variables (specifying initial values for some) and then uses assignment statements to assign the values of expressions to them.

Example 2–24 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 13-125.

Example 2–25 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 2–25 Assigning Value to Variable with SELECT INTO Statement

```
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 = 2646
```

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 "Subprogram Parameters" on page 8-9.

Example 2–26 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 2–26 Assigning Value to Variable as IN OUT Subprogram Parameter

```
DECLARE
  emp_salary  NUMBER(8,2);

  PROCEDURE adjust_salary (
    emp        NUMBER,
    sal IN OUT NUMBER,
    adjustment NUMBER
  ) IS
  BEGIN
    sal := sal + adjustment;
  END;

BEGIN
  SELECT salary INTO emp_salary
```

```
  FROM employees
  WHERE employee_id = 100;

  DBMS_OUTPUT.PUT_LINE
   ('Before invoking procedure, emp_salary: ' || emp_salary);

  adjust_salary (100, emp_salary, 1000);

  DBMS_OUTPUT.PUT_LINE
   ('After invoking procedure, emp_salary: ' || emp_salary);
END;
/
```

Result:

```
Before invoking procedure, emp_salary: 24000
After invoking procedure, emp_salary: 25000
```

Assigning Values to BOOLEAN Variables

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

Example 2–27 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 2–27 Assigning Value to BOOLEAN Variable

```
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 "BOOLEAN Data Type" on page 3-7.

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 invocation, or placeholder—or another expression. Therefore, expressions can be arbitrarily complex. For expression syntax, see "Expression" on page 13-61.

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. The data type of that result is the data type of the expression.

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 2–28 shows.

Example 2–28 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 2–29 shows.

Example 2–29 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 13-63.

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 2–3 shows operator precedence from highest to lowest. Operators with equal precedence are evaluated in no particular order.

Table 2–3 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 2–30.

Example 2–30 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 2–31, 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 2–31 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 2–32, where the parentheses do not affect evaluation order.

Example 2–32 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 2–33 shows the effect of operator precedence and parentheses in several more complex expressions.

Example 2–33 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 2–4. AND and OR are binary operators; NOT is a unary operator.

Table 2–4 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

Example 2–34 creates a procedure, print_boolean, that prints the value of a BOOLEAN variable. The procedure uses the "IS [NOT] NULL Operator" on page 2-34. Several examples in this chapter invoke print_boolean.

Example 2–34 Procedure Prints BOOLEAN Variable

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

As Table 2–4 and Example 2–35 show, AND returns TRUE if and only if both operands are TRUE.

Example 2–35 AND Operator

```
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 2–4 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 2–4 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 2–5 summarizes the relational operators.

Table 2–5 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 "Value Comparisons" on page 3-6.

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 2–12
- 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 BOOLEAN literal, 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 2–5. For the complete syntax of a BOOLEAN expression, see "boolean_expression ::=" on page 13-62.

Typically, you use BOOLEAN expressions as conditions in control statements (explained 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

```
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 13-65 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 13-65 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 ... 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)
- 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

- These miscellaneous functions:
  - CUBE_TABLE
  - DATAOBJ_TO_PARTITION
  - LNNVL
  - NVL2
  - 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 error-reporting functions, SQLCODE and SQLERRM, for use in PL/SQL exception-handling code. For their descriptions, see "SQLCODE Function" on page 13-131 and "SQLERRM Function" on page 13-132.

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

Pragmas

A **pragma** is an instruction to the compiler that it processes at compile time. For information about pragmas, see:

- "AUTONOMOUS_TRANSACTION Pragma" on page 13-6
- "EXCEPTION_INIT Pragma" on page 13-46
- "INLINE Pragma" on page 13-94
- "RESTRICT_REFERENCES Pragma" on page 13-114
- "SERIALLY_REUSABLE Pragma" on page 13-130

Conditional Compilation

Conditional compilation lets you customize the functionality of a PL/SQL application without removing source text. 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 10*g* Release 1 (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 `$` and *`plsql_identifier`*. For information about *`plsql_identifier`*, see "Identifiers" on page 2-4. The character `$` 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 2-47. The *text* can be anything, but typically, it is either a statement (see "statement ::=" on page 13-13) or an error directive (explained in "Error Directives" on page 2-43).

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 compiled, 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 compiled; otherwise, no text is compiled.

For examples of selection directives, see "Conditional Compilation Examples" on page 2-49.

> **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 2-47.

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 2-4.

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 1–2.

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 = 99
$$PLSQL_CODE_TYPE = INTERPRETED
$$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
  DBMS_OUTPUT.PUT_LINE($$Some_Flag);
  DBMS_OUTPUT.PUT_LINE($$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 1-10.

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 PL/SQL issues the warning `PLW-6003` and substitutes `NULL` 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 Text 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 2-23 for general information about expressions

PLS_INTEGER Static Expressions `PLS_INTEGER` static expressions are:

- `PLS_INTEGER` literals

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

- PLS_INTEGER static constants

  For information about static constants, see "Static Constants" on page 2-48.

- NULL

  See Also: "PLS_INTEGER and BINARY_INTEGER Data Types" on page 3-8 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 2-48.

- 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 2-46.

- 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 2-37.

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

  For information about static expressions, see "Static Expressions" on page 2-46.

  See Also: "BOOLEAN Data Type" on page 3-7 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 literals, see "Literals" on page 2-8.

- 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 2-46.

> **See Also:** "CHAR and VARCHAR2 Variables" on page 3-3 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 recompilation (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 2-45 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 2-12 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 (10.1) 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 Oracle Database 10*g* Release 2 (10.2); otherwise, it displays a message saying that

the version and release are supported and uses a COMMIT statement that became available at Oracle Database 10*g* Release 2 (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 := 3.14159265358979323846264338327950288420;
    my_e  := 2.71828182845904523536028747135266249775;
  $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;
/
```

Result:

```
PACKAGE my_pkg AS
SUBTYPE my_real IS
BINARY_DOUBLE;
my_pi my_real;
my_e my_real;
END my_pkg;

Call completed.
```

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 Textsource text

```
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 14-68). 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 PL/SQL constant, variable, parameter, and function return value has a **data type** that determines its storage format and its valid values and operations.

This chapter explains **scalar data types**, which store values with no internal components. For information about **composite data types**, see Chapter 5, "PL/SQL Collections and Records".

A scalar data type can have subtypes. A **subtype** is a data type that is a subset of another data type, which is its **base type**. A subtype has the same valid operations as its base type. A data type and its subtypes comprise a **data type family**.

PL/SQL predefines many types and subtypes in the package `STANDARD` and lets you define your own subtypes.

The PL/SQL scalar data types are:

- The SQL data types
- `BOOLEAN`
- `PLS_INTEGER`
- `BINARY_INTEGER`
- `REF CURSOR`, explained in "Cursor Variables" on page 6-28
- User-defined subtypes

Topics

- SQL Data Types
- BOOLEAN Data Type
- PLS_INTEGER and BINARY_INTEGER Data Types
- SIMPLE_INTEGER Subtype of PLS_INTEGER
- User-Defined PL/SQL Subtypes

See Also:

- "CREATE TYPE Statement" on page 14-68 for information about creating schema-level user-defined data types
- Appendix E, "PL/SQL Predefined Data Types" for the predefined PL/SQL data types and subtypes, grouped by data type family

SQL Data Types

The PL/SQL data types include the SQL data types. For information about the SQL data types, see *Oracle Database SQL Language Reference*—all information there about data types and subtypes, data type comparison rules, data conversion, literals, and format models applies to both SQL and PL/SQL, except as noted here:

- Different Maximum Sizes
- Additional PL/SQL Constants for BINARY_FLOAT and BINARY_DOUBLE
- Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE

Unlike SQL, PL/SQL lets you declare variables, to which the following topics apply:

- CHAR and VARCHAR2 Variables
- LONG and LONG RAW Variables
- ROWID and UROWID Variables

Different Maximum Sizes

The SQL data types listed in Table 3–1 have different maximum sizes in PL/SQL and SQL.

Table 3–1 Data Types with Different Maximum Sizes in PL/SQL and SQL

Data Type	Maximum Size in PL/SQL	Maximum Size in SQL
`CHAR`[1]	32,676 bytes	2,000 bytes
`NCHAR`[1]	32,676 bytes	2,000 bytes
`RAW`[1]	32,676 bytes	2,000 bytes
`VARCHAR2`[1]	32,676 bytes	4,000 bytes
`NVARCHAR2`[1]	32,676 bytes	4,000 bytes
`LONG`[2]	32,670 bytes	2 gigabytes (GB) - 1
`LONG RAW`[2]	32,670 bytes	2 GB
`BLOB`	128 terabytes (TB)	(4 GB - 1) * *`database_block_size`*
`CLOB`	128 TB	(4 GB - 1) * *`database_block_size`*
`NCLOB`	128 TB	(4 GB - 1) * *`database_block_size`*

[1] When specifying the maximum size of a value of this data type in PL/SQL, use an integer literal (not a constant or variable) whose value is in the range from 1 through 32,676.

[2] Supported only for backward compatibility with existing applications.

Additional PL/SQL Constants for BINARY_FLOAT and BINARY_DOUBLE

The SQL data types `BINARY_FLOAT` and `BINARY_DOUBLE` represent single-precision and double-precision IEEE 754-format floating-point numbers, respectively.

`BINARY_FLOAT` and `BINARY_DOUBLE` computations do not raise exceptions, so you must check the values that they produce for conditions such as overflow and underflow by comparing them to predefined constants (for examples, see *Oracle Database SQL Language Reference*). PL/SQL has more of these constants than SQL does.

Table 3–2 lists and describes the predefined PL/SQL constants for `BINARY_FLOAT` and `BINARY_DOUBLE`, and identifies those that SQL also defines.

Table 3–2 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	Maximum normal BINARY_FLOAT value
BINARY_FLOAT_MIN_NORMAL	Minimum normal BINARY_FLOAT value
BINARY_FLOAT_MAX_SUBNORMAL	Maximum subnormal BINARY_FLOAT value
BINARY_FLOAT_MIN_SUBNORMAL	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] SQL also predefines this constant.

Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE

PL/SQL predefines these subtypes:

- SIMPLE_FLOAT, a subtype of SQL data type BINARY_FLOAT
- SIMPLE_DOUBLE, a subtype of SQL data type BINARY_DOUBLE

Each subtype has the same range as its base type and has a NOT NULL constraint (explained in "NOT NULL Constraint" on page 2-13).

If you know that a variable will never have the value NULL, declare it as SIMPLE_FLOAT or SIMPLE_DOUBLE, rather than BINARY_FLOAT or BINARY_DOUBLE. Without the overhead of checking for nullness, the subtypes provide significantly better performance than their base types. The performance improvement is greater with PLSQL_CODE_TYPE='NATIVE' than with PLSQL_CODE_TYPE='INTERPRETED' (for more information, see "Use Data Types that Use Hardware Arithmetic" on page 12-9).

CHAR and VARCHAR2 Variables

Topics

- Assigning or Inserting Too-Long Values
- Declaring Variables for Multibyte Characters
- Differences Between CHAR and VARCHAR2 Data Types

Assigning or Inserting Too-Long Values

If the value that you assign to a character variable is longer than the maximum size of the variable, an error occurs. For example:

```
DECLARE
```

```
  c VARCHAR2(3 CHAR);
BEGIN
  c := 'abc  ';
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 4
```

Similarly, if you insert a character variable into a column, and the value of the variable is longer than the defined width of the column, an error occurs. For example:

```
DROP TABLE t;
CREATE TABLE t (c CHAR(3 CHAR));

DECLARE
  s VARCHAR2(5 CHAR) := 'abc  ';
BEGIN
  INSERT INTO t(c) VALUES(s);
END;
/
```

Result:

```
BEGIN
*
ERROR at line 1:
ORA-12899: value too large for column "HR"."T"."C" (actual: 5, maximum: 3)
ORA-06512: at line 4
```

To strip trailing blanks from a character value before assigning it to a variable or inserting it into a column, use the `RTRIM` function, explained in *Oracle Database SQL Language Reference*. For example:

```
DECLARE
  c VARCHAR2(3 CHAR);
BEGIN
  c := RTRIM('abc  ');
  INSERT INTO t(c) VALUES(RTRIM('abc  '));
END;
/
```

Result:

```
PL/SQL procedure successfully completed.
```

Declaring Variables for Multibyte Characters

The maximum *size* of a `CHAR` or `VARCHAR2` variable is 32,767 bytes, whether you specify the maximum size in characters or bytes. The maximum *number of characters* in the variable depends on the character set. For a single-byte character set, the maximum number of characters is 32,767; for an *n*-byte character set, it is 32,767/*n* characters, rounded down to the nearest integer.

When declaring a `CHAR` or `VARCHAR2` variable to hold values from a multibyte character set, specify the maximum size in characters to ensure that a `CHAR(n)` or `VARCHAR2(n)` variable can store *n* multibyte characters.

> **See Also:** *Oracle Database Globalization Support Guide* for information about Oracle Database character set support

Differences Between CHAR and VARCHAR2 Data Types

CHAR and VARCHAR2 data types differ in:

- Predefined Subtypes
- Memory Allocation
- Blank-Padding
- Value Comparisons

Predefined Subtypes The CHAR data type has one predefined subtype in both PL/SQL and SQL—CHARACTER.

The VARCHAR2 data type has one predefined subtype in both PL/SQL and SQL, VARCHAR, and an additional predefined subtype in PL/SQL, STRING.

Each subtype has the same range of values as its base type.

> **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 a CHAR variable, PL/SQL allocates at compile time enough memory for the maximum size.

For a VARCHAR2 variable, memory allocation depends on maximum size:

- If the maximum size is less than 4,000 bytes, PL/SQL allocates at compile time enough memory for the maximum size.
- If the maximum size is 4,000 bytes or more, PL/SQL allocates at run time enough memory for the actual value.

For example, suppose that variables a and b are declared as follows:

```
a VARCHAR2(3999);
b VARCHAR2(4000);
```

If you assign the same 500-byte value to both variables, PL/SQL allocates 3,999 bytes for a at compile time and 500 bytes for b at run time.

Thus, PL/SQL optimizes smaller VARCHAR2 variables for performance and larger ones for efficient memory use.

Blank-Padding Consider these situations:

- The value that you assign to a variable is shorter than the maximum size of the variable.
- The value that you insert into a column is shorter than the defined width of the column.
- The value that you retrieve from a column into a variable is shorter than the maximum size 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.

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.

In Example 3–1, both the CHAR variable and the VARCHAR2 variable have the maximum size of 10 characters. Each variable receives a five-character value with one trailing blank. The value assigned to the CHAR variable is blank-padded to 10 characters, and you cannot tell that one of the six trailing blanks in the resulting value was in the original value. The value assigned to the VARCHAR2 variable is not changed, and you can see that it has one trailing blank.

Example 3–1 CHAR and VARCHAR2 Blank-Padding Difference

```
DECLARE
  first_name  CHAR(10 CHAR);
  last_name   VARCHAR2(10 CHAR);
BEGIN
  first_name := 'John ';
  last_name  := 'Chen ';

  DBMS_OUTPUT.PUT_LINE('*' || first_name || '*');
  DBMS_OUTPUT.PUT_LINE('*' || last_name || '*');
END;
/
```

Result:

```
*John      *
*Chen *
```

Value Comparisons The SQL rules for comparing character values apply to PL/SQL character variables. Whenever one or both values in the comparison have the data type VARCHAR2 or NVARCHAR2, nonpadded comparison semantics apply; otherwise, blank-padded semantics apply. For more information, see *Oracle Database SQL Language Reference*.

LONG and LONG RAW Variables

Note: Oracle supports the LONG and LONG RAW data types only for backward compatibility with existing applications. For new applications:

- Instead of LONG, use VARCHAR2(32760), BLOB, CLOB or NCLOB.
- Instead of LONG RAW, use BLOB.

You can insert any LONG value into a LONG column. You can insert any LONG RAW value into a LONG RAW column. You cannot retrieve a value longer than 32,760 bytes from a LONG or LONG RAW column into a LONG or LONG RAW variable.

You can insert any CHAR or VARCHAR2 value into a LONG column. You cannot retrieve a value longer than 32,767 bytes from a LONG column into a CHAR or VARCHAR2 variable.

You can insert any RAW value into a LONG RAW column. You cannot retrieve a value longer than 32,767 bytes from a LONG RAW column into a RAW variable.

See Also: "Trigger LONG and LONG RAW Data Type Restrictions" on page 9-37 for restrictions on LONG and LONG RAW data types in triggers

ROWID and UROWID Variables

When you retrieve a rowid into a ROWID variable, use the ROWIDTOCHAR function to convert the binary value to a character value. For information about this function, see *Oracle Database SQL Language Reference*.

To convert the value of a ROWID variable to a rowid, use the CHARTOROWID function, explained in *Oracle Database SQL Language Reference*. If the value does not represent a valid rowid, PL/SQL raises the predefined exception SYS_INVALID_ROWID.

To retrieve a rowid into a UROWID variable, or to convert the value of a UROWID variable to a rowid, use an assignment statement; conversion is implicit.

Note: UROWID is a more versatile data type than ROWID, because it is compatible with both logical and physical rowids.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about the DBMS_ROWID package, whose subprograms let you create and return information about ROWID values (but not UROWID values)

BOOLEAN Data Type

The PL/SQL data type BOOLEAN stores **logical values**, which are the Boolean values TRUE and FALSE and the value NULL. NULL represents an unknown value.

The syntax for declaring an BOOLEAN variable is:

```
variable_name BOOLEAN
```

The only value that you can assign to a BOOLEAN variable is a BOOLEAN expression. For details, see "BOOLEAN Expressions" on page 2-37.

Because SQL has no data type equivalent to BOOLEAN, you cannot:

- Assign a BOOLEAN value to a database table column
- Select or fetch the value of a database table column into a BOOLEAN variable
- Use a BOOLEAN value in a SQL statement, SQL function, or PL/SQL function invoked from a SQL statement

You cannot pass a BOOLEAN value to the DBMS_OUTPUT.PUT or DBMS_OUTPUT.PUTLINE subprogram. To print a BOOLEAN value, use an IF or CASE statement to translate it to a character value (for information about these statements, see "Conditional Selection Statements" on page 4-1).

In Example 3–2, the procedure accepts a BOOLEAN parameter and uses a CASE statement to print Unknown if the value of the parameter is NULL, Yes if it is TRUE, and No if it is FALSE.

Example 3–2 Printing BOOLEAN Values

```
CREATE PROCEDURE print_boolean (b BOOLEAN)
AS
BEGIN
```

```
  CASE
    WHEN b IS NULL THEN DBMS_OUTPUT.PUT_LINE('Unknown');
    WHEN b THEN DBMS_OUTPUT.PUT_LINE('Yes');
    WHEN NOT b THEN DBMS_OUTPUT.PUT_LINE('No');
  END CASE;
END;
/
BEGIN
  print_boolean(TRUE);
  print_boolean(FALSE);
  print_boolean(NULL);
END;
/
```

Result:

```
Yes
No
Unknown
```

> **See Also:** Example 2–35, which creates a `print_boolean` procedure that uses an `IF` statement.

PLS_INTEGER and BINARY_INTEGER Data Types

The PL/SQL data types `PLS_INTEGER` and `BINARY_INTEGER` 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 in its range.

Topics

- Preventing PLS_INTEGER Overflow
- Predefined PLS_INTEGER Subtypes
- SIMPLE_INTEGER Subtype of PLS_INTEGER

Preventing PLS_INTEGER Overflow

A calculation with two `PLS_INTEGER` values that overflows the `PLS_INTEGER` range raises an overflow exception, even if you assign the result to a `NUMBER` data type (as in Example 3–3). For calculations outside the `PLS_INTEGER` range, use `INTEGER`, a predefined subtype of the `NUMBER` data type (as in Example 3–4).

Example 3–3 PLS_INTEGER Calculation Raises Overflow Exception

```
DECLARE
  p1 PLS_INTEGER := 2147483647;
  p2 PLS_INTEGER := 1;
```

```
  n NUMBER;
BEGIN
  n := p1 + p2;
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-01426: numeric overflow
ORA-06512: at line 6
```

Example 3–4 Preventing Example 3–3 Overflow

```
DECLARE
  p1 PLS_INTEGER := 2147483647;
  p2 INTEGER := 1;
  n NUMBER;
BEGIN
  n := p1 + p2;
END;
/
```

Result:

```
PL/SQL procedure successfully completed.
```

Predefined PLS_INTEGER Subtypes

Table 3–3 lists the predefined subtypes of the PLS_INTEGER data type and describes the data they store.

Table 3–3 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[1]
POSITIVE	Positive PLS_INTEGER value
POSITIVEN	Positive PLS_INTEGER value with NOT NULL constraint[1]
SIGNTYPE	PLS_INTEGER value -1, 0, or 1 (useful for programming tri-state logic)
SIMPLE_INTEGER	PLS_INTEGER value with NOT NULL constraint. For more information, see "SIMPLE_INTEGER Subtype of PLS_INTEGER" on page 3-10.

[1] For information about the NOT NULL constraint, see "NOT NULL Constraint" on page 2-13.

PLS_INTEGER and its subtypes can be implicitly converted to these data types:

- CHAR
- VARCHAR2
- NUMBER
- LONG

All of the preceding data types except LONG, and all PLS_INTEGER subtypes, can be implicitly converted to PLS_INTEGER.

A PLS_INTEGER value can be implicitly converted to a PLS_INTEGER subtype only if the value does not violate a constraint of the subtype. For example, casting the PLS_INTEGER value NULL to the SIMPLE_INTEGER subtype raises an exception, as Example 3–5 shows.

Example 3–5 Violating Constraint of SIMPLE_INTEGER Subtype

```
DECLARE
  a SIMPLE_INTEGER := 1;
  b PLS_INTEGER := NULL;
BEGIN
  a := b;
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 5
```

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 and has a NOT NULL constraint (explained in "NOT NULL Constraint" on page 2-13). It differs significantly from PLS_INTEGER in its overflow semantics.

If you know that a variable will never have the value NULL or need overflow checking, declare it as SIMPLE_INTEGER rather than PLS_INTEGER. Without the overhead of checking for nullness and overflow, SIMPLE_INTEGER performs significantly better than PLS_INTEGER.

Topics

- SIMPLE_INTEGER Overflow Semantics
- Expressions with Both SIMPLE_INTEGER and Other Operands
- Integer Literals in SIMPLE_INTEGER Range

SIMPLE_INTEGER Overflow Semantics

If and only if all operands in an expression 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

For example, this block runs without errors:

```
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

PL/SQL procedure successfully completed.
```

Expressions with Both SIMPLE_INTEGER and Other Operands

If an expression has both `SIMPLE_INTEGER` and other operands, PL/SQL implicitly converts the `SIMPLE_INTEGER` values to `PLS_INTEGER NOT NULL`.

The PL/SQL compiler issues a warning when `SIMPLE_INTEGER` and other values are mixed in a way that might negatively impact performance by inhibiting some optimizations.

Integer Literals in SIMPLE_INTEGER Range

Integer literals in the `SIMPLE_INTEGER` range have the data type `SIMPLE_INTEGER`. However, 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`.

User-Defined PL/SQL Subtypes

PL/SQL lets you define your own subtypes. The base type can be any scalar PL/SQL type, including a previously defined user-defined subtype.

> **Note:** The information in this topic applies to both user-defined subtypes and the predefined subtypes listed in Appendix E, "PL/SQL Predefined Data Types".

Subtypes can:

- Provide compatibility with ANSI/ISO data types
- Show the intended use of data items of that type
- Detect out-of-range values

Topics

- Unconstrained Subtypes
- Constrained Subtypes

- Subtypes with Base Types in Same Data Type Family

Unconstrained Subtypes

An **unconstrained subtype** has the same set of values as its base type, so it is only another name for the base type. Therefore, unconstrained subtypes of the same base type are interchangeable with each other and with the base type. No data type conversion occurs.

To define an unconstrained subtype, use this syntax:

```
SUBTYPE subtype_name IS base_type
```

For information about *subtype_name* and *base_type*, see **subtype_definition** on page 13-14.

An example of an unconstrained subtype, which PL/SQL predefines for compatibility with ANSI, is:

```
SUBTYPE "DOUBLE PRECISION" IS FLOAT
```

In Example 3–6, the unconstrained subtypes `Balance` and `Counter` show the intended uses of data items of their types.

Example 3–6 User-Defined Unconstrained Subtypes Show Intended Use

```
DECLARE
  SUBTYPE Balance IS NUMBER;

  checking_account        Balance(6,2);
  savings_account         Balance(8,2);
  certificate_of_deposit  Balance(8,2);
  max_insured  CONSTANT   Balance(8,2) := 250000.00;

  SUBTYPE Counter IS NATURAL;

  accounts     Counter := 1;
  deposits     Counter := 0;
  withdrawals  Counter := 0;
  overdrafts   Counter := 0;

  PROCEDURE deposit (
    account  IN OUT Balance,
    amount   IN     Balance
  ) IS
  BEGIN
    account  := account + amount;
    deposits := deposits + 1;
  END;

BEGIN
  NULL;
END;
/
```

Constrained Subtypes

A **constrained subtype** has only a subset of the values of its base type.

If the base type lets you specify size, precision and scale, or a range of values, then you can specify them for its subtypes. The subtype definition syntax is:

```
SUBTYPE subtype_name IS base_type
  { precision [, scale ] | RANGE low_value .. high_value } [ NOT NULL ]
```

Otherwise, the only constraint that you can put on its subtypes is NOT NULL (described in "NOT NULL Constraint" on page 2-13):

```
SUBTYPE subtype_name IS base_type [ NOT NULL ]
```

> **See Also:** Syntax diagram "subtype_definition ::=" on page 13-11 and semantic description "subtype_definition" on page 13-14.

In Example 3–7, the constrained subtype Balance detects out-of-range values.

Example 3–7 User-Defined Constrained Subtype Detects Out-of-Range Values

```
DECLARE
  SUBTYPE Balance IS NUMBER(8,2);

  checking_account  Balance;
  savings_account   Balance;

BEGIN
  checking_account := 2000.00;
  savings_account  := 1000000.00;
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: number precision too large
ORA-06512: at line 9
```

A constrained subtype can be implicitly converted to its base type, but the base type can be implicitly converted to the constrained subtype only if the value does not violate a constraint of the subtype (see Example 3–5).

A constrained subtype can be implicitly converted to another constrained subtype with the same base type only if the source value does not violate a constraint of the target subtype.

In Example 3–8, the three constrained subtypes have the same base type. The first two subtypes can be implicitly converted to the third subtype, but not to each other.

Example 3–8 Implicit Conversion Between Constrained Subtypes with Same Base Type

```
DECLARE
  SUBTYPE Digit        IS PLS_INTEGER RANGE 0..9;
  SUBTYPE Double_digit IS PLS_INTEGER RANGE 10..99;
  SUBTYPE Under_100    IS PLS_INTEGER RANGE 0..99;

  d   Digit        :=  4;
  dd  Double_digit := 35;
  u   Under_100;
BEGIN
  u := d;   -- Succeeds; Under_100 range includes Digit range
  u := dd;  -- Succeeds; Under_100 range includes Double_digit range
  dd := d;  -- Raises error; Double_digit range does not include Digit range
END;
```

```
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error
ORA-06512: at line 12
```

> **See Also:** "Formal and Actual Subprogram Parameters" on page 8-9 for information about subprogram parameters of constrained data types

Subtypes with Base Types in Same Data Type Family

If two subtypes have different base types in the same data type family, then one subtype can be implicitly converted to the other only if the source value does not violate a constraint of the target subtype. (For the predefined PL/SQL data types and subtypes, grouped by data type family, see Appendix E, "PL/SQL Predefined Data Types".)

In Example 3–9, the subtypes `Word` and `Text` have different base types in the same data type family. The first assignment statement implicitly converts a `Word` value to `Text`. The second assignment statement implicitly converts a `Text` value to `Word`. The third assignment statement cannot implicitly convert the `Text` value to `Word`, because the value is too long.

Example 3–9 Implicit Conversion Between Subtypes with Base Types in Same Family

```
DECLARE
  SUBTYPE Word IS CHAR(6);
  SUBTYPE Text IS VARCHAR2(15);

  verb       Word := 'run';
  sentence1  Text;
  sentence2  Text := 'Hurry!';
  sentence3  Text := 'See Tom run.';

BEGIN
  sentence1 := verb;  -- 3-character value, 15-character limit
  verb := sentence2;  -- 5-character value, 6-character limit
  verb := sentence3;  -- 12-character value, 6-character limit
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
ORA-06502: PL/SQL: numeric or value error: character string buffer too small
ORA-06512: at line 13
```

4

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 basic `LOOP`, `FOR LOOP`, and `WHILE 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.

Topics

- Conditional Selection Statements
- LOOP Statements
- Sequential Control Statements

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 13-89.)

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 13-89.)

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 13-89.)

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 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 13-20.)

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 ... 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 13-20.)

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 values. The loop statements are:

- Basic `LOOP`
- `FOR LOOP`
- Cursor `FOR LOOP`
- `WHILE 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. Oracle recommends using these statements instead of 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. For information about exceptions, see "Overview of Exception Handling" on page 11-4.)

`LOOP` statements can be labeled, and `LOOP` statements can be nested. Labels are recommended for nested loops to improve readability. You must ensure that the label in the `END LOOP` statement matches the label at the beginning of the same loop statement (the compiler does not check).

Topics

- Basic LOOP Statement
- EXIT Statement
- EXIT WHEN Statement
- CONTINUE Statement
- CONTINUE WHEN Statement
- FOR LOOP Statement
- WHILE LOOP Statement

For information about the cursor `FOR LOOP`, see "Query Result Set Processing With Cursor FOR LOOP Statements" on page 6-24.

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, a statement or raised exception must exit the loop.

See Also: "Basic LOOP Statement" on page 13-7

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 13-55

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 13-55

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 13-38

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 13-38

FOR LOOP Statement

The `FOR LOOP` statement runs one or more statements while the loop 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`*. If *`lower_bound`* is greater than *`upper_bound`*, then the *`statements`* never run.

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`*. If *`upper_bound`* is less than *`lower_bound`*, then the *`statements`* never run.

An `EXIT`, `EXIT WHEN`, `CONTINUE`, or `CONTINUE WHEN` in the *`statements`* can cause the loop or the current iteration of the loop to end early.

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 "Query Result Set Processing With Cursor FOR LOOP Statements" on page 6-24.

See Also: "FOR LOOP Statement" on page 13-74

In Example 4–14, *`index`* is `i`, *`lower_bound`* is 1, and *`upper_bound`* is 3. The loop prints the numbers from 1 to 3.

Example 4–14 FOR LOOP Statements

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('lower_bound < upper_bound');

  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('lower_bound = upper_bound');

  FOR i IN 2..2 LOOP
```

```
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('lower_bound > upper_bound');

  FOR i IN 3..1 LOOP
    DBMS_OUTPUT.PUT_LINE (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 4–15 is the reverse of the one in Example 4–14: It prints the numbers from 3 to 1.

Example 4–15 Reverse FOR LOOP Statements

```
BEGIN
  DBMS_OUTPUT.PUT_LINE ('upper_bound > lower_bound');

  FOR i IN REVERSE 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('upper_bound = lower_bound');

  FOR i IN REVERSE 2..2 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE ('upper_bound < lower_bound');

  FOR i IN REVERSE 3..1 LOOP
    DBMS_OUTPUT.PUT_LINE (i);
  END LOOP;
END;
/
```

Result:

```
upper_bound > lower_bound
3
2
1
upper_bound = lower_bound
2
upper_bound > lower_bound
```

In some languages, the FOR LOOP has a STEP clause that lets you specify a loop index increment other than 1. To simulate the STEP clause in PL/SQL, multiply each reference to the loop index by the desired increment.

In Example 4–16, the FOR LOOP effectively increments the index by five.

Example 4–16 Simulating STEP Clause in FOR LOOP Statement

```
DECLARE
  step  PLS_INTEGER := 5;
BEGIN
  FOR i IN 1..3 LOOP
    DBMS_OUTPUT.PUT_LINE (i*step);
  END LOOP;
END;
/
```

Result:

```
5
10
15
```

Topics

- FOR LOOP Index
- Lower Bound and Upper Bound
- EXIT WHEN or CONTINUE WHEN 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 Outside Statement References FOR LOOP Statement 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 Statement Index with Same Name as 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 Statement References 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 lower and upper bounds of a FOR LOOP statement can be either numeric literals, numeric variables, or numeric expressions. If a bound does not have a numeric value, then PL/SQL raises the predefined exception VALUE_ERROR.

Example 4–22 FOR LOOP Statement 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;
/
```

In Example 4–23, the upper bound of the FOR LOOP statement is a variable whose value is determined at run time.

Example 4–23 Specifying FOR LOOP Statement Bounds 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;
/
```

EXIT WHEN or CONTINUE WHEN Statement in FOR LOOP Statement

Suppose that you must exit a FOR LOOP statement immediately if a certain condition arises. You can put the condition in an EXIT WHEN statement inside the FOR LOOP statement.

In Example 4–24, the FOR LOOP statement executes 10 times unless the FETCH statement inside it fails to return a row, in which case it ends immediately.

Example 4–24 EXIT WHEN Statement in FOR LOOP Statement

```
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;
/
```

Now suppose that the FOR LOOP statement that you must exit early is nested inside another FOR LOOP statement. If, when you exit the inner loop early, you also want to exit the outer loop, then label the outer loop and specify its name in the EXIT WHEN statement, as in Example 4–25.

If you want to exit the inner loop early but complete the current iteration of the outer loop, then label the outer loop and specify its name in the CONTINUE WHEN statement, as in Example 4–26.

Example 4–25 EXIT WHEN Statement in Inner FOR LOOP Statement

```
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;
/
```

Example 4–26 CONTINUE WHEN Statement in Inner FOR LOOP Statement

```
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;
      CONTINUE outer_loop WHEN c1%NOTFOUND;
      -- Process data here
    END LOOP;
  END LOOP outer_loop;

  CLOSE c1;
END;
/
```

See Also: "Overview of Exception Handling" on page 11-4 for information about exceptions, which can also cause a loop to end immediately if a certain condition arises

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 13-138.

An EXIT, EXIT WHEN, CONTINUE, or CONTINUE WHEN in the *statements* can cause the loop or the current iteration of the loop to end early.

In Example 4–27, the statements in the first WHILE LOOP statement never run, and the statements in the second WHILE LOOP statement run once.

Example 4–27 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;
```

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 13-87.

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 GOTO Statement Goes to Labeled NULL Statement

```
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 11-2.

Example 4–34 NULL Statement as Placeholder During Subprogram Creation

```
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

A **composite data type** stores values that have internal components. You can pass entire composite variables to subprograms as parameters, and you can access internal components of composite variables individually. Internal components can be either scalar or composite. You can use scalar components wherever you can use scalar variables. PL/SQL lets you define two kinds of composite data types, collection and record. You can use composite components wherever you can use composite variables of the same type.

In a **collection**, the internal components always have the same data type, and are called **elements**. You can access each element of a collection variable by its unique subscript, with this syntax: `variable_name(subscript)`. To create a collection variable, you either define a collection type and then create a variable of that type or use `%TYPE`.

In a **record**, the internal components can have different data types, and are called **fields**. You can access each field of a record variable by its name, with this syntax: `variable_name.field_name`. To create a record variable, you either define a `RECORD` type and then create a variable of that type or use `%ROWTYPE` or `%TYPE`.

You can create a collection of records, and a record that contains collections.

Collection Topics

- Collection Types
- Associative Arrays
- Varrays (Variable-Size Arrays)
- Nested Tables
- Collection Constructors
- Assigning Values to Collection Variables
- Multidimensional Collections
- Collection Comparisons
- Collection Methods
- Collection Types Defined in Package Specifications

See Also:

- "BULK COLLECT Clause" on page 12-23 for information about retrieving query results into a collection
- "Collection Variable Declaration" on page 13-25 for syntax and semantics of collection type definition and collection variable declaration

Record Topics

- Record Variables
- Assigning Values to Record Variables
- Record Comparisons
- Inserting Records into Tables
- Updating Rows with Records
- Restrictions on Record Inserts and Updates

Note: Several examples in this chapter define procedures that print their composite variables. Several of those procedures invoke this standalone stored procedure, which prints either its integer parameter (if it is not `NULL`) or the string `'NULL'`:

```
CREATE OR REPLACE PROCEDURE print (n INTEGER) IS
BEGIN
  IF n IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE(n);
  ELSE
    DBMS_OUTPUT.PUT_LINE('NULL');
  END IF;
END print;
/
```

Some examples in this chapter define functions that return values of composite types.

You can understand the examples in this chapter without completely understanding PL/SQL procedures and functions, which are explained in Chapter 8, "PL/SQL Subprograms".

Collection Types

PL/SQL has three collection types—associative array, `VARRAY` (variable-size array), and nested table. Table 5–1 summarizes their similarities and differences.

Table 5–1 PL/SQL Collection Types

Collection Type	Number of Elements	Subscript Type	Dense or Sparse	Uninitialized Status	Where Defined	Can Be ADT Attribute Data Type
Associative array (or index-by table)	Unbounded	String or integer	Either	Empty	In PL/SQL block or package	No
VARRAY (variable-size array)	Bounded	Integer	Always dense	Null	In PL/SQL block or package or at schema level	Only if defined at schema level
Nested table	Unbounded	Integer	Starts dense, can become sparse	Null	In PL/SQL block or package or at schema level	Only if defined at schema level

Number of Elements

An **unbounded collection** has no declared maximum number of elements—its maximum number of elements is the upper limit of `PLS_INTEGER`. A **bounded collection** has a declared maximum number of elements.

Dense or Sparse

A **dense collection** has no gaps between elements—every element between the first and last element is defined and has a value (the value can be `NULL` unless the element has a `NOT NULL` constraint). A **sparse collection** has gaps between elements.

Uninitialized Status

An **empty collection** exists but has no elements. To add elements to an empty collection, invoke the `EXTEND` method (described in "EXTEND Collection Method" on page 5-26).

A **null collection** (also called an **atomically null collection**) does not exist. To change a null collection to an existing collection, you must initialize it, either by making it empty or by assigning a non-`NULL` value to it (for details, see "Collection Constructors" on page 5-13 and "Assigning Values to Collection Variables" on page 5-14). You cannot use the `EXTEND` method to initialize a null collection.

Where Defined

A collection type defined in a PL/SQL block is a **local type**. It is available only in the block, and is stored in the database only if the block is in a standalone stored or package subprogram. (Standalone stored and package subprograms are explained in "Nested, Package, and Standalone Stored Subprograms" on page 8-2.)

A collection type defined in a package specification is a **public item**. You can reference it from outside the package by qualifying it with the package name (*`package_name.type_name`*). It is stored in the database until you drop the package. (Packages are explained in Chapter 10, "PL/SQL Packages.")

A collection type defined at schema level is a **standalone stored type**. You create it with the "CREATE TYPE Statement" on page 14-68. It is stored in the database until you drop it with the "DROP TYPE Statement" on page 14-98.

Note: A collection type defined in a package specification is incompatible with an identically defined local or standalone stored collection type (see Example 5–30 and Example 5–31).

Can Be ADT Attribute Data Type

To be an ADT attribute data type, a collection type must be a standalone stored collection type. For other restrictions, see **Restrictions on datatype** on page 14-74.

Translating Non-PL/SQL Composite Types to PL/SQL Composite Types

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:

Non-PL/SQL Composite Type	Equivalent PL/SQL Composite Type
Hash table	Associative array
Unordered table	Associative array
Set	Nested table
Bag	Nested table
Array	`VARRAY`

> **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 **PL/SQL table** or **index-by table**) is an unbounded set of key-value pairs. Each key is a unique subscript, used to locate the associated value with the syntax *`variable_name(subscript)`*.

The subscripts of an associative array can be either strings or integers. Subscripts are stored in sort order, not creation order. For strings, sort order is determined by the initialization parameters `NLS_SORT` and `NLS_COMP`.

Like a database table, an associative array:

- Is empty (but not null) until you populate it
- Can hold an arbitrary number of elements, which you can access without knowing their positions

Unlike a database table, an associative array:

- Does not need disk space or network operations
- Cannot be manipulated with DML statements

You cannot specify an initial value when you declare an associative array variable (therefore, you cannot declare an associative array constant). 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 defines a type of associative array indexed by string, declares a variable of that type, populates the variable with three elements, changes the value of one element, and prints the values (in sort order, not creation order). (`FIRST` and `NEXT` are collection methods, described in "Collection Methods" on page 5-21.)

Example 5–1 Associative Array Indexed by String

```
DECLARE
```

```
  -- Associative array indexed by string:

  TYPE population IS TABLE OF NUMBER  -- Associative array type
    INDEX BY VARCHAR2(64);            --  indexed by string

  city_population  population;        -- Associative array variable
  i  VARCHAR2(64);                    -- Scalar variable

BEGIN
  -- Add elements (key-value pairs) 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;  -- Get first element of array

  WHILE i IS NOT NULL LOOP
    DBMS_Output.PUT_LINE
      ('Population of ' || i || ' is ' || city_population(i));
    i := city_population.NEXT(i);  -- Get next element of array
  END LOOP;
END;
/
```

Result:

```
Population of Megalopolis is 1000000
Population of Midland is 750000
Population of Smallville is 2001
```

Example 5–2 defines a type of associative array indexed by integer and a function that returns an associative array.

Example 5–2 Function Returns Associative Array Indexed by Integer

```
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
  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;
/
```

Result:

```
Sum of the first 5 multiples of 3 is 45
```

Topics

- NLS Parameter Values Affect Associative Arrays Indexed by String
- Appropriate Uses for Associative Arrays

See Also:

- Table 5–1 for a summary of associative array characteristics
- "assoc_array_type_def ::=" on page 13-26 for the syntax of an associative array type definition

NLS Parameter Values Affect Associative Arrays Indexed by String

National Language Support (NLS) parameters such as `NLS_SORT`, `NLS_COMP`, and `NLS_DATE_FORMAT` affect associative arrays indexed by string.

Topics

- Changing NLS Parameter Values After Populating Associative Arrays
- Subscripts of Data Types Other Than VARCHAR2
- Passing Associative Arrays to Remote Databases

See Also: *Oracle Database Globalization Support Guide* for information about linguistic sort parameters

Changing NLS Parameter Values After Populating Associative Arrays

The initialization parameters `NLS_SORT` and `NLS_COMP` determine the storage order of string subscripts of an associative array. If you change the value of either parameter after populating an associative array indexed by string, then the collection methods `FIRST`, `LAST`, `NEXT`, and `PRIOR` (described in "Collection Methods" on page 5-21) might return unexpected values or raise exceptions. If you must change these parameter values during your session, restore their original values before operating on associative arrays indexed by string.

Subscripts of Data Types Other Than VARCHAR2

In the declaration of an associative array indexed by string, the string type must be `VARCHAR2` or one of its subtypes. However, you can populate the associative array with subscripts of any data type that the `TO_CHAR` function can convert to `VARCHAR2`. (For information about `TO_CHAR`, see *Oracle Database SQL Language Reference*.)

If your subscripts have data types other than `VARCHAR2` and its subtypes, ensure that these subscripts remain consistent and unique if the values of initialization parameters change. For example:

- Do not use `TO_CHAR(SYSDATE)` as a subscript.

  If the value of `NLS_DATE_FORMAT` changes, then the value of `(TO_CHAR(SYSDATE))` might also change.

- Do not use different `NVARCHAR2` subscripts that might be converted to the same `VARCHAR2` value.

- Do not use `CHAR` or `VARCHAR2` subscripts that differ only in case, accented characters, or punctuation characters.

  If the value of `NLS_SORT` ends in `_CI` (case-insensitive comparisons) or `_AI` (accent- and case-insensitive comparisons), then subscripts that differ only in case, accented characters, or punctuation characters might be converted to the same value.

Passing Associative Arrays to Remote Databases

If you pass an associative array as a parameter to a remote database, and the local and the remote databases have different `NLS_SORT` or `NLS_COMP` values, then:

- The collection method `FIRST`, `LAST`, `NEXT` or `PRIOR` (described in "Collection Methods" on page 5-21) might return unexpected values or raise exceptions.

- Subscripts that are unique on the local database might not be unique on the remote database, raising the predefined exception `VALUE_ERROR`.

Appropriate Uses for Associative Arrays

An associative array is appropriate for:

- A relatively small lookup table, which can be constructed in memory each time you invoke the subprogram or initialize the package that declares it

- Passing collections to and from the database server

  Declare formal subprogram parameters of associative array types. With Oracle Call Interface (OCI) or an Oracle precompiler, bind the host arrays to the corresponding actual parameters. PL/SQL automatically converts between host arrays and associative arrays indexed by integer.

  Note: You cannot declare an associative array type at schema level. Therefore, to pass an associative array variable as a parameter to a standalone stored subprogram, you must declare the type of that variable in a package specification. Doing so makes the type available to both the invoked subprogram (which declares a formal parameter of that type) and the invoking subprogram or anonymous block (which declares and passes the variable of that type). See Example 10–2.

  Tip: The most efficient way to pass collections to and from the database server is to use associative arrays with the `FORALL` statement or `BULK COLLECT` clause. For details, see "FORALL Statement" on page 12-11 and "BULK COLLECT Clause" on page 12-23.

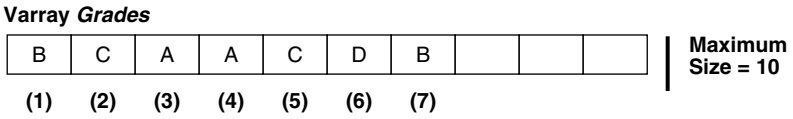
An associative array is intended for temporary data storage. To make an associative array persistent for the life of a database session, declare it in a package specification and populate it in the package body.

Varrays (Variable-Size Arrays)

A **varray (variable-size array)** is an array whose number of elements can vary from zero (empty) to the declared maximum size. To access an element of a varray variable, use the syntax *variable_name*(*subscript*). The lower bound of *subscript* is 1; the upper bound is the current number of elements. The upper bound changes as you add or delete elements, but it cannot exceed the maximum size. When you store and retrieve a varray from the database, its subscripts and element order remain stable.

Figure 5–1shows a varray variable named `Grades`, which has maximum size 10 and contains seven elements. `Grades`(*n*) references the *n*th element of `Grades`. The upper bound of `Grades` is 7, and it cannot exceed 10.

Figure 5–1 Varray of Maximum Size 10 with 7 Elements



The database stores a varray variable as a single object. If a varray variable is less than 4 KB, it resides inside the table of which it is a column; otherwise, it resides outside the table but in the same tablespace.

An uninitialized varray variable is a null collection. You must initialize it, either by making it empty or by assigning a non-`NULL` value to it. For details, see "Collection Constructors" on page 5-13 and "Assigning Values to Collection Variables" on page 5-14.

Example 5–3 defines a local `VARRAY` type, declares a variable of that type (initializing it with a constructor), and defines a procedure that prints the varray. The example invokes the procedure three times: After initializing the variable, after changing the values of two elements individually, and after using a constructor to the change the values of all elements. (For an example of a procedure that prints a varray that might be null or empty, see Example 5–23.)

Example 5–3 Varray (Variable-Size Array)

```
DECLARE
  TYPE Foursome IS VARRAY(4) OF VARCHAR2(15);  -- VARRAY type

  -- varray variable initialized with constructor:

  team Foursome := Foursome('John', 'Mary', 'Alberto', 'Juanita');

  PROCEDURE print_team (heading VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(heading);

    FOR i IN 1..4 LOOP
      DBMS_OUTPUT.PUT_LINE(i || '.' || team(i));
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('---');
  END;

BEGIN
  print_team('2001 Team:');
```

```
  team(3) := 'Pierre';  -- Change values of two elements
  team(4) := 'Yvonne';
  print_team('2005 Team:');

  -- Invoke constructor to assign new values to varray variable:

  team := Foursome('Arun', 'Amitha', 'Allan', 'Mae');
  print_team('2009 Team:');
END;
/
```

Result:

```
2001 Team:
1.John
2.Mary
3.Alberto
4.Juanita
---
2005 Team:
1.John
2.Mary
3.Pierre
4.Yvonne
---
2009 Team:
1.Arun
2.Amitha
3.Allan
4.Mae
---
```

Topics

- Appropriate Uses for Varrays

See Also:

- Table 5–1 for a summary of varray characteristics
- "varray_type_def ::=" on page 13-26 for the syntax of a `VARRAY` type definition
- "CREATE TYPE Statement" on page 14-68 for information about creating standalone stored `VARRAY` types
- *Oracle Database SQL Language Reference* for more information about varrays

Appropriate Uses for Varrays

A varray is appropriate when:

- You know the maximum number of elements.
- You usually access the elements sequentially.

Because you must store or retrieve all elements at the same time, a varray might be impractical for large numbers of elements.

Nested Tables

In the database, a **nested table** is a column type that stores an unbounded set of rows in no particular order. When you retrieve a nested table value from the database into a PL/SQL nested table variable, PL/SQL gives the rows consecutive subscripts, starting at 1. Using these subscripts, you can access the individual rows of the nested table variable. The syntax is *variable_name*(*subscript*). The subscripts and row order of a nested table might not remain stable as you store and retrieve the nested table from the database.

The amount of memory that a nested table variable occupies can increase or decrease dynamically, as you add or delete elements.

An uninitialized nested table variable is a null collection. You must initialize it, either by making it empty or by assigning a non-NULL value to it. For details, see "Collection Constructors" on page 5-13 and "Assigning Values to Collection Variables" on page 5-14.

Example 5–4 defines a local nested table type, declares a variable of that type (initializing it with a constructor), and defines a procedure that prints the nested table. (The procedure uses the collection methods FIRST and LAST, described in "Collection Methods" on page 5-21.) The example invokes the procedure three times: After initializing the variable, after changing the value of one element, and after using a constructor to the change the values of all elements. After the second constructor invocation, the nested table has only two elements. Referencing element 3 would raise error ORA-06533.

Example 5–4 Nested Table of Local Type

```
DECLARE
  TYPE Roster IS TABLE OF VARCHAR2(15);  -- nested table type

  -- nested table variable initialized with constructor:

  names Roster := Roster('D Caruso', 'J Hamil', 'D Piro', 'R Singh');

  PROCEDURE print_names (heading VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(heading);

    FOR i IN names.FIRST .. names.LAST LOOP  -- For first to last element
      DBMS_OUTPUT.PUT_LINE(names(i));
    END LOOP;

    DBMS_OUTPUT.PUT_LINE('---');
  END;

BEGIN
  print_names('Initial Values:');

  names(3) := 'P Perez';  -- Change value of one element
  print_names('Current Values:');

  names := Roster('A Jansen', 'B Gupta');  -- Change entire table
  print_names('Current Values:');
END;
/
```

Result:

```
Initial Values:
D Caruso
J Hamil
D Piro
R Singh
---
Current Values:
D Caruso
J Hamil
P Perez
R Singh
---
Current Values:
A Jansen
B Gupta
```

Example 5–5 defines a standalone stored nested table type, `nt_type`, and a standalone stored procedure to print a variable of that type, `print_nt`. (The procedure uses the collection methods `FIRST` and `LAST`, described in "Collection Methods" on page 5-21.) An anonymous block declares a variable of type `nt_type`, initializing it to empty with a constructor, and invokes `print_nt` twice: After initializing the variable and after using a constructor to the change the values of all elements.

> **Note:** Example 5–16, Example 5–18, and Example 5–19 reuse `nt_type` and `print_nt`.

Example 5–5 Nested Table of Standalone Stored Type

```
CREATE OR REPLACE TYPE nt_type IS TABLE OF NUMBER;
/
CREATE OR REPLACE PROCEDURE print_nt (nt nt_type) IS
  i  NUMBER;
BEGIN
  i := nt.FIRST;

  IF i IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('nt is empty');
  ELSE
    WHILE i IS NOT NULL LOOP
      DBMS_OUTPUT.PUT('nt.(' || i || ') = '); print(nt(i));
      i := nt.NEXT(i);
    END LOOP;
  END IF;

  DBMS_OUTPUT.PUT_LINE('---');
END print_nt;
/
DECLARE
  nt nt_type := nt_type();  -- nested table variable initialized to empty
BEGIN
  print_nt(nt);
  nt := nt_type(90, 9, 29, 58);
  print_nt(nt);
END;
/
```

Result:

```
nt is empty
---
nt.(1) = 90
nt.(2) = 9
nt.(3) = 29
nt.(4) = 58
---
```

Topics

- Important Differences Between Nested Tables and Arrays
- Appropriate Uses for Nested Tables

See Also:

- Table 5–1 for a summary of nested table characteristics
- "nested_table_type_def ::=" on page 13-26 for the syntax of a nested table type definition
- "CREATE TYPE Statement" on page 14-68 for information about creating standalone stored nested table types
- "INSTEAD OF Triggers on Nested Table Columns of Views" on page 9-12 for information about triggers that update nested table columns of views
- *Oracle Database SQL Language Reference* for more information about nested tables

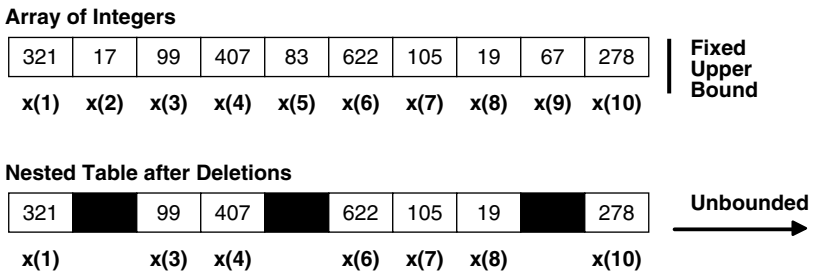
Important Differences Between Nested Tables and Arrays

Conceptually, a nested table is like a one-dimensional array with an arbitrary number of elements. However, 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.
- An array is always dense. A nested array is dense initially, but it can become sparse, because you can delete elements from it.

Figure 5–2 shows the important differences between a nested table and an array.

Figure 5–2 Array and Nested Table



Appropriate Uses for Nested Tables

A nested table is appropriate when:

- The number of elements is not set.

- Subscript values are not consecutive.
- 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.

Collection Constructors

> **Note:** This topic applies only to varrays and nested tables. Associative arrays do not have constructors. In this topic, *collection* means *varray or nested table*.

A **collection constructor (constructor)** is a system-defined function with the same name as a collection type, which returns a collection of that type. The syntax of a constructor invocation is:

```
collection_type ( [ value [, value ]... ] )
```

If the parameter list is empty, the constructor returns an empty collection. Otherwise, the constructor returns a collection that contains the specified values. For semantic details, see "collection_constructor" on page 13-67.

You can assign the returned collection to a collection variable (of the same type) in the variable declaration and in the executable part of a block.

Example 5–6 invokes a constructor twice: to initialize the varray variable `team` to empty in its declaration, and to give it new values in the executable part of the block. The procedure `print_team` shows the initial and final values of `team`. To determine when `team` is empty, `print_team` uses the collection method `COUNT`, described in "Collection Methods" on page 5-21. (For an example of a procedure that prints a varray that might be null, see Example 5–23.)

Example 5–6 Initializing Collection (Varray) Variable to Empty

```
DECLARE
  TYPE Foursome IS VARRAY(4) OF VARCHAR2(15);
  team Foursome := Foursome();  -- initialize to empty

  PROCEDURE print_team (heading VARCHAR2)
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(heading);

    IF team.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE('Empty');
    ELSE
      FOR i IN 1..4 LOOP
        DBMS_OUTPUT.PUT_LINE(i || '.' || team(i));
      END LOOP;
    END IF;

    DBMS_OUTPUT.PUT_LINE('---');
```

```
  END;

BEGIN
  print_team('Team:');
  team := Foursome('John', 'Mary', 'Alberto', 'Juanita');
  print_team('Team:');
END;
/
```

Result:

```
Team:
Empty
---
Team:
1.John
2.Mary
3.Alberto
4.Juanita
---
```

Assigning Values to Collection Variables

You can assign a value to a collection variable in these ways:

- Invoke a constructor to create a collection and assign it to the collection variable, as explained in "Collection Constructors" on page 5-13.
- Use the assignment statement (described in "Assignment Statement" on page 13-3) to assign it the value of another existing collection variable.
- Pass it to a subprogram as an `OUT` or `IN OUT` parameter, and then assign the value inside the subprogram.

To assign a value to a scalar element of a collection variable, reference the element as *`collection_variable_name(subscript)`* and assign it a value as instructed in "Assigning Values to Variables" on page 2-20.

Topics

- Data Type Compatibility
- Assigning Null Values to Varray or Nested Table Variables
- Assigning Set Operation Results to Nested Table Variables

See Also: "BULK COLLECT Clause" on page 12-23

Data Type Compatibility

You can assign a collection to a collection variable only if they have the same data type. Having the same element type is not enough.

In Example 5–7, `VARRAY` types `triplet` and `trio` have the same element type, `VARCHAR(15)`. Collection variables `group1` and `group2` have the same data type, `triplet`, but collection variable `group3` has the data type `trio`. The assignment of `group1` to `group2` succeeds, but the assignment of `group1` to `group3` fails.

Example 5–7 Data Type Compatibility for Collection Assignment

```
DECLARE
  TYPE triplet IS VARRAY(3) OF VARCHAR2(15);
```

```
  TYPE trio    IS VARRAY(3) OF VARCHAR2(15);

  group1 triplet := triplet('Jones', 'Wong', 'Marceau');
  group2 triplet;
  group3 trio;
BEGIN
  group2 := group1;  -- succeeds
  group3 := group1;  -- fails
END;
/
```

Result:

```
ERROR at line 10:
ORA-06550: line 10, column 13:
PLS-00382: expression is of wrong type
ORA-06550: line 10, column 3:
PL/SQL: Statement ignored
```

Assigning Null Values to Varray or Nested Table Variables

To a varray or nested table variable, you can assign the value `NULL` or a null collection of the same data type. Either assignment makes the variable null.

Example 5–6 initializes the nested table variable `dname_tab` to a non-null value; assigns a null collection to it, making it null; and re-initializes it to a different non-null value.

Example 5–8 Assigning Null Value to Nested Table Variable

```
DECLARE
  TYPE dnames_tab IS TABLE OF VARCHAR2(30);

  dept_names dnames_tab := dnames_tab(
    'Shipping','Sales','Finance','Payroll');  -- Initialized to non-null value

  empty_set dnames_tab;  -- Not initialized, therefore null

  PROCEDURE print_dept_names_status IS
  BEGIN
    IF dept_names IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('dept_names is null.');
    ELSE
      DBMS_OUTPUT.PUT_LINE('dept_names is not null.');
    END IF;
  END  print_dept_names_status;

BEGIN
  print_dept_names_status;
  dept_names := empty_set;  -- Assign null collection to dept_names.
  print_dept_names_status;
  dept_names := dnames_tab (
    'Shipping','Sales','Finance','Payroll');  -- Re-initialize dept_names
  print_dept_names_status;
END;
/
```

Result:

```
dept_names is not null.
dept_names is null.
```

```
dept_names is not null.
```

Assigning Set Operation Results to Nested Table Variables

To a nested table variable, you can assign the result of a SQL `MULTISET` operation or SQL `SET` function invocation.

The SQL `MULTISET` operators combine two nested tables into a single nested table. The elements of the two nested tables must have comparable data types. For information about the `MULTISET` operators, see *Oracle Database SQL Language Reference*.

The SQL `SET` function takes a nested table argument and returns a nested table of the same data type whose elements are distinct (the function eliminates duplicate elements). For information about the `SET` function, see *Oracle Database SQL Language Reference*.

Example 5–9 assigns the results of several `MULTISET` operations and one `SET` function invocation of the nested table variable `answer`, using the procedure `print_nested_table` to print `answer` after each assignment. The procedure use the collection methods `FIRST` and `LAST`, described in "Collection Methods" on page 5-21.

Example 5–9 Assigning Set Operation Results to Nested Table Variable

```
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;

  PROCEDURE print_nested_table (nt nested_typ) IS
    output VARCHAR2(128);
  BEGIN
    IF nt IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Result: null set');
    ELSIF nt.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE('Result: empty set');
    ELSE
      FOR i IN nt.FIRST .. nt.LAST LOOP  -- For first to last element
        output := output || nt(i) || ' ';
      END LOOP;
      DBMS_OUTPUT.PUT_LINE('Result: ' || output);
    END IF;
  END print_nested_table;

BEGIN
  answer := nt1 MULTISET UNION nt4;
  print_nested_table(answer);
  answer := nt1 MULTISET UNION nt3;
  print_nested_table(answer);
  answer := nt1 MULTISET UNION DISTINCT nt3;
  print_nested_table(answer);
  answer := nt2 MULTISET INTERSECT nt3;
  print_nested_table(answer);
  answer := nt2 MULTISET INTERSECT DISTINCT nt3;
  print_nested_table(answer);
  answer := SET(nt3);
```

```
  print_nested_table(answer);
  answer := nt3 MULTISET EXCEPT nt2;
  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
```

Multidimensional Collections

Although a collection has only one dimension, you can model a multidimensional collection with a collection whose elements are collections.

In Example 5–10, `nva` is a two-dimensional varray—a varray of varrays of integers.

Example 5–10 Two-Dimensional Varray (Varray of Varrays)

```
DECLARE
  TYPE t1 IS VARRAY(10) OF INTEGER;  -- varray of integer
  va t1 := t1(2,3,5);

  TYPE nt1 IS VARRAY(10) OF t1;      -- varray of varray of integer
  nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);

  i INTEGER;
  va1 t1;
BEGIN
  i := nva(2)(3);
  DBMS_OUTPUT.PUT_LINE('i = ' || i);

  nva.EXTEND;
  nva(5) := t1(56, 32);          -- replace inner varray elements
  nva(4) := t1(45,43,67,43345);  -- replace an inner integer element
  nva(4)(4) := 1;                -- replace 43345 with 1

  nva(4).EXTEND;    -- add element to 4th varray element
  nva(4)(5) := 89;  -- store integer 89 there
END;
/
```

Result:

```
i = 73
```

In Example 5–11, `ntb1` is a nested table of nested tables of strings, and `ntb2` is a nested table of varrays of integers.

Example 5–11 Nested Tables of Nested Tables and Varrays of Integers

```
DECLARE
```

```
  TYPE tb1 IS TABLE OF VARCHAR2(20);  -- nested table of strings
  vtb1 tb1 := tb1('one', 'three');

  TYPE ntb1 IS TABLE OF tb1; -- nested table of nested tables of strings
  vntb1 ntb1 := ntb1(vtb1);

  TYPE tv1 IS VARRAY(10) OF INTEGER;  -- varray of integers
  TYPE ntb2 IS TABLE OF tv1;          -- nested table of varrays of integers
  vntb2 ntb2 := ntb2(tv1(3,5), tv1(5,7,3));

BEGIN
  vntb1.EXTEND;
  vntb1(2) := vntb1(1);
  vntb1.DELETE(1);     -- delete first element of vntb1
  vntb1(2).DELETE(1);  -- delete first string from second table in nested table
END;
/
```

In Example 5–12, `ntb1` is a nested table of associative arrays, and `ntb2` is a nested table of varrays of strings.

Example 5–12 Nested Tables of Associative Arrays and Varrays of Strings

```
DECLARE
  TYPE tb1 IS TABLE OF INTEGER INDEX BY PLS_INTEGER;  -- associative arrays
  v4 tb1;
  v5 tb1;

  TYPE ntb1 IS TABLE OF tb1 INDEX BY PLS_INTEGER;  -- nested table of
  v2 ntb1;                                         --  associative arrays

  TYPE va1 IS VARRAY(10) OF VARCHAR2(20);  -- varray of strings
  v1 va1 := va1('hello', 'world');

  TYPE ntb2 IS TABLE OF va1 INDEX BY PLS_INTEGER;  -- nested table of varrays
  v3 ntb2;

BEGIN
  v4(1)   := 34;     -- populate associative array
  v4(2)   := 46456;
  v4(456) := 343;

  v2(23) := v4;  -- populate nested table of associative arrays

  v3(34) := va1(33, 456, 656, 343);  -- populate nested table of varrays

  v2(35) := v5;      -- assign empty associative array to v2(35)
  v2(35)(2) := 78;
END;
/
```

Collection Comparisons

You cannot compare associative array variables to the value `NULL` or to each other.

Except for Comparing Nested Tables for Equality and Inequality, you cannot natively compare two collection variables with relational operators (listed in Table 2–5). This restriction also applies to implicit comparisons. For example, a collection variable cannot appear in a `DISTINCT`, `GROUP BY`, or `ORDER BY` clause.

To determine if one collection variable is less than another (for example), you must define what less than means in that context and write a function that returns TRUE or FALSE. For information about writing functions, see Chapter 8, "PL/SQL Subprograms."

Topics

- Comparing Varray and Nested Table Variables to NULL
- Comparing Nested Tables for Equality and Inequality
- Comparing Nested Tables with SQL Multiset Conditions

Comparing Varray and Nested Table Variables to NULL

You can compare varray and nested table variables to the value NULL with the "IS [NOT] NULL Operator" on page 2-34, but not with the relational operators equal (=) and not equal (<>, !=, ~=, or ^=).

Example 5–13 compares a varray variable and a nested table variable to NULL correctly.

Example 5–13 Comparing Varray and Nested Table Variables to NULL

```
DECLARE
  TYPE Foursome IS VARRAY(4) OF VARCHAR2(15);  -- VARRAY type
  team Foursome;                               -- varray variable

  TYPE Roster IS TABLE OF VARCHAR2(15);        -- nested table type
  names Roster := Roster('Adams', 'Patel');    -- nested table variable

BEGIN
  IF team IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('team IS NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('team IS NOT NULL');
  END IF;

  IF names IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE('names IS NOT NULL');
  ELSE
    DBMS_OUTPUT.PUT_LINE('names IS NULL');
  END IF;
END;
/
```

Result:

```
team IS NULL
names IS NOT NULL
```

Comparing Nested Tables for Equality and Inequality

If two nested table variables have the same nested table type, and that nested table type does not have elements of a record type, then you can compare the two variables for equality or inequality with the relational operators equal (=) and not equal (<>, !=, ~=, ^=). Two nested table variables are equal if and only if they have the same set of elements (in any order).

See Also: "Record Comparisons" on page 5-49

Example 5–14 compares nested table variables for equality and inequality with relational operators.

Example 5–14 Comparing Nested Tables for Equality and Inequality

```
DECLARE
  TYPE dnames_tab IS TABLE OF VARCHAR2(30); -- element type is not record type

  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
  IF dept_names1 = dept_names2 THEN
    DBMS_OUTPUT.PUT_LINE('dept_names1 = dept_names2');
  END IF;

  IF dept_names2 != dept_names3 THEN
    DBMS_OUTPUT.PUT_LINE('dept_names2 != dept_names3');
  END IF;
END;
/
```

Result:

```
dept_names1 = dept_names2
dept_names2 != dept_names3
```

Comparing Nested Tables with SQL Multiset Conditions

You can compare nested table variables, and test some of their properties, with SQL multiset conditions (described in *Oracle Database SQL Language Reference*).

Example 5–15 uses the SQL multiset conditions and two SQL functions that take nested table variable arguments, `CARDINALITY` (described in *Oracle Database SQL Language Reference*) and `SET` (described in *Oracle Database SQL Language Reference*).

Example 5–15 Comparing Nested Tables with SQL Multiset Conditions

```
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);

  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
  testify(truth => (nt1 IN (nt2,nt3,nt4)));        -- condition
  testify(truth => (nt1 SUBMULTISET OF nt3));      -- condition
  testify(truth => (nt1 NOT SUBMULTISET OF nt4));  -- condition
  testify(truth => (4 MEMBER OF nt1));             -- condition
  testify(truth => (nt3 IS A SET));                -- condition
  testify(truth => (nt3 IS NOT A SET));            -- condition
  testify(truth => (nt1 IS EMPTY));                -- condition
  testify(quantity => (CARDINALITY(nt3)));         -- function
  testify(quantity => (CARDINALITY(SET(nt3))));    -- 2 functions
END;
/
```

Result:

```
True
True
True
False
False
True
False
4
3
```

Collection Methods

A collection method is a 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 your applications easier to maintain. Table 5–2 summarizes the collection methods.

> **Note:** With a null collection, `EXISTS` is the only collection method that does not raise the predefined exception `COLLECTION_IS_NULL`.

Table 5–2 Collection Methods

Method	Type	Description
`DELETE`	Procedure	Deletes elements from collection.
`TRIM`	Procedure	Deletes elements from end of varray or nested table.
`EXTEND`	Procedure	Adds elements to end of varray or nested table.
`EXISTS`	Function	Returns `TRUE` if and only if specified element of varray or nested table exists.
`FIRST`	Function	Returns first subscript in collection.
`LAST`	Function	Returns last subscript in collection.
`COUNT`	Function	Returns number of elements in collection.

Table 5–2 (Cont.) Collection Methods

Method	Type	Description
LIMIT	Function	Returns maximum number of elements that collection can have.
PRIOR	Function	Returns subscript that precedes specified subscript.
NEXT	Function	Returns subscript that succeeds specified subscript.

The basic syntax of a collection method invocation is:

```
collection_name.method
```

For detailed syntax, see "Collection Method Invocation" on page 13-31.

A collection method invocation can appear anywhere that an invocation of a PL/SQL subprogram of its type (function or procedure) can appear, except in a SQL statement. (For general information about PL/SQL subprograms, see Chapter 8, "PL/SQL Subprograms.")

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

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

DELETE Collection Method

DELETE is a procedure that deletes elements from a collection. This method has these forms:

- DELETE deletes all elements from a collection of any type.

  This operation immediately frees the memory allocated to the deleted elements.

- From an associative array or nested table:

  – DELETE(*n*) deletes the element whose subscript is *n*, if that element exists; otherwise, it does nothing.

  – DELETE(*m*,*n*) deletes all elements whose subscripts are in the range *m*..*n*, if both *m* and *n* exist and *m* <= *n*; otherwise, it does nothing.

  For these two forms of DELETE, PL/SQL keeps placeholders for the deleted elements. Therefore, the deleted elements are included in the internal size of the collection, and you can restore a deleted element by assigning a valid value to it.

Example 5–16 declares a nested table variable, initializing it with six elements; deletes and then restores the second element; deletes a range of elements and then restores one of them; and then deletes all elements. The restored elements occupy the same memory as the corresponding deleted elements. The procedure `print_nt` prints the nested table variable after initialization and after each `DELETE` operation. The type `nt_type` and procedure `print_nt` are defined in Example 5–5.

Example 5–16 DELETE Method with Nested Table

```
DECLARE
  nt nt_type := nt_type(11, 22, 33, 44, 55, 66);
BEGIN
  print_nt(nt);

  nt.DELETE(2);     -- Delete second element
  print_nt(nt);

  nt(2) := 2222;    -- Restore second element
  print_nt(nt);

  nt.DELETE(2, 4);  -- Delete range of elements
  print_nt(nt);

  nt(3) := 3333;    -- Restore third element
  print_nt(nt);

  nt.DELETE;        -- Delete all elements
  print_nt(nt);
END;
/
```

Result:

```
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(2) = 2222
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(3) = 3333
nt.(5) = 55
```

```
nt.(6) = 66
---
nt is empty
---
```

Example 5–17 populates an associative array indexed by string and deletes all elements, which frees the memory allocated to them. Next, the example replaces the deleted elements—that is, adds new elements that have the same subscripts as the deleted elements. The new replacement elements do not occupy the same memory as the corresponding deleted elements. Finally, the example deletes one element and then a range of elements. The procedure `print_aa_str` shows the effects of the operations.

Example 5–17 DELETE Method with Associative Array Indexed by String

```
DECLARE
  TYPE aa_type_str IS TABLE OF INTEGER INDEX BY VARCHAR2(10);
  aa_str  aa_type_str;

  PROCEDURE print_aa_str IS
    i  VARCHAR2(10);
  BEGIN
    i := aa_str.FIRST;

    IF i IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('aa_str is empty');
    ELSE
      WHILE i IS NOT NULL LOOP
        DBMS_OUTPUT.PUT('aa_str.(' || i || ') = '); print(aa_str(i));
        i := aa_str.NEXT(i);
      END LOOP;
    END IF;

    DBMS_OUTPUT.PUT_LINE('---');
  END print_aa_str;

BEGIN
  aa_str('M') := 13;
  aa_str('Z') := 26;
  aa_str('C') := 3;
  print_aa_str;

  aa_str.DELETE;  -- Delete all elements
  print_aa_str;

  aa_str('M') := 13;   -- Replace deleted element with same value
  aa_str('Z') := 260;  -- Replace deleted element with new value
  aa_str('C') := 30;   -- Replace deleted element with new value
  aa_str('W') := 23;   -- Add new element
  aa_str('J') := 10;   -- Add new element
  aa_str('N') := 14;   -- Add new element
  aa_str('P') := 16;   -- Add new element
  aa_str('W') := 23;   -- Add new element
  aa_str('J') := 10;   -- Add new element
  print_aa_str;

  aa_str.DELETE('C');      -- Delete one element
  print_aa_str;

  aa_str.DELETE('N','W');  -- Delete range of elements
```

```
  print_aa_str;

  aa_str.DELETE('Z','M');  -- Does nothing
  print_aa_str;
END;
/
```

Result:

```
aa_str.(C) = 3
aa_str.(M) = 13
aa_str.(Z) = 26
---
aa_str is empty
---
aa_str.(C) = 30
aa_str.(J) = 10
aa_str.(M) = 13
aa_str.(N) = 14
aa_str.(P) = 16
aa_str.(W) = 23
aa_str.(Z) = 260
---
aa_str.(J) = 10
aa_str.(M) = 13
aa_str.(N) = 14
aa_str.(P) = 16
aa_str.(W) = 23
aa_str.(Z) = 260
---
aa_str.(J) = 10
aa_str.(M) = 13
aa_str.(Z) = 260
---
aa_str.(J) = 10
aa_str.(M) = 13
aa_str.(Z) = 260
---
```

TRIM Collection Method

`TRIM` is a procedure that deletes elements from the end of a varray or nested table. This method has these forms:

- `TRIM` removes one element from the end of the collection, if the collection has at least one element; otherwise, it raises the predefined exception `SUBSCRIPT_BEYOND_COUNT`.
- `TRIM(`*n*`)` removes *n* elements from the end of the collection, if there are at least *n* elements at the end; otherwise, it raises the predefined exception `SUBSCRIPT_BEYOND_COUNT`.

`TRIM` operates on the internal size of a collection. That is, if `DELETE` deletes an element but keeps a placeholder for it, then `TRIM` considers the element to exist. Therefore, `TRIM` can delete a deleted element.

PL/SQL does not keep placeholders for trimmed elements. Therefore, trimmed elements are not included in the internal size of the collection, and you cannot restore a trimmed element by assigning a valid value to it.

> **Caution:** Do not depend on interaction between TRIM and DELETE. Treat nested tables like either fixed-size arrays (and use only DELETE) or stacks (and use only TRIM and EXTEND).

Example 5–18 declares a nested table variable, initializing it with six elements; trims the last element; deletes the fourth element; and then trims the last two elements—one of which is the deleted fourth element. The procedure print_nt prints the nested table variable after initialization and after the TRIM and DELETE operations. The type nt_type and procedure print_nt are defined in Example 5–5.

Example 5–18 TRIM Method with Nested Table

```
DECLARE
  nt nt_type := nt_type(11, 22, 33, 44, 55, 66);
BEGIN
  print_nt(nt);

  nt.TRIM;       -- Trim last element
  print_nt(nt);

  nt.DELETE(4);  -- Delete fourth element
  print_nt(nt);

  nt.TRIM(2);    -- Trim last two elements
  print_nt(nt);
END;
/
```

Result:

```
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
nt.(6) = 66
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 44
nt.(5) = 55
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(5) = 55
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
---
```

EXTEND Collection Method

EXTEND is a procedure that adds elements to the end of a varray or nested table. The collection can be empty, but not null. (To make a collection empty or add elements to a

null collection, use a constructor. For more information, see "Collection Constructors" on page 5-13.)

The EXTEND method has these forms:

- EXTEND appends one null element to the collection.
- EXTEND(*n*) appends *n* null elements to the collection.
- EXTEND(*n*,*i*) appends *n* copies of the *i*th element to the collection.

> **Note:** EXTEND(*n*,*i*) is the only form that you can use for a collection whose elements have the NOT NULL constraint.

EXTEND operates on the internal size of a collection. That is, if DELETE deletes an element but keeps a placeholder for it, then EXTEND considers the element to exist.

Example 5–19 declares a nested table variable, initializing it with three elements; appends two copies of the first element; deletes the fifth (last) element; and then appends one null element. Because EXTEND considers the deleted fifth element to exist, the appended null element is the sixth element. The procedure print_nt prints the nested table variable after initialization and after the EXTEND and DELETE operations. The type nt_type and procedure print_nt are defined in Example 5–5.

Example 5–19 EXTEND Method with Nested Table

```
DECLARE
  nt nt_type := nt_type(11, 22, 33);
BEGIN
  print_nt(nt);

  nt.EXTEND(2,1);  -- Append two copies of first element
  print_nt(nt);

  nt.DELETE(5);    -- Delete fifth element
  print_nt(nt);

  nt.EXTEND;       -- Append one null element
  print_nt(nt);
END;
/
```

Result:

```
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 11
nt.(5) = 11
---
nt.(1) = 11
nt.(2) = 22
nt.(3) = 33
nt.(4) = 11
---
nt.(1) = 11
```

```
nt.(2) = 22
nt.(3) = 33
nt.(4) = 11
nt.(6) = NULL
---
```

EXISTS Collection Method

EXISTS is a function that tells you whether the specified element of a varray or nested table 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 the predefined exception SUBSCRIPT_OUTSIDE_LIMIT.

For a deleted element, EXISTS(*n*) returns FALSE, even if DELETE kept a placeholder for it.

Example 5–20 initializes a nested table with four elements, deletes the second element, and prints either the value or status of elements 1 through 6.

Example 5–20 EXISTS Method with Nested Table

```
DECLARE
  TYPE NumList IS TABLE OF INTEGER;
  n NumList := NumList(1,3,5,7);
BEGIN
  n.DELETE(2); -- Delete second element

  FOR i IN 1..6 LOOP
    IF n.EXISTS(i) THEN
      DBMS_OUTPUT.PUT_LINE('n(' || i || ') = ' || n(i));
    ELSE
      DBMS_OUTPUT.PUT_LINE('n(' || i || ') does not exist');
    END IF;
  END LOOP;
END;
/
```

Result:

```
n(1) = 1
n(2) does not exist
n(3) = 5
n(4) = 7
n(5) does not exist
n(6) does not exist
```

FIRST and LAST Collection Methods

FIRST and LAST are functions. If the collection has at least one element, FIRST and LAST return the subscripts of the first and last elements, respectively (ignoring deleted elements, even if DELETE kept placeholders for them). If the collection has only one element, FIRST and LAST return the same subscript. If the collection is empty, FIRST and LAST return NULL.

Topics

- FIRST and LAST Methods for Associative Array
- FIRST and LAST Methods for Varray

- FIRST and LAST Methods for Nested Table

FIRST and LAST Methods for Associative Array

For an associative array indexed by integers, the first and last elements are those with the smallest and largest subscripts, respectively.

Example 5–21 shows the values of FIRST and LAST for an associative array indexed by integer, deletes the first and last elements, and shows the values of FIRST and LAST again.

Example 5–21 FIRST and LAST Values for Associative Array Indexed by Integer

```
DECLARE
  TYPE aa_type_int IS TABLE OF INTEGER INDEX BY PLS_INTEGER;
  aa_int  aa_type_int;

  PROCEDURE print_first_and_last IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('FIRST = ' || aa_int.FIRST);
    DBMS_OUTPUT.PUT_LINE('LAST = ' || aa_int.LAST);
  END print_first_and_last;

BEGIN
  aa_int(1) := 3;
  aa_int(2) := 6;
  aa_int(3) := 9;
  aa_int(4) := 12;

  DBMS_OUTPUT.PUT_LINE('Before deletions:');
  print_first_and_last;

  aa_int.DELETE(1);
  aa_int.DELETE(4);

  DBMS_OUTPUT.PUT_LINE('After deletions:');
  print_first_and_last;
END;
/
```

Result:

```
Before deletions:
FIRST = 1
LAST = 4
After deletions:
FIRST = 2
LAST = 3
```

For an associative array indexed by string, the first and last elements are those with the lowest and highest key values, respectively. Key values are in sorted order (for more information, see "NLS Parameter Values Affect Associative Arrays Indexed by String" on page 5-6).

Example 5–22 shows the values of FIRST and LAST for an associative array indexed by string, deletes the first and last elements, and shows the values of FIRST and LAST again.

Example 5–22 FIRST and LAST Values for Associative Array Indexed by String

```
DECLARE
```

```
  TYPE aa_type_str IS TABLE OF INTEGER INDEX BY VARCHAR2(10);
  aa_str  aa_type_str;

  PROCEDURE print_first_and_last IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('FIRST = ' || aa_str.FIRST);
    DBMS_OUTPUT.PUT_LINE('LAST = ' || aa_str.LAST);
  END print_first_and_last;

BEGIN
  aa_str('Z') := 26;
  aa_str('A') := 1;
  aa_str('K') := 11;
  aa_str('R') := 18;

  DBMS_OUTPUT.PUT_LINE('Before deletions:');
  print_first_and_last;

  aa_str.DELETE('A');
  aa_str.DELETE('Z');

  DBMS_OUTPUT.PUT_LINE('After deletions:');
  print_first_and_last;
END;
/
```

Result:

```
Before deletions:
FIRST = A
LAST = Z
After deletions:
FIRST = K
LAST = R
```

FIRST and LAST Methods for Varray

For a varray that is not empty, `FIRST` always returns 1. For every varray, `LAST` always equals `COUNT` (see Example 5–25).

Example 5–23 prints the varray `team` using a `FOR LOOP` statement with the bounds `team.FIRST` and `team.LAST`. Because a varray is always dense, `team(i)` inside the loop always exists.

Example 5–23 Printing Varray with FIRST and LAST in FOR LOOP

```
DECLARE
  TYPE team_type IS VARRAY(4) OF VARCHAR2(15);
  team team_type;

  PROCEDURE print_team (heading VARCHAR2)
  IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(heading);

    IF team IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Does not exist');
    ELSIF team.FIRST IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Has no members');
    ELSE
      FOR i IN team.FIRST..team.LAST LOOP
```

```
        DBMS_OUTPUT.PUT_LINE(i || '. ' || team(i));
      END LOOP;
    END IF;

    DBMS_OUTPUT.PUT_LINE('---');
  END;

BEGIN
  print_team('Team Status:');

  team := team_type();  -- Team is funded, but nobody is on it.
  print_team('Team Status:');

  team := team_type('John', 'Mary');  -- Put 2 members on team.
  print_team('Initial Team:');

  team := team_type('Arun', 'Amitha', 'Allan', 'Mae');  -- Change team.
  print_team('New Team:');
END;
/
```

Result:

```
Team Status:
Does not exist
---
Team Status:
Has no members
---
Initial Team:
1. John
2. Mary
---
New Team:
1. Arun
2. Amitha
3. Allan
4. Mae
---
```

FIRST and LAST Methods for Nested Table

For a nested table, `LAST` equals `COUNT` unless you delete elements from its middle, in which case `LAST` is larger than `COUNT` (see Example 5–26).

Example 5–24 prints the nested table `team` using a `FOR LOOP` statement with the bounds `team.FIRST` and `team.LAST`. Because a nested table can be sparse, the `FOR LOOP` statement prints `team(i)` only if `team.EXISTS(i)` is `TRUE`.

Example 5–24 Printing Nested Table with FIRST and LAST in FOR LOOP

```
DECLARE
  TYPE team_type IS TABLE OF VARCHAR2(15);
  team team_type;

  PROCEDURE print_team (heading VARCHAR2) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(heading);

    IF team IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Does not exist');
```

```
    ELSIF team.FIRST IS NULL THEN
      DBMS_OUTPUT.PUT_LINE('Has no members');
    ELSE
      FOR i IN team.FIRST..team.LAST LOOP
        DBMS_OUTPUT.PUT(i || '. ');
        IF team.EXISTS(i) THEN
          DBMS_OUTPUT.PUT_LINE(team(i));
        ELSE
          DBMS_OUTPUT.PUT_LINE('(to be hired)');
        END IF;
      END LOOP;
    END IF;

    DBMS_OUTPUT.PUT_LINE('---');
  END;

BEGIN
  print_team('Team Status:');

  team := team_type();  -- Team is funded, but nobody is on it.
  print_team('Team Status:');

  team := team_type('Arun', 'Amitha', 'Allan', 'Mae');  -- Add members.
  print_team('Initial Team:');

  team.DELETE(2,3);  -- Remove 2nd and 3rd members.
  print_team('Current Team:');
END;
/
```

Result:

```
Team Status:
Does not exist
---
Team Status:
Has no members
---
Initial Team:
1. Arun
2. Amitha
3. Allan
4. Mae
---
Current Team:
1. Arun
2. (to be hired)
3. (to be hired)
4. Mae
---
```

COUNT Collection Method

COUNT is a function that returns the number of elements in the collection (ignoring deleted elements, even if DELETE kept placeholders for them).

Topics

- COUNT Method for Varray
- COUNT Method for Nested Table

COUNT Method for Varray

For a varray, COUNT always equals LAST. If you increase or decrease the size of a varray (with the EXTEND or TRIM method), the value of COUNT changes.

Example 5–25 shows the values of COUNT and LAST for a varray after initialization with four elements, after EXTEND(3), and after TRIM(5).

Example 5–25 COUNT and LAST Values for Varray

```
DECLARE
  TYPE NumList IS VARRAY(10) OF INTEGER;
  n NumList := NumList(1,3,5,7);

  PROCEDURE print_count_and_last IS
  BEGIN
    DBMS_OUTPUT.PUT('n.COUNT = ' || n.COUNT || ', ');
    DBMS_OUTPUT.PUT_LINE('n.LAST = ' || n.LAST);
  END  print_count_and_last;

BEGIN
  print_count_and_last;

  n.EXTEND(3);
  print_count_and_last;

  n.TRIM(5);
  print_count_and_last;
END;
/
```

Result:

```
n.COUNT = 4, n.LAST = 4
n.COUNT = 7, n.LAST = 7
n.COUNT = 2, n.LAST = 2
```

COUNT Method for Nested Table

For a nested table, COUNT equals LAST unless you delete elements from the middle of the nested table, in which case COUNT is smaller than LAST.

Example 5–26 shows the values of COUNT and LAST for a nested table after initialization with four elements, after deleting the third element, and after adding two null elements to the end. Finally, the example prints the status of elements 1 through 8.

Example 5–26 COUNT and LAST Values for Nested Table

```
DECLARE
  TYPE NumList IS TABLE OF INTEGER;
  n NumList := NumList(1,3,5,7);

  PROCEDURE print_count_and_last IS
  BEGIN
    DBMS_OUTPUT.PUT('n.COUNT = ' || n.COUNT || ', ');
    DBMS_OUTPUT.PUT_LINE('n.LAST = ' || n.LAST);
  END  print_count_and_last;

BEGIN
  print_count_and_last;

  n.DELETE(3);  -- Delete third element
```

```
  print_count_and_last;

  n.EXTEND(2);  -- Add two null elements to end
  print_count_and_last;

  FOR i IN 1..8 LOOP
    IF n.EXISTS(i) THEN
      IF n(i) IS NOT NULL THEN
        DBMS_OUTPUT.PUT_LINE('n(' || i || ') = ' || n(i));
      ELSE
        DBMS_OUTPUT.PUT_LINE('n(' || i || ') = NULL');
      END IF;
    ELSE
      DBMS_OUTPUT.PUT_LINE('n(' || i || ') does not exist');
    END IF;
  END LOOP;
END;
/
```

Result:

```
n.COUNT = 4, n.LAST = 4
n.COUNT = 3, n.LAST = 4
n.COUNT = 5, n.LAST = 6
n(1) = 1
n(2) = 3
n(3) does not exist
n(4) = 7
n(5) = NULL
n(6) = NULL
n(7) does not exist
n(8) does not exist
```

LIMIT Collection Method

`LIMIT` is a function that returns the maximum number of elements that the collection can have. If the collection has no maximum number of elements, `LIMIT` returns `NULL`. Only a varray has a maximum size.

Example 5–27 and prints the values of `LIMIT` and `COUNT` for an associative array with four elements, a varray with two elements, and a nested table with three elements.

Example 5–27 LIMIT and COUNT Values for Different Collection Types

```
DECLARE
  TYPE aa_type IS TABLE OF INTEGER INDEX BY PLS_INTEGER;
  aa aa_type;                          -- associative array

  TYPE va_type IS VARRAY(4) OF INTEGER;
  va  va_type := va_type(2,4);   -- varray

  TYPE nt_type IS TABLE OF INTEGER;
  nt  nt_type := nt_type(1,3,5);  -- nested table

BEGIN
  aa(1):=3; aa(2):=6; aa(3):=9; aa(4):= 12;
  DBMS_OUTPUT.PUT('aa.COUNT = '); print(aa.COUNT);
  DBMS_OUTPUT.PUT('aa.LIMIT = '); print(aa.LIMIT);

  DBMS_OUTPUT.PUT('va.COUNT = '); print(va.COUNT);
  DBMS_OUTPUT.PUT('va.LIMIT = '); print(va.LIMIT);
```

```
  DBMS_OUTPUT.PUT('nt.COUNT = '); print(nt.COUNT);
  DBMS_OUTPUT.PUT('nt.LIMIT = '); print(nt.LIMIT);
END;
/
```

Result:

```
aa.COUNT = 4
aa.LIMIT = NULL
va.COUNT = 2
va.LIMIT = 4
nt.COUNT = 3
nt.LIMIT = NULL
```

PRIOR and NEXT Collection Methods

`PRIOR` and `NEXT` are functions that let you move backward and forward in the collection (ignoring deleted elements, even if `DELETE` kept placeholders for them). These methods are useful for traversing sparse collections.

Given a subscript:

- `PRIOR` returns the subscript of the preceding existing element of the collection, if one exists. Otherwise, `PRIOR` returns `NULL`.

  For any collection `c`, `c.PRIOR(c.FIRST)` returns `NULL`.

- `NEXT` returns the subscript of the succeeding existing element of the collection, if one exists. Otherwise, `NEXT` returns `NULL`.

  For any collection `c`, `c.NEXT(c.LAST)` returns `NULL`.

The given subscript need not exist. However, if the collection is a varray, then the subscript cannot exceed `LIMIT`.

Example 5–28 initializes a nested table with six elements, deletes the fourth element, and then shows the values of `PRIOR` and `NEXT` for elements 1 through 7. Elements 4 and 7 do not exist. Element 2 exists, despite its null value.

Example 5–28 PRIOR and NEXT Methods

```
DECLARE
  TYPE nt_type IS TABLE OF NUMBER;
  nt nt_type := nt_type(18, NULL, 36, 45, 54, 63);

BEGIN
  nt.DELETE(4);
  DBMS_OUTPUT.PUT_LINE('nt(4) was deleted.');

  FOR i IN 1..7 LOOP
    DBMS_OUTPUT.PUT('nt.PRIOR(' || i || ') = '); print(nt.PRIOR(i));
    DBMS_OUTPUT.PUT('nt.NEXT(' || i || ')  = '); print(nt.NEXT(i));
  END LOOP;
END;
/
```

Result:

```
nt(4) was deleted.
nt.PRIOR(1) = NULL
nt.NEXT(1)  = 2
nt.PRIOR(2) = 1
```

```
nt.NEXT(2)  = 3
nt.PRIOR(3) = 2
nt.NEXT(3)  = 5
nt.PRIOR(4) = 3
nt.NEXT(4)  = 5
nt.PRIOR(5) = 3
nt.NEXT(5)  = 6
nt.PRIOR(6) = 5
nt.NEXT(6)  = NULL
nt.PRIOR(7) = 6
nt.NEXT(7)  = NULL
```

For an associative array indexed by string, the prior and next subscripts are determined by key values, which are in sorted order (for more information, see "NLS Parameter Values Affect Associative Arrays Indexed by String" on page 5-6). Example 5–1 uses `FIRST`, `NEXT`, and a `WHILE LOOP` statement to print the elements of an associative array.

Example 5–29 prints the elements of a sparse nested table from first to last, using `FIRST` and `NEXT`, and from last to first, using `LAST` and `PRIOR`.

Example 5–29 Printing Elements of Sparse Nested Table

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  n NumList := NumList(1, 2, NULL, NULL, 5, NULL, 7, 8, 9, NULL);
  subscript INTEGER;

BEGIN
  DBMS_OUTPUT.PUT_LINE('First to last:');
  subscript := n.FIRST;
  WHILE subscript IS NOT NULL LOOP
    DBMS_OUTPUT.PUT('n(' || subscript || ') = ');
    print(n(subscript));
    subscript := n.NEXT(subscript);
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('--------------');

  DBMS_OUTPUT.PUT_LINE('Last to first:');
  subscript := n.LAST;
  WHILE subscript IS NOT NULL LOOP
    DBMS_OUTPUT.PUT('n(' || subscript || ') = ');
    print(n(subscript));
    subscript := n.PRIOR(subscript);
  END LOOP;
END;
/
```

Result:

```
First to last:
n(1) = 1
n(2) = 2
n(3) = NULL
n(4) = NULL
n(5) = 5
n(6) = NULL
n(7) = 7
n(8) = 8
n(9) = 9
```

```
n(10) = NULL
--------------
Last to first:
n(10) = NULL
n(9) = 9
n(8) = 8
n(7) = 7
n(6) = NULL
n(5) = 5
n(4) = NULL
n(3) = NULL
n(2) = 2
n(1) = 1
```

Collection Types Defined in Package Specifications

A collection type defined in a package specification is incompatible with an identically defined local or standalone stored collection type.

> **Note:** The examples in this topic define packages and procedures, which are explained in Chapter 10, "PL/SQL Packages" and Chapter 8, "PL/SQL Subprograms," respectively.

In Example 5–30, the package specification and the anonymous block define the collection type `NumList` identically. The package defines a procedure, `print_numlist`, which has a `NumList` parameter. The anonymous block declares the variable `n1` of the type `pkg.NumList` (defined in the package) and the variable `n2` of the type `NumList` (defined in the block). The anonymous block can pass `n1` to `print_numlist`, but it cannot pass `n2` to `print_numlist`.

Example 5–30 Identically Defined Package and Local Collection Types

```
CREATE OR REPLACE PACKAGE pkg AS
  TYPE NumList IS TABLE OF NUMBER;
  PROCEDURE print_numlist (nums NumList);
END pkg;
/
CREATE OR REPLACE 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;  -- local type identical to package type
  n1 pkg.NumList := pkg.NumList(2,4);  -- package type
  n2     NumList :=     NumList(6,8);  -- local type
BEGIN
  pkg.print_numlist(n1);  -- succeeds
  pkg.print_numlist(n2);  -- fails
END;
/
```

Result:

```
  pkg.print_numlist(n2);  -- fails
  *
ERROR at line 7:
ORA-06550: line 7, column 3:
PLS-00306: wrong number or types of arguments in call to 'PRINT_NUMLIST'
ORA-06550: line 7, column 3:
PL/SQL: Statement ignored
```

Example 5–31 defines a standalone stored collection type `NumList` that is identical to the collection type `NumList` defined in the package specification in Example 5–30. The anonymous block declares the variable `n1` of the type `pkg.NumList` (defined in the package) and the variable `n2` of the standalone stored type `NumList`. The anonymous block can pass `n1` to `print_numlist`, but it cannot pass `n2` to `print_numlist`.

Example 5–31 Identically Defined Package and Standalone Stored Collection Types

```
CREATE OR REPLACE TYPE NumList IS TABLE OF NUMBER;
  -- standalone stored collection type identical to package type
/
DECLARE
  n1 pkg.NumList := pkg.NumList(2,4);  -- package type
  n2     NumList :=     NumList(6,8);  -- standalone stored type

BEGIN
  pkg.print_numlist(n1);  -- succeeds
  pkg.print_numlist(n2);  -- fails
END;
/
```

Result:

```
  pkg.print_numlist(n2);  -- fails
  *
ERROR at line 7:
ORA-06550: line 7, column 3:
PLS-00306: wrong number or types of arguments in call to 'PRINT_NUMLIST'
ORA-06550: line 7, column 3:
PL/SQL: Statement ignored
```

Record Variables

You can create a record variable in any of these ways:

- Define a `RECORD` type and then declare a variable of that type.
- Use `%ROWTYPE` to declare a record variable that represents either a full or partial row of a database table or view.
- Use `%TYPE` to declare a record variable of the same type as a previously declared record variable.

For syntax and semantics, see "Record Variable Declaration" on page 13-111.

Topics

- Initial Values of Record Variables
- RECORD Types
- %ROWTYPE Attribute

Initial Values of Record Variables

You cannot specify an initial value when you declare a record variable (therefore, you cannot declare a record constant).

For a record variable of a RECORD type, the initial value of each field is NULL unless you specify a different initial value for it when you define the type (as in Example 5–32).

For a record variable declared with %ROWTYPE or %TYPE, the initial value of each field is NULL. The variable does not inherit the initial value of the referenced item.

RECORD Types

A RECORD type defined in a PL/SQL block is a **local type**. It is available only in the block, and is stored in the database only if the block is in a standalone stored or package subprogram. (Standalone stored and package subprograms are explained in "Nested, Package, and Standalone Stored Subprograms" on page 8-2).

A RECORD type defined in a package specification is a **public item**. You can reference it from outside the package by qualifying it with the package name (*package_name.type_name*). It is stored in the database until you drop the package with the DROP PACKAGE statement. (Packages are explained in Chapter 10, "PL/SQL Packages.")

You cannot create a RECORD type at schema level. Therefore, a RECORD type cannot be an ADT attribute data type.

> **Note:** A RECORD type defined in a package specification is incompatible with an identically defined local RECORD type (see Example 5–35).

To define a RECORD type, specify its name and define its fields. To define a field, specify its name and data type. By default, the initial value of a field is NULL. You can specify the NOT NULL constraint for a field, in which case you must also specify a non-NULL initial value. Without the NOT NULL constraint, a non-NULL initial value is optional.

Example 5–32 defines a RECORD type named DeptRecTyp, specifying an initial value for each field. Then it declares a variable of that type named dept_rec and prints its fields.

Example 5–32 RECORD Type Definition and Variable Declaration

```
DECLARE
  TYPE DeptRecTyp IS RECORD (
    dept_id    NUMBER(4) NOT NULL := 10,
    dept_name  VARCHAR2(30) NOT NULL := 'Administration',
    mgr_id     NUMBER(6) := 200,
    loc_id     NUMBER(4) := 1700
  );

  dept_rec DeptRecTyp;
BEGIN
  DBMS_OUTPUT.PUT_LINE('dept_id:   ' || dept_rec.dept_id);
  DBMS_OUTPUT.PUT_LINE('dept_name: ' || dept_rec.dept_name);
  DBMS_OUTPUT.PUT_LINE('mgr_id:    ' || dept_rec.mgr_id);
  DBMS_OUTPUT.PUT_LINE('loc_id:    ' || dept_rec.loc_id);
END;
```

```
/
```

Result:

```
dept_id:   10
dept_name: Administration
mgr_id:    200
loc_id:    1700
```

Example 5–33 defines two RECORD types, name_rec and contact. The type contact has a field of type name_rec.

Example 5–33 RECORD Type with RECORD Field (Nested Record)

```
DECLARE
  TYPE name_rec IS RECORD (
    first  employees.first_name%TYPE,
    last   employees.last_name%TYPE
  );

  TYPE contact IS RECORD (
    name  name_rec,                    -- nested record
    phone employees.phone_number%TYPE
  );

  friend contact;
BEGIN
  friend.name.first := 'John';
  friend.name.last := 'Smith';
  friend.phone := '1-650-555-1234';

  DBMS_OUTPUT.PUT_LINE (
    friend.name.first  || ' ' ||
    friend.name.last   || ', ' ||
    friend.phone
  );
END;
/
```

Result:

```
John Smith, 1-650-555-1234
```

Example 5–34 defines a VARRAY type, full_name, and a RECORD type, contact. The type contact has a field of type full_name.

Example 5–34 RECORD Type with Varray Field

```
DECLARE
  TYPE full_name IS VARRAY(2) OF VARCHAR2(20);

  TYPE contact IS RECORD (
    name  full_name := full_name('John', 'Smith'),  -- varray field
    phone employees.phone_number%TYPE
  );

  friend contact;
BEGIN
  friend.phone := '1-650-555-1234';

  DBMS_OUTPUT.PUT_LINE (
```

```
    friend.name(1) || ' ' ||
    friend.name(2) || ', ' ||
    friend.phone
  );
END;
/
```

Result:

```
John Smith, 1-650-555-1234
```

A RECORD type defined in a package specification is incompatible with an identically defined local RECORD type.

> **Note:** The example in this topic defines a package and a procedure, which are explained in Chapter 10, "PL/SQL Packages" and Chapter 8, "PL/SQL Subprograms," respectively.

In Example 5–35, the package pkg and the anonymous block define the RECORD type rec_type identically. The package defines a procedure, print_rec_type, which has a rec_type parameter. The anonymous block declares the variable r1 of the package type (pkg.rec_type) and the variable r2 of the local type (rec_type). The anonymous block can pass r1 to print_rec_type, but it cannot pass r2 to print_rec_type.

Example 5–35 Identically Defined Package and Local RECORD Types

```
CREATE OR REPLACE PACKAGE pkg AS
  TYPE rec_type IS RECORD (       -- package RECORD type
    f1 INTEGER,
    f2 VARCHAR2(4)
  );
  PROCEDURE print_rec_type (rec rec_type);
END pkg;
/
CREATE OR REPLACE PACKAGE BODY pkg AS
  PROCEDURE print_rec_type (rec rec_type) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(rec.f1);
    DBMS_OUTPUT.PUT_LINE(rec.f2);
  END;
END pkg;
/
DECLARE
  TYPE rec_type IS RECORD (       -- local RECORD type
    f1 INTEGER,
    f2 VARCHAR2(4)
  );
  r1 pkg.rec_type;                -- package type
  r2     rec_type;                -- local type

BEGIN
  r1.f1 := 10; r1.f2 := 'abcd';
  r2.f1 := 25; r2.f2 := 'wxyz';

  pkg.print_rec_type(r1);  -- succeeds
  pkg.print_rec_type(r2);  -- fails
END;
/
```

Result:

```
  pkg.print_rec_type(r2);  -- fails
  *
ERROR at line 14:
ORA-06550: line 14, column 3:
PLS-00306: wrong number or types of arguments in call to 'PRINT_REC_TYPE'
ORA-06550: line 14, column 3:
PL/SQL: Statement ignored
```

%ROWTYPE Attribute

The `%ROWTYPE` attribute lets you declare a record variable that represents either a full or partial row of a database table or view. For every column of the full or partial row, the record has a field with the same name and data type. If the structure of the row changes, then the structure of the record changes accordingly.

The record fields do not inherit the constraints or initial values of the corresponding columns (see Example 5–37).

Topics

- Record Variable that Always Represents Full Row
- Record Variable that Can Represent Partial Row

Record Variable that Always Represents Full Row

To declare a record variable that always represents a full row of a database table or view, use this syntax:

```
variable_name table_or_view_name%ROWTYPE;
```

For every column of the table or view, the record has a field with the same name and data type.

> **See Also:** "%ROWTYPE Attribute" on page 13-121 for more information about `%ROWTYPE`

Example 5–36 declares a record variable that represents a row of the table `departments`, assigns values to its fields, and prints them. Compare this example to Example 5–32.

Example 5–36 %ROWTYPE Variable Represents Full Database Table Row

```
DECLARE
  dept_rec departments%ROWTYPE;
BEGIN
  -- Assign values to fields:

  dept_rec.department_id   := 10;
  dept_rec.department_name := 'Administration';
  dept_rec.manager_id      := 200;
  dept_rec.location_id     := 1700;

  -- Print fields:

  DBMS_OUTPUT.PUT_LINE('dept_id:   ' || dept_rec.department_id);
  DBMS_OUTPUT.PUT_LINE('dept_name: ' || dept_rec.department_name);
  DBMS_OUTPUT.PUT_LINE('mgr_id:    ' || dept_rec.manager_id);
```

```
  DBMS_OUTPUT.PUT_LINE('loc_id:    ' || dept_rec.location_id);
END;
/
```

Result:

```
dept_id:   10
dept_name: Administration
mgr_id:    200
loc_id:    1700
```

Example 5–37 creates a table with two columns, each with an initial value and a NOT NULL constraint. Then it declares a record variable that represents a row of the table and prints its fields, showing that they did not inherit the initial values or NOT NULL constraints.

Example 5–37 %ROWTYPE Variable Does Not Inherit Initial Values or Constraints

```
DROP TABLE t1;
CREATE TABLE t1 (
  c1 INTEGER DEFAULT 0 NOT NULL,
  c2 INTEGER DEFAULT 1 NOT NULL
);

DECLARE
  t1_row t1%ROWTYPE;
BEGIN
  DBMS_OUTPUT.PUT('t1.c1 = '); print(t1_row.c1);
  DBMS_OUTPUT.PUT('t1.c2 = '); print(t1_row.c2);
END;
/
```

Result:

```
t1.c1 = NULL
t1.c2 = NULL
```

Record Variable that Can Represent Partial Row

To declare a record variable that can represent a partial row of a database table or view, use this syntax:

```
variable_name cursor%ROWTYPE;
```

A cursor is associated with a query. For every column that the query selects, the record variable must have a corresponding, type-compatible field. If the query selects every column of the table or view, then the variable represents a full row; otherwise, the variable represents a partial row. The cursor must be either an explicit cursor or a strong cursor variable.

See Also:

- "FETCH Statement" on page 13-71 for complete syntax
- "Cursors" on page 6-5 for information about cursors
- "Explicit Cursors" on page 6-8 for information about explicit cursors
- "Cursor Variables" on page 6-28 for information about cursor variables

Example 5–38 defines an explicit cursor whose query selects only the columns `first_name`, `last_name`, and `phone_number` from the `employees` table in the sample schema `HR`. Then the example declares a record variable that has a field for each column that the cursor selects. The variable represents a partial row of `employees`. Compare this example to Example 5–33.

Example 5–38 %ROWTYPE Variable Represents Partial Database Table Row

```
DECLARE
  CURSOR c IS
    SELECT first_name, last_name, phone_number
    FROM employees;

  friend c%ROWTYPE;
BEGIN
  friend.first_name   := 'John';
  friend.last_name    := 'Smith';
  friend.phone_number := '1-650-555-1234';

  DBMS_OUTPUT.PUT_LINE (
    friend.first_name  || ' ' ||
    friend.last_name   || ', ' ||
    friend.phone_number
  );
END;
/
```

Result:

```
John Smith, 1-650-555-1234
```

Example 5–38 defines an explicit cursor whose query is a join and then declares a record variable that has a field for each column that the cursor selects. (For information about joins, see *Oracle Database SQL Language Reference*.)

Example 5–39 %ROWTYPE Variable Represents Join Row

```
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;
/
```

Assigning Values to Record Variables

Note: In this topic, *record variable* means either a record variable or a record component of a composite variable (for example, `friend.name` in Example 5–33).

To any record variable, you can assign a value to each field individually.

In some cases, you can assign the value of one record variable to another record variable.

If a record variable represents a full or partial row of a database table or view, you can assign the represented row to the record variable.

Topics

- Assigning One Record Variable to Another
- Assigning Full or Partial Rows to Record Variables

Assigning One Record Variable to Another

You can assign the value of one record variable to another record variable only in these cases:

- The two variables have the same `RECORD` type (as in Example 5–40).
- The target variable is declared with a `RECORD` type, the source variable is declared with `%ROWTYPE`, their fields match in number and order, and corresponding fields have the same data type (as in Example 5–41).

For record components of composite variables, the types of the composite variables need not match (see Example 5–42).

Example 5–40 Assigning Record to Another Record of Same RECORD Type

```
DECLARE
  TYPE name_rec IS RECORD (
    first  employees.first_name%TYPE DEFAULT 'John',
    last   employees.last_name%TYPE DEFAULT 'Doe'
  );

  name1 name_rec;
  name2 name_rec;

BEGIN
  name1.first := 'Jane'; name1.last := 'Smith';
  DBMS_OUTPUT.PUT_LINE('name1: ' || name1.first || ' ' || name1.last);
  name2 := name1;
  DBMS_OUTPUT.PUT_LINE('name2: ' || name2.first || ' ' || name2.last);
END;
/
```

Result:

```
name1: Jane Smith
name2: Jane Smith
```

Example 5–41 Assigning %ROWTYPE Record to RECORD Type Record

```
DECLARE
  TYPE name_rec IS RECORD (
    first  employees.first_name%TYPE DEFAULT 'John',
    last   employees.last_name%TYPE DEFAULT 'Doe'
  );

  CURSOR c IS
    SELECT first_name, last_name
    FROM employees;
```

```
  target name_rec;
  source c%ROWTYPE;

BEGIN
  source.first_name := 'Jane'; source.last_name := 'Smith';

  DBMS_OUTPUT.PUT_LINE (
    'source: ' || source.first_name || ' ' || source.last_name
  );

 target := source;

 DBMS_OUTPUT.PUT_LINE (
   'target: ' || target.first || ' ' || target.last
 );
END;
/
```

Result:

```
source: Jane Smith
target: Jane Smith
```

Example 5–42 assigns the value of one nested record to another nested record. The nested records have the same `RECORD` type, but the records in which they are nested do not.

Example 5–42 Assigning Nested Record to Another Record of Same RECORD Type

```
DECLARE
  TYPE name_rec IS RECORD (
    first  employees.first_name%TYPE,
    last   employees.last_name%TYPE
  );

  TYPE phone_rec IS RECORD (
    name  name_rec,                    -- nested record
    phone employees.phone_number%TYPE
  );

  TYPE email_rec IS RECORD (
    name  name_rec,                    -- nested record
    email employees.email%TYPE
  );

  phone_contact phone_rec;
  email_contact email_rec;

BEGIN
  phone_contact.name.first := 'John';
  phone_contact.name.last := 'Smith';
  phone_contact.phone := '1-650-555-1234';

  email_contact.name := phone_contact.name;
  email_contact.email := (
    email_contact.name.first || '.' ||
    email_contact.name.last  || '@' ||
    'example.com'
  );

  DBMS_OUTPUT.PUT_LINE (email_contact.email);
```

```
END;
/
```

Result:

```
John.Smith@example.com
```

Assigning Full or Partial Rows to Record Variables

If a record variable represents a full or partial row of a database table or view, you can assign the represented row to the record variable.

Topics

- SELECT INTO Statement for Assigning Row to Record Variable
- FETCH Statement for Assigning Row to Record Variable
- SQL Statements that Return Rows in PL/SQL Record Variables

SELECT INTO Statement for Assigning Row to Record Variable

The syntax of a simple `SELECT INTO` statement is:

```
SELECT select_list INTO record_variable_name FROM table_or_view_name;
```

For each column in *`select_list`*, the record variable must have a corresponding, type-compatible field. The columns in *`select_list`* must appear in the same order as the record fields.

> **See Also:** "SELECT INTO Statement" on page 13-125 for complete syntax

In Example 5–43, the record variable `rec1` represents a partial row of the `employees` table—the columns `last_name` and `employee_id`. The `SELECT INTO` statement selects from `employees` the row for which `job_id` is `'AD_PRES'` and assigns the values of the columns `last_name` and `employee_id` in that row to the corresponding fields of `rec1`.

Example 5–43 SELECT INTO Assigns Values to Record Variable

```
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 job_id = 'AD_PRES';

  DBMS_OUTPUT.PUT_LINE ('Employee #' || rec1.id || ' = ' || rec1.last);
END;
/
```

Result:

```
Employee #100 = King
```

FETCH Statement for Assigning Row to Record Variable

The syntax of a simple FETCH statement is:

```
FETCH cursor INTO record_variable_name;
```

A cursor is associated with a query. For every column that the query selects, the record variable must have a corresponding, type-compatible field. The cursor must be either an explicit cursor or a strong cursor variable.

See Also:

- "FETCH Statement" on page 13-71 for complete syntax
- "Cursors" on page 6-5 for information about all cursors
- "Explicit Cursors" on page 6-8 for information about explicit cursors
- "Cursor Variables" on page 6-28 for information about cursor variables

In Example 5–44, each variable of RECORD type EmpRecTyp represents a partial row of the employees table—the columns employee_id and salary. Both the cursor and the function return a value of type EmpRecTyp. In the function, a FETCH statement assigns the values of the columns employee_id and salary to the corresponding fields of a local variable of type EmpRecTyp.

Example 5–44 FETCH Assigns Values to Record that Function Returns

```
DECLARE
  TYPE EmpRecTyp IS RECORD (
    emp_id  employees.employee_id%TYPE,
    salary  employees.salary_id%TYPE
  );

  CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT employee_id, salary
    FROM employees
    ORDER BY salary DESC;

  highest_paid_emp       EmpRecTyp;
  next_highest_paid_emp  EmpRecTyp;

  FUNCTION nth_highest_salary (n INTEGER) 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
  highest_paid_emp := nth_highest_salary(1);
  next_highest_paid_emp := nth_highest_salary(2);

  DBMS_OUTPUT.PUT_LINE(
    'Highest Paid: #' ||
    highest_paid_emp.emp_id || ', $' ||
```

```
    highest_paid_emp.salary
  );
  DBMS_OUTPUT.PUT_LINE(
    'Next Highest Paid: #' ||
    next_highest_paid_emp.emp_id || ', $' ||
    next_highest_paid_emp.salary
  );
END;
/
```

Result:

```
Highest Paid: #100, $26460
Next Highest Paid: #101, $18742.5
```

SQL Statements that Return Rows in PL/SQL Record Variables

The SQL statements `INSERT`, `UPDATE`, and `DELETE` have an optional `RETURNING INTO` clause that can return the affected row in a PL/SQL record variable. For information about this clause, see "RETURNING INTO Clause" on page 13-118.

In Example 5–45, the `UPDATE` statement updates the salary of an employee and returns the name and new salary of the employee in a record variable.

Example 5–45 UPDATE Statement Assigns Values to Record Variable

```
DECLARE
  TYPE EmpRec IS RECORD (
    last_name  employees.last_name%TYPE,
    salary     employees.salary%TYPE
  );
  emp_info    EmpRec;
  old_salary  employees.salary%TYPE;
BEGIN
  SELECT salary INTO old_salary
   FROM employees
   WHERE employee_id = 100;

  UPDATE employees
    SET salary = salary * 1.1
    WHERE employee_id = 100
    RETURNING last_name, salary INTO emp_info;

  DBMS_OUTPUT.PUT_LINE (
    'Salary of ' || emp_info.last_name || ' raised from ' ||
    old_salary || ' to ' || emp_info.salary
  );
END;
/
```

Result:

```
Salary of King raised from 26460 to 29106
```

Record Comparisons

Records cannot be tested natively for nullity, equality, or inequality. These `BOOLEAN` expressions are illegal:

- `My_Record IS NULL`

- `My_Record_1 = My_Record_2`
- `My_Record_1 > My_Record_2`

You must write your own functions to implement such tests. For information about writing functions, see Chapter 8, "PL/SQL Subprograms."

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 13-96. For restrictions on inserting records into tables, see "Restrictions on Record Inserts and Updates" on page 5-52.

Example 5–46 creates the table `schedule` and initializes it by putting default values in a record and inserting the record into the table for each week. (The `COLUMN` formatting commands are from SQL*Plus.)

Example 5–46 Initializing Table by Inserting 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       WEED      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 "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 13-136. For restrictions on updating table rows with a record, see "Restrictions on Record Inserts and Updates" on page 5-52.

Example 5–47 updates the first three weeks of the table `schedule` (defined in Example 5–46) by putting the new values in a record and updating the first three rows of the table with that record.

Example 5–47 Updating Rows with 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       WEED      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 "FORALL Statement" on page 12-11.

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` type
  - Record inserts and updates using the `EXECUTE IMMEDIATE` statement.

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
- Cursors
- Query Result Set Processing
- Cursor Variables
- CURSOR Expressions
- Transaction Processing and Control
- Autonomous Transactions

See Also: "Resolution of Names in Static SQL Statements" on page B-5

Description of Static SQL

Static SQL has the same syntax as SQL, except as noted.

Topics

- Statements
- Pseudocolumns

Statements

These are the PL/SQL static SQL statements, which have the same syntax as the corresponding SQL statements, except as noted:

- `SELECT` (this statement is also called a **query**)

  For the PL/SQL syntax, see "SELECT INTO Statement" on page 13-125.

- Data manipulation language (DML) statements:
  - `INSERT`

    For the PL/SQL syntax, see "INSERT Statement Extension" on page 13-96
  - `UPDATE`

For the PL/SQL syntax, see "UPDATE Statement Extensions" on page 13-136

– `DELETE`

For the PL/SQL syntax, see "DELETE Statement Extension" on page 13-45

– `MERGE` (for syntax, see *Oracle Database SQL Language Reference*)

Note: *Oracle Database SQL Language Reference* defines DML differently.)

- Transaction control language (TCL) statements:
  - `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*)
- `LOCK TABLE` (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.

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.

Pseudocolumns

A pseudocolumn behaves like a table column, but it is not stored in the table. For general information about pseudocolumns, including restrictions, see *Oracle Database SQL Language Reference*.

Static SQL includes these SQL pseudocolumns:

- `CURRVAL` and `NEXTVAL`, described in "CURRVAL and NEXTVAL in PL/SQL" on page 6-3.
- `LEVEL`, described in *Oracle Database SQL Language Reference*
- `OBJECT_VALUE`, described in *Oracle Database SQL Language Reference*

  > **See Also:** "OBJECT_VALUE Pseudocolumn" on page 9-9 for information about using `OBJECT_VALUE` in triggers

- `ROWID`, described in *Oracle Database SQL Language Reference*

  > **See Also:** "Simulating CURRENT OF Clause with ROWID Pseudocolumn" on page 6-48

- `ROWNUM`, described in *Oracle Database SQL Language Reference*

CURRVAL and NEXTVAL in PL/SQL

After a sequence is created, you can access its values in SQL statements with the `CURRVAL` pseudocolumn, which returns the current value of the sequence, or the `NEXTVAL` pseudocolumn, which increments the sequence and returns the new value. (For general information about sequences, see *Oracle Database SQL Language Reference*.)

To reference these pseudocolumns, use dot notation—for example, *`sequence_name`*`.CURRVAL`. For complete syntax, see *Oracle Database SQL Language Reference*.

> **Note:** Each time you reference *`sequence_name`*`.NEXTVAL`, the sequence is incremented immediately and permanently, whether you commit or roll back the transaction.

As of Oracle Database 11*g* Release 1 (11.1), you can use *`sequence_name`*`.CURRVAL` and *`sequence_name`*`.NEXTVAL` in a PL/SQL expression wherever you can use a `NUMBER` expression. However:

- Using *sequence_name*.CURRVAL or *sequence_name*.NEXTVAL to provide a default 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 for every row in which it appears).

Example 6–2 generates a sequence number for the sequence HR.EMPLOYEES_SEQ and refers to that number in multiple statements.

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;
/
```

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), which are explained in *Oracle Database Concepts*.

A session cursor that is constructed and managed by PL/SQL is an **implicit cursor**. A session cursor that you construct and manage is an **explicit cursor**.

You can get information about any session cursor from its attributes (which you can reference 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`, explained in *Oracle Database Reference*.

> **Note:** Generally, PL/SQL parses an explicit cursor only the first time the session opens it 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 an explicit cursor before you can reopen it, PL/SQL need not reparse the associated query. If you close and immediately reopen an explicit cursor, PL/SQL does not reparse the associated query.

Topics

- Implicit Cursors
- Explicit Cursors

Implicit Cursors

An **implicit cursor** is a session cursor that is constructed and managed by PL/SQL. 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 invocations, 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 "Getting Number of Rows Affected by FORALL Statement" on page 12-21)
- SQL%BULK_EXCEPTIONS (see "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-19)

> **See Also:** "Implicit Cursor Attribute" on page 13-91 for complete syntax and semantics

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 Implicit Cursor 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;
/
```

Result:

```
Delete succeeded for department number 270
No department number 400
```

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 Implicit Cursor 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

An **explicit cursor** is a session cursor that you construct and manage. You must declare and define an explicit cursor, giving it a name and associating it with a query (typically, the query returns multiple rows). Then you can process the query result set in either of these ways:

- Open the explicit cursor (with the OPEN statement), fetch rows from the result set (with the FETCH statement), and close the explicit cursor (with the CLOSE statement).
- Use the explicit cursor in a cursor FOR LOOP statement (see "Query Result Set Processing With Cursor FOR LOOP Statements" on page 6-24).

You cannot assign a value to an explicit cursor, use it in an expression, or use it as a formal subprogram parameter or host variable. You *can* do those things with a cursor variable (see "Cursor Variables" on page 6-28).

Unlike an implicit cursor, you can reference an explicit cursor or cursor variable by its name. Therefore, an explicit cursor or cursor variable is called a **named cursor**.

Topics

- Declaring and Defining Explicit Cursors
- Opening and Closing Explicit Cursors
- Fetching Data with Explicit Cursors
- Variables in Explicit Cursor Queries
- When Explicit Cursor Queries Need Column Aliases
- Explicit Cursors that Accept Parameters
- Explicit Cursor Attributes

Declaring and Defining Explicit Cursors

You can either declare an explicit cursor first and then define it later in the same block, subprogram, or package, or declare and define it at the same time.

An **explicit cursor declaration**, which only declares a cursor, has this syntax:

```
CURSOR cursor_name [ parameter_list ] RETURN return_type;
```

An **explicit cursor definition** has this syntax:

```
CURSOR cursor_name [ parameter_list ] [ RETURN return_type ]
  IS select_statement;
```

If you declared the cursor earlier, then the explicit cursor definition defines it; otherwise, it both declares and defines it.

Example 6–5 declares and defines three explicit cursors.

Example 6–5 Explicit Cursor Declaration and Definition

```
DECLARE
  CURSOR c1 RETURN departments%ROWTYPE;    -- Declare c1

  CURSOR c2 IS                             -- Declare and define c2
    SELECT employee_id, job_id, salary FROM employees
    WHERE salary > 2000;

  CURSOR c1 RETURN departments%ROWTYPE IS  -- Define c1,
    SELECT * FROM departments              -- repeating return type
    WHERE department_id = 110;

  CURSOR c3 RETURN locations%ROWTYPE;      -- Declare c3

  CURSOR c3 IS                             -- Define c3,
    SELECT * FROM locations                -- omitting return type
    WHERE country_id = 'JP';
BEGIN
  NULL;
END;
/
```

See Also:

- "Explicit Cursor Declaration and Definition" on page 13-57 for the complete syntax and semantics of explicit cursor declaration and definition
- "Explicit Cursors that Accept Parameters" on page 6-15

Opening and Closing Explicit Cursors

After declaring and defining an explicit cursor, you can open it with the `OPEN` statement, which does the following:

1. Allocates database resources to process the query
2. Processes the query; that is:
   1. Identifies the result set

      If the query references variables or cursor parameters, their values affect the result set. For details, see "Variables in Explicit Cursor Queries" on page 6-12 and "Explicit Cursors that Accept Parameters" on page 6-15.
   2. If the query has a `FOR UPDATE` clause, locks the rows of the result set

      For details, see "SELECT FOR UPDATE and FOR UPDATE Cursors" on page 6-46.
3. Positions the cursor before the first row of the result set

You close an open explicit cursor with the `CLOSE` statement, thereby allowing its resources to be reused. After closing a cursor, you cannot fetch records from its result set or reference its attributes. If you try, PL/SQL raises the predefined exception `INVALID_CURSOR`.

You can reopen a closed cursor. You must close an explicit cursor before you try to reopen it. Otherwise, PL/SQL raises the predefined exception CURSOR_ALREADY_OPEN.

See Also:

- "OPEN Statement" on page 13-101 for its syntax and semantics
- "CLOSE Statement" on page 13-23 for its syntax and semantics

Fetching Data with Explicit Cursors

After opening an explicit cursor, you can fetch the rows of the query result set with the FETCH statement. The basic syntax of a FETCH statement that returns one row is:

```
FETCH cursor_name INTO into_clause
```

The *into_clause* is either a list of variables or a single record variable. For each column that the query returns, the variable list or record must have a corresponding type-compatible variable or field. The %TYPE and %ROWTYPE attributes are useful for declaring variables and records for use in FETCH statements.

The FETCH statement retrieves the current row of the result set, stores the column values of that row into the variables or record, and advances the cursor to the next row.

Typically, you use the FETCH statement inside a LOOP statement, which you exit when the FETCH statement runs out of rows. To detect this exit condition, use the cursor attribute %NOTFOUND (described in "%NOTFOUND Attribute: Has No Row Been Fetched?" on page 6-21). PL/SQL does not raise an exception when a FETCH statement returns no rows.

Example 6–6 fetches the result sets of two explicit cursors one row at a time, using FETCH and %NOTFOUND inside LOOP statements. The first FETCH statement retrieves column values into variables. The second FETCH statement retrieves column values into a record. The variables and record are declared with %TYPE and %ROWTYPE, respectively.

Example 6–6 FETCH Statements Inside LOOP Statements

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK')
    ORDER BY last_name;

  v_lastname  employees.last_name%TYPE;  -- variable for last_name
  v_jobid     employees.job_id%TYPE;     -- variable for job_id

  CURSOR c2 IS
    SELECT * FROM employees
    WHERE REGEXP_LIKE (job_id, '[ACADFIMKSA]_M[ANGR]')
    ORDER BY job_id;

  v_employees employees%ROWTYPE;  -- record variable for row of table

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:

```
Atkinson                  ST_CLERK
Bell                      SH_CLERK
Bissot                    ST_CLERK
...
Walsh                     SH_CLERK
-------------------------------------
Higgins                   AC_MGR
Greenberg                 FI_MGR
Hartstein                 MK_MAN
...
Zlotkey                   SA_MAN
```

Example 6–7 fetches the first five rows of a result set into five records, using five `FETCH` statements, each of which fetches into a different record variable. The record variables are declared with `%ROWTYPE`.

Example 6–7 Fetching Same Explicit Cursor into Different Variables

```
DECLARE
  CURSOR c IS
    SELECT e.job_id, j.job_title
    FROM employees e, jobs j
    WHERE e.job_id = j.job_id AND e.manager_id = 100
    ORDER BY last_name;

  -- Record variables for rows of cursor result set:

  job1 c%ROWTYPE;
  job2 c%ROWTYPE;
  job3 c%ROWTYPE;
  job4 c%ROWTYPE;
  job5 c%ROWTYPE;

BEGIN
  OPEN c;
  FETCH c INTO job1;  -- fetches first row
  FETCH c INTO job2;  -- fetches second row
  FETCH c INTO job3;  -- fetches third row
  FETCH c INTO job4;  -- fetches fourth row
  FETCH c INTO job5;  -- fetches fifth row
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE(job1.job_title || ' (' || job1.job_id || ')');
```

```
  DBMS_OUTPUT.PUT_LINE(job2.job_title || ' (' || job2.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job3.job_title || ' (' || job3.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job4.job_title || ' (' || job4.job_id || ')');
  DBMS_OUTPUT.PUT_LINE(job5.job_title || ' (' || job5.job_id || ')');
END;
/
```

Result:

```
Sales Manager (SA_MAN)
Administration Vice President (AD_VP)
Sales Manager (SA_MAN)
Stock Manager (ST_MAN)
Marketing Manager (MK_MAN)

PL/SQL procedure successfully completed.
```

See Also:

- "FETCH Statement" on page 13-71 for its complete syntax and semantics
- "FETCH Statement with BULK COLLECT Clause" on page 12-31 for information about `FETCH` statements that return more than one row at a time

Variables in Explicit Cursor Queries

An explicit cursor query can reference any variable in its scope. When you open an explicit cursor, PL/SQL evaluates any variables in the query and uses those values when identifying the result set. Changing the values of the variables later does not change the result set.

In Example 6–8, the explicit cursor query references the variable `factor`. When the cursor opens, `factor` has the value 2. Therefore, `sal_multiple` is always 2 times `sal`, despite that `factor` is incremented after every fetch.

Example 6–8 Variable in Explicit Cursor Query—No Result Set Change

```
DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  CURSOR c1 IS
    SELECT salary, salary*factor FROM employees
    WHERE job_id LIKE 'AD_%';

BEGIN
  OPEN c1;  -- PL/SQL evaluates factor

  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
    factor := factor + 1;  -- Does not affect sal_multiple
  END LOOP;

  CLOSE c1;
```

```
END;
/
```

Result:

```
factor = 2
sal          = 4451
sal_multiple = 8902
factor = 3
sal          = 26460
sal_multiple = 52920
factor = 4
sal          = 18742.5
sal_multiple = 37485
factor = 5
sal          = 18742.5
sal_multiple = 37485
```

To change the result set, you must close the cursor, change the value of the variable, and then open the cursor again, as in Example 6–9.

Example 6–9 Variable in Explicit Cursor Query—Result Set Change

```
DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  CURSOR c1 IS
    SELECT salary, salary*factor FROM employees
    WHERE job_id LIKE 'AD_%';

BEGIN
  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
  OPEN c1;  -- PL/SQL evaluates factor
  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;
  CLOSE c1;

  factor := factor + 1;

  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
  OPEN c1;  -- PL/SQL evaluates factor
  LOOP
    FETCH c1 INTO sal, sal_multiple;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;
  CLOSE c1;
END;
/
```

Result:

```
factor = 2
sal          = 4451
```

```
sal_multiple = 8902
sal          = 26460
sal_multiple = 52920
sal          = 18742.5
sal_multiple = 37485
sal          = 18742.5
sal_multiple = 37485
factor = 3
sal          = 4451
sal_multiple = 13353
sal          = 26460
sal_multiple = 79380
sal          = 18742.5
sal_multiple = 56227.5
sal          = 18742.5
sal_multiple = 56227.5
```

When Explicit Cursor Queries Need Column Aliases

When an explicit cursor query 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 6–10, the calculated column in the explicit cursor needs an alias for both of the preceding reasons.

Example 6–10 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'
    ORDER BY employee_id;
  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;
/
```

Result:

```
Raise for employee #114 is $550
Raise for employee #120 is $533.61
Raise for employee #121 is $520.905
Raise for employee #122 is $501.8475
Raise for employee #123 is $412.9125
Raise for employee #124 is $368.445
Raise for employee #145 is $700
```

```
Raise for employee #146 is $675
Raise for employee #147 is $600
Raise for employee #148 is $550
Raise for employee #149 is $525
Raise for employee #201 is $650
```

See Also: Example 6–21, "Cursor FOR Loop References Calculated Columns"

Explicit Cursors that Accept Parameters

You can create an explicit cursor that has formal parameters, and then pass different actual parameters to the cursor each time you open it. In the cursor query, you can use a formal cursor parameter anywhere that you can use a constant. Outside the cursor query, you cannot reference formal cursor parameters.

Tip: To avoid confusion, use different names for formal and actual cursor parameters.

Example 6–11 creates an explicit cursor whose two formal parameters represent a job and its maximum salary. When opened with a specified job and maximum salary, the cursor query selects the employees with that job who are overpaid (for each such employee, the query selects the first and last name and amount overpaid). Next, the example creates a procedure that prints the cursor query result set (for information about procedures, see Chapter 8, "PL/SQL Subprograms"). Finally, the example opens the cursor with one set of actual parameters, prints the result set, closes the cursor, opens the cursor with different actual parameters, prints the result set, and closes the cursor.

Example 6–11 Explicit Cursor that Accepts Parameters

```
DECLARE
  CURSOR c (job VARCHAR2, max_sal NUMBER) IS
    SELECT last_name, first_name, (salary - max_sal) overpayment
    FROM employees
    WHERE job_id = job
    AND salary > max_sal
    ORDER BY salary;

  PROCEDURE print_overpaid IS
    last_name_   employees.last_name%TYPE;
    first_name_  employees.first_name%TYPE;
    overpayment_      employees.salary%TYPE;
  BEGIN
    LOOP
      FETCH c INTO last_name_, first_name_, overpayment_;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(last_name_ || ', ' || first_name_ ||
        ' (by ' || overpayment_ || ')');
    END LOOP;
  END print_overpaid;

BEGIN
  DBMS_OUTPUT.PUT_LINE('----------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Stock Clerks:');
  DBMS_OUTPUT.PUT_LINE('----------------------');
  OPEN c('ST_CLERK', 5000);
  print_overpaid;
  CLOSE c;
```

```
  DBMS_OUTPUT.PUT_LINE('------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives:');
  DBMS_OUTPUT.PUT_LINE('------------------------------');
  OPEN c('SA_REP', 10000);
  print_overpaid;
  CLOSE c;
END;
/
```

Result:

```
----------------------
Overpaid Stock Clerks:
----------------------
Davies, Curtis (by 15.3)
Nayer, Julia (by 177.08)
Stiles, Stephen (by 177.08)
Bissot, Laura (by 338.87)
Mallin, Jason (by 338.87)
Rajs, Trenna (by 662.43)
Ladwig, Renske (by 824.21)
------------------------------
Overpaid Sales Representatives:
------------------------------
Fox, Tayler (by 80)
Tucker, Peter (by 500)
King, Janette (by 500)
Bloom, Harrison (by 500)
Vishney, Clara (by 1025)
Abel, Ellen (by 1550)
Ozer, Lisa (by 2075)

PL/SQL procedure successfully completed.
```

Topics

- Formal Cursor Parameters with Default Values
- Adding Formal Cursor Parameters with Default Values

See Also:

- "Explicit Cursor Declaration and Definition" on page 13-57 for more information about formal cursor parameters
- "OPEN Statement" on page 13-101 for more information about actual cursor parameters

Formal Cursor Parameters with Default Values When you create an explicit cursor with formal parameters, you can specify default values for them. When a formal parameter has a default value, its corresponding actual parameter is optional. If you open the cursor without specifying the actual parameter, then the formal parameter has its default value.

Example 6–12 creates an explicit cursor whose formal parameter represents a location ID. The default value of the parameter is the location ID of company headquarters.

Example 6–12 Cursor Parameters with Default Values

```
DECLARE
  CURSOR c (location NUMBER DEFAULT 1700) IS
```

```
    SELECT d.department_name,
           e.last_name manager,
           l.city
    FROM departments d, employees e, locations l
    WHERE l.location_id = location
      AND l.location_id = d.location_id
      AND d.department_id = e.department_id
    ORDER BY d.department_id;

  PROCEDURE print_depts IS
    dept_name  departments.department_name%TYPE;
    mgr_name   employees.last_name%TYPE;
    city_name  locations.city%TYPE;
  BEGIN
    LOOP
      FETCH c INTO dept_name, mgr_name, city_name;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(dept_name || ' (Manager: ' || mgr_name || ')');
    END LOOP;
  END print_depts;

BEGIN
  DBMS_OUTPUT.PUT_LINE('DEPARTMENTS AT HEADQUARTERS:');
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  OPEN c;
  print_depts;
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('DEPARTMENTS IN CANADA:');
  DBMS_OUTPUT.PUT_LINE('--------------------------------');
  OPEN c(1800); -- Toronto
  print_depts;
  CLOSE c;
  OPEN c(1900); -- Whitehorse
  print_depts;
  CLOSE c;
END;
/
```

Result:

```
DEPARTMENTS AT HEADQUARTERS:
--------------------------------
Administration (Manager: Whalen)
Purchasing (Manager: Colmenares)
Purchasing (Manager: Baida)
Purchasing (Manager: Himuro)
Purchasing (Manager: Raphaely)
Purchasing (Manager: Khoo)
Purchasing (Manager: Tobias)
Executive (Manager: Kochhar)
Executive (Manager: De Haan)
Executive (Manager: King)
Finance (Manager: Popp)
Finance (Manager: Greenberg)
Finance (Manager: Faviet)
Finance (Manager: Chen)
Finance (Manager: Urman)
Finance (Manager: Sciarra)
Accounting (Manager: Gietz)
```

```
Accounting (Manager: Higgins)
--------------------------------
DEPARTMENTS IN CANADA:
--------------------------------
Marketing (Manager: Hartstein)
Marketing (Manager: Fay)

PL/SQL procedure successfully completed.
```

Adding Formal Cursor Parameters with Default Values If you add formal parameters to a cursor, and you specify default values for the added parameters, then you need not change existing references to the cursor. Compare Example 6–13 to Example 6–11.

Example 6–13 Adding Formal Parameter to Existing Cursor

```
DECLARE
  CURSOR c (job VARCHAR2, max_sal NUMBER, hired DATE DEFAULT '31-DEC-99') IS
    SELECT last_name, first_name, (salary - max_sal) overpayment
    FROM employees
    WHERE job_id = job
    AND salary > max_sal
    AND hire_date > hired
    ORDER BY salary;

  PROCEDURE print_overpaid IS
    last_name_   employees.last_name%TYPE;
    first_name_  employees.first_name%TYPE;
    overpayment_      employees.salary%TYPE;
  BEGIN
    LOOP
      FETCH c INTO last_name_, first_name_, overpayment_;
      EXIT WHEN c%NOTFOUND;
      DBMS_OUTPUT.PUT_LINE(last_name_ || ', ' || first_name_ ||
        ' (by ' || overpayment_ || ')');
    END LOOP;
  END print_overpaid;

BEGIN
  DBMS_OUTPUT.PUT_LINE('-------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives:');
  DBMS_OUTPUT.PUT_LINE('-------------------------------');
  OPEN c('SA_REP', 10000);  -- existing reference
  print_overpaid;
  CLOSE c;

  DBMS_OUTPUT.PUT_LINE('-----------------------------------------------');
  DBMS_OUTPUT.PUT_LINE('Overpaid Sales Representatives Hired After 2004:');
  DBMS_OUTPUT.PUT_LINE('-----------------------------------------------');
  OPEN c('SA_REP', 10000, '31-DEC-04');  -- new reference
  print_overpaid;
  CLOSE c;
END;
/
```

Result:

```
--------------------------------
Overpaid Sales Representatives:
--------------------------------
Fox, Tayler (by 80)
```

```
Tucker, Peter (by 500)
King, Janette (by 500)
Bloom, Harrison (by 500)
Vishney, Clara (by 1025)
Abel, Ellen (by 1550)
Ozer, Lisa (by 2075)
-----------------------------------------------
Overpaid Sales Representatives Hired After 2004:
-----------------------------------------------
Fox, Tayler (by 80)
Tucker, Peter (by 500)
Bloom, Harrison (by 500)
Vishney, Clara (by 1025)
Ozer, Lisa (by 2075)

PL/SQL procedure successfully completed.
```

Explicit Cursor Attributes

The syntax for the value of an explicit cursor attribute is *cursor_name* immediately followed by *attribute* (for example, `c1%ISOPEN`).

> **Note:** Explicit cursors and cursor variables (named cursors) have the same attributes. This topic applies to all named cursors except where noted.

The explicit 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 an explicit cursor is not open, referencing any attribute except `%ISOPEN` raises the predefined exception `INVALID_CURSOR`.

> **See Also:** "Named Cursor Attribute" on page 13-98 for complete syntax and semantics of named cursor (explicit cursor and cursor variable) attributes

%ISOPEN Attribute: Is the Cursor Open? `%ISOPEN` returns `TRUE` if its explicit cursor is open; `FALSE` otherwise.

`%ISOPEN` is useful for:

- Checking that an explicit cursor is not already open before you try to open it.

  If you try to open an explicit cursor that is already open, PL/SQL raises the predefined exception `CURSOR_ALREADY_OPEN`. You must close an explicit cursor before you can reopen it.

  > **Note:** The preceding paragraph does not apply to cursor variables.

- Checking that an explicit cursor is open before you try to close it.

Example 6–14 opens the explicit cursor `c1` only if it is not open and closes it only if it is open.

Example 6–14 %ISOPEN Explicit Cursor 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 NOT c1%ISOPEN THEN
    OPEN c1;
  END IF;

  FETCH c1 INTO the_name, the_salary;

  IF c1%ISOPEN THEN
    CLOSE c1;
  END IF;
END;
/
```

%FOUND Attribute: Has a Row Been Fetched? `%FOUND` returns:

- `NULL` after the explicit cursor is opened but before the first fetch
- `TRUE` if the most recent fetch from the explicit cursor returned a row
- `FALSE` otherwise

`%FOUND` is useful for determining whether there is a fetched row to process.

Example 6–15 loops through a result set, printing each fetched row and exiting when there are no more rows to fetch.

Example 6–15 %FOUND Explicit Cursor Attribute

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11
    ORDER BY last_name;

  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 = Abel, salary = 11000
Name = Ande, salary = 6400
Name = Atkinson, salary = 3557.4
Name = Austin, salary = 4800
Name = Baer, salary = 10000
Name = Baida, salary = 2900
Name = Banda, salary = 6200
Name = Bates, salary = 7300
Name = Bell, salary = 5082
Name = Bernstein, salary = 9500
```

%NOTFOUND Attribute: Has No Row Been Fetched? `%NOTFOUND` (the logical opposite of `%FOUND`) returns:

- `NULL` after the explicit cursor is opened but before the first fetch
- `FALSE` if the most recent fetch from the explicit cursor returned a row
- `TRUE` otherwise

`%NOTFOUND` is useful for exiting a loop when `FETCH` fails to return a row, as in Example 6–16.

Example 6–16 %NOTFOUND Explicit Cursor Attribute

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, salary FROM employees
    WHERE ROWNUM < 11
    ORDER BY last_name;

   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
      EXIT;
    ELSE  -- fetch succeeded
      DBMS_OUTPUT.PUT_LINE
        ('Name = ' || my_ename || ', salary = ' || my_salary);
    END IF;
  END LOOP;
END;
/
```

Result:

```
Name = Abel, salary = 11000
Name = Ande, salary = 6400
Name = Atkinson, salary = 3557.4
Name = Austin, salary = 4800
Name = Baer, salary = 10000
Name = Baida, salary = 2900
Name = Banda, salary = 6200
Name = Bates, salary = 7300
Name = Bell, salary = 5082
Name = Bernstein, salary = 9500
```

> **Note:** In Example 6–16, if `FETCH` never fetches a row, then `c1%NOTFOUND` is always `NULL` and the loop is never exited. 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 explicit cursor is opened but before the first fetch
- Otherwise, the number of rows fetched

Example 6–17 numbers and prints the rows that it fetches and prints a message after fetching the fifth row.

Example 6–17 %ROWCOUNT Explicit Cursor Attribute

```
DECLARE
  CURSOR c1 IS
    SELECT last_name FROM employees
    WHERE ROWNUM < 11
    ORDER BY last_name;

  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 row ---');
    END IF;
  END LOOP;
  CLOSE c1;
END;
/
```

Result:

```
1. Abel
2. Ande
3. Atkinson
4. Austin
5. Baer
--- Fetched 5th row ---
6. Baida
7. Banda
8. Bates
9. Bell
10. Bernstein
```

Query Result Set Processing

In PL/SQL, as in traditional database programming, you use cursors to process query result sets. However, in PL/SQL, you can use either implicit or explicit cursors. The former need less code, but the latter are more flexible. For example, explicit cursors can accept parameters (see "Explicit Cursors that Accept Parameters" on page 6-15).

The following PL/SQL statements use implicit cursors that PL/SQL defines and manages for you:

- `SELECT INTO`
- Implicit cursor `FOR LOOP`

The following PL/SQL statements use explicit cursors:

- Explicit cursor `FOR LOOP`

  You define the explicit cursor, but PL/SQL manages it while the statement runs.
- `OPEN`, `FETCH`, and `CLOSE`

  You define and manage the explicit cursor.

Topics

- Query Result Set Processing With SELECT INTO Statements
- Query Result Set Processing With Cursor FOR LOOP Statements
- Query Result Set Processing With Explicit Cursors, OPEN, FETCH, and CLOSE
- Query Result Set Processing with Subqueries

Note: If a query returns no rows, PL/SQL raises the exception `NO_DATA_FOUND`. For information about handling exceptions, see "Exception Handler" on page 13-50.

Query Result Set Processing With SELECT INTO Statements

Using an implicit cursor, the `SELECT INTO` statement retrieves values from one or more database tables (as the SQL `SELECT` statement does) and stores them in variables (which the SQL `SELECT` statement does not do).

Topics

- Single-Row Result Sets
- Large Multiple-Row Result Sets

See Also: "SELECT INTO Statement" on page 13-125 for its complete syntax and semantics

Single-Row Result Sets

If you expect the query to return only one row, then use the `SELECT INTO` statement to store values from that row in either one or more scalar variables (see "Assigning Values to Variables with the SELECT INTO Statement" on page 2-22) or one record variable (see "SELECT INTO Statement for Assigning Row to Record Variable" on page 5-47).

If the query might return multiple rows, but you care about only the *n*th row, then restrict the result set to that row with the clause `WHERE ROWNUM=`*n*. For more information about the `ROWNUM` pseudocolumn, see *Oracle Database SQL Language Reference*.

Large Multiple-Row Result Sets

If you must assign a large quantity of table data to variables, Oracle recommends using the `SELECT INTO` statement with the `BULK COLLECT` clause. This statement

retrieves an entire result set into one or more collection variables. For more information, see "SELECT INTO Statement with BULK COLLECT Clause" on page 12-24.

Query Result Set Processing With Cursor FOR LOOP Statements

The cursor `FOR LOOP` statement lets you run a `SELECT` statement and then immediately loop through the rows of the result set. This statement can use either an implicit or explicit cursor.

If you use the `SELECT` statement only in the cursor `FOR LOOP` statement, then specify the `SELECT` statement inside the cursor `FOR LOOP` statement, as in Example 6–18. This form of the cursor `FOR LOOP` statement uses an implicit cursor, and is called an **implicit cursor `FOR LOOP` statement**. Because the implicit cursor is internal to the statement, you cannot reference it with the name `SQL`.

If you use the `SELECT` statement multiple times in the same PL/SQL unit, then define an explicit cursor for it and specify that cursor in the cursor `FOR LOOP` statement, as in Example 6–19. This form of the cursor `FOR LOOP` statement is called an **explicit cursor `FOR LOOP` statement**. You can use the same explicit cursor elsewhere in the same PL/SQL unit.

The cursor `FOR LOOP` statement implicitly declares its loop index as a `%ROWTYPE` record variable of the type that its cursor returns. This record is local to the loop and exists only during loop execution. Statements inside the loop can reference the record and its fields. They can reference calculated columns only by aliases, as in Example 6–21.

After declaring the loop index record variable, the `FOR LOOP` statement opens the specified cursor. With each iteration of the loop, the `FOR LOOP` statement fetches a row from the result set and stores it in 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 PL/SQL raises an exception.

> **See Also:** "Cursor FOR LOOP Statement" on page 13-40 for its complete syntax and semantics

In Example 6–18, an implicit cursor `FOR LOOP` statement prints the last name and job ID of every clerk whose manager has an ID greater than 120.

Example 6–18 Implicit Cursor FOR LOOP Statement

```
BEGIN
  FOR item IN (
    SELECT last_name, job_id
    FROM employees
    WHERE job_id LIKE '%CLERK%'
    AND manager_id > 120
    ORDER BY last_name
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/
```

Result:

```
Name = Atkinson, Job = ST_CLERK
Name = Bell, Job = SH_CLERK
Name = Bissot, Job = ST_CLERK
...
Name = Walsh, Job = SH_CLERK
```

Example 6–19 is like Example 6–18, except that it uses an explicit cursor `FOR LOOP` statement.

Example 6–19 Explicit Cursor FOR LOOP Statement

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id FROM employees
    WHERE job_id LIKE '%CLERK%' AND manager_id > 120
    ORDER BY last_name;
BEGIN
  FOR item IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/
```

Result:

```
Name = Atkinson, Job = ST_CLERK
Name = Bell, Job = SH_CLERK
Name = Bissot, Job = ST_CLERK
...
Name = Walsh, Job = SH_CLERK
```

Example 6–20 declares and defines an explicit cursor that accepts two parameters, and then uses it in an explicit cursor `FOR LOOP` statement to display the wages paid to employees who earn more than a specified wage in a specified department.

Example 6–20 Passing Parameters to Explicit Cursor FOR LOOP Statement

```
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('ST_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;
/
```

Result:

```
Name = Nayer, salary = 4065.6, Job Id = ST_CLERK
Name = Mikkilineni, salary = 3430.35, Job Id = ST_CLERK
Name = Landry, salary = 3049.2, Job Id = ST_CLERK
```

```
...
Name = Vargas, salary = 3176.25, Job Id = ST_CLERK
```

In Example 6–21, the implicit cursor `FOR LOOP` references calculated columns by their aliases, `full_name` and `dream_salary`.

Example 6–21 Cursor FOR Loop References Calculated Columns

```
BEGIN
  FOR item IN (
    SELECT first_name || ' ' || last_name AS full_name,
           salary * 10                    AS dream_salary
    FROM employees
    WHERE ROWNUM <= 5
    ORDER BY dream_salary DESC, last_name ASC
  ) LOOP
    DBMS_OUTPUT.PUT_LINE
      (item.full_name || ' dreams of making ' || item.dream_salary);
  END LOOP;
END;
/
```

Result:

```
Michael Hartstein dreams of making 143325
Pat Fay dreams of making 66150
Jennifer Whalen dreams of making 48510
Douglas Grant dreams of making 31531.5
Donald OConnell dreams of making 31531.5
```

Note: When an exception is raised inside a cursor `FOR LOOP` statement, the cursor closes before the exception handler runs. Therefore, the values of explicit cursor attributes are not available in the handler.

Query Result Set Processing With Explicit Cursors, OPEN, FETCH, and CLOSE

For full control over query result set processing, declare explicit cursors and manage them with the statements `OPEN`, `FETCH`, and `CLOSE`. (For instructions and examples, see "Explicit Cursors" on page 6-8.)

This result set processing technique is more complicated than the others, but it is also more flexible. For example, you can:

- Process multiple result sets in parallel, using multiple cursors.
- Process multiple rows in a single loop iteration, skip rows, or split the processing into multiple loops.
- Specify the query in one PL/SQL unit but retrieve the rows in another.

Query Result Set Processing with Subqueries

If you process a query result set by looping through it and running another query for each row, then you can improve performance by removing the second query from inside the loop and making it a subquery of the first query. For more information about subqueries, see *Oracle Database SQL Language Reference*.

Example 6–22 defines explicit cursor `c1` with a query whose `FROM` clause contains a subquery.

Example 6–22 Subquery in FROM Clause of Parent Query

```
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
    ORDER BY staff;

BEGIN
   FOR dept IN c1
   LOOP
     DBMS_OUTPUT.PUT_LINE ('Department = '
       || dept.department_name || ', staff = ' || dept.staff);
   END LOOP;
END;
/
```

Result:

```
Department = IT, staff = 5
Department = Purchasing, staff = 6
Department = Finance, staff = 6
Department = Sales, staff = 34
Department = Shipping, staff = 45
```

While an ordinary subquery is evaluated for each table, a **correlated subquery** is evaluated 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, last_name;
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 = Bell
...
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.
- You can assign a value to it.
- You can use it in an expression.
- It can be a subprogram parameter.

  You can use cursor variables to pass query result sets between subprograms.
- It can be a host variable.

  You can use cursor variables to pass query result sets between PL/SQL stored subprograms and their clients.
- It cannot accept parameters.

  You cannot pass parameters to a cursor variable, but you can pass whole queries to it.

A cursor variable has this flexibility because it is a pointer; that is, its value is the address of an item, not the item itself.

Before you can reference a cursor variable, you must make it point to a SQL work area, either by opening it or by assigning it the value of an open PL/SQL cursor variable or open host cursor variable.

Note: Cursor variables and explicit cursors 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` statement.

Topics

- Creating Cursor Variables
- Opening and Closing Cursor Variables
- Fetching Data with Cursor Variables
- Assigning Values to Cursor Variables
- Variables in Cursor Variable Queries
- Cursor Variable Attributes
- Cursor Variables as Subprogram Parameters
- Cursor Variables as Host Variables

See Also: "Restrictions on Cursor Variables" on page 13-42

Creating Cursor Variables

To create a cursor variable, either declare a variable of the predefined type `SYS_REFCURSOR` or define a `REF CURSOR` type and then declare a variable of that type.

> **Note:** Informally, a cursor variable is sometimes called a REF CURSOR).

The basic syntax of a REF CURSOR type definition is:

```
TYPE type_name IS REF CURSOR [ RETURN return_type ]
```

(For the complete syntax and semantics, see "Cursor Variable Declaration" on page 13-42.)

If you specify *return_type*, then the REF CURSOR type and cursor variables of that type are **strong**; if not, they are **weak**. SYS_REFCURSOR and cursor variables of that type are weak.

With a strong cursor variable, you can associate only queries that return the specified type. With a weak cursor variable, you can associate any query.

Weak cursor variables are more error-prone than strong ones, but they are also more flexible. Weak REF CURSOR types are interchangeable with each other and with the predefined type SYS_REFCURSOR. You can assign the value of a weak cursor variable to any other weak cursor variable.

You can assign the value of a strong cursor variable to another strong cursor variable only if both cursor variables have the same type (not merely the same return type).

> **Note:** You can partition weak cursor variable arguments to table functions only with the PARTITION BY ANY clause, not with range or hash partitioning.

Example 6–24 defines strong and weak REF CURSOR types, variables of those types, and a variable of the predefined type SYS_REFCURSOR.

Example 6–24 Cursor Variable Declarations

```
DECLARE
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;  -- strong type
  TYPE genericcurtyp IS REF CURSOR;                       -- weak type

  cursor1  empcurtyp;       -- strong cursor variable
  cursor2  genericcurtyp;   -- weak cursor variable
  my_cursor SYS_REFCURSOR;  -- weak cursor variable

  TYPE deptcurtyp IS REF CURSOR RETURN departments%ROWTYPE;  -- strong type
  dept_cv deptcurtyp;  -- strong cursor variable
BEGIN
  NULL;
END;
/
```

In Example 6–25, *return_type* is a user-defined RECORD type.

Example 6–25 Cursor Variable with User-Defined Return 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;
/
```

Opening and Closing Cursor Variables

After declaring a cursor variable, you can open it with the `OPEN FOR` statement, which does the following:

1. Associates the cursor variable with a query (typically, the query returns multiple rows)

   The query can include placeholders for bind variables, whose values you specify in the `USING` clause of the `OPEN FOR` statement.

2. Allocates database resources to process the query
3. Processes the query; that is:
   1. Identifies the result set

      If the query references variables, their values affect the result set. For details, see "Variables in Cursor Variable Queries" on page 6-33.

   2. If the query has a `FOR UPDATE` clause, locks the rows of the result set

      For details, see "SELECT FOR UPDATE and FOR UPDATE Cursors" on page 6-46.

4. Positions the cursor before the first row of the result set

You need not close a cursor variable before reopening it (that is, using it in another `OPEN FOR` statement). After you reopen a cursor variable, the query previously associated with it is lost.

When you no longer need a cursor variable, close it with the `CLOSE` statement, thereby allowing its resources to be reused. After closing a cursor variable, you cannot fetch records from its result set or reference its attributes. If you try, PL/SQL raises the predefined exception `INVALID_CURSOR`.

You can reopen a closed cursor variable.

See Also:

- "OPEN FOR Statement" on page 13-103 for its syntax and semantics
- "CLOSE Statement" on page 13-23 for its syntax and semantics

Fetching Data with Cursor Variables

After opening a cursor variable, you can fetch the rows of the query result set with the `FETCH` statement, which works as described in "Fetching Data with Explicit Cursors" on page 6-10.

The return type of the cursor variable must be compatible with the *`into_clause`* of the `FETCH` statement. If the cursor variable is strong, PL/SQL catches incompatibility at compile time. If the cursor variable is weak, PL/SQL catches incompatibility at run time, raising the predefined exception `ROWTYPE_MISMATCH` before the first fetch.

Example 6–26 uses one cursor variable to do what Example 6–6 does with two explicit cursors. The first OPEN FOR statement includes the query itself. The second OPEN FOR statement references a variable whose value is a query.

Example 6–26 Fetching Data with Cursor Variables

```
DECLARE
  cv SYS_REFCURSOR;  -- cursor variable

  v_lastname  employees.last_name%TYPE;  -- variable for last_name
  v_jobid     employees.job_id%TYPE;     -- variable for job_id

  query_2 VARCHAR2(200) :=
    'SELECT * FROM employees
    WHERE REGEXP_LIKE (job_id, ''[ACADFIMKSA]_M[ANGR]'')
    ORDER BY job_id';

  v_employees employees%ROWTYPE;  -- record variable row of table

BEGIN
  OPEN cv FOR
    SELECT last_name, job_id FROM employees
    WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK')
    ORDER BY last_name;

  LOOP  -- Fetches 2 columns into variables
    FETCH cv INTO v_lastname, v_jobid;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_lastname, 25, ' ') || v_jobid );
  END LOOP;

  DBMS_OUTPUT.PUT_LINE( '-------------------------------------' );

  OPEN cv FOR query_2;

  LOOP  -- Fetches entire row into the v_employees record
    FETCH cv INTO v_employees;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE( RPAD(v_employees.last_name, 25, ' ') ||
                              v_employees.job_id );
  END LOOP;

  CLOSE cv;
END;
/
```

Result:

```
Atkinson                  ST_CLERK
Bell                      SH_CLERK
Bissot                    ST_CLERK
...
Walsh                     SH_CLERK
-------------------------------------
Higgins                   AC_MGR
Greenberg                 FI_MGR
Hartstein                 MK_MAN
...
Zlotkey                   SA_MAN
```

Example 6–27 fetches from a cursor variable into two collections (nested tables), using the BULK COLLECT clause of the FETCH statement.

Example 6–27 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'
    ORDER BY salary DESC;

  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 = Ozer, salary = 12075
Name = Abel, salary = 11550
Name = Vishney, salary = 11025
...
Name = Kumar, salary = 6405
```

See Also:

- "FETCH Statement" on page 13-71 for its complete syntax and semantics
- "FETCH Statement with BULK COLLECT Clause" on page 12-31 for information about FETCH statements that return more than one row at a time

Assigning Values to Cursor Variables

You can assign to a PL/SQL cursor variable the value of another PL/SQL cursor variable or host cursor variable. The syntax is:

```
target_cursor_variable := source_cursor_variable;
```

If *source_cursor_variable* is open, then after the assignment, *target_cursor_variable* is also open. The two cursor variables point to the same SQL work area.

If *source_cursor_variable* is not open, opening *target_cursor_variable* after the assignment does not open *source_cursor_variable*.

Variables in Cursor Variable Queries

The query associated with a cursor variable can reference any variable in its scope. When you open a cursor variable with the `OPEN FOR` statement, PL/SQL evaluates any variables in the query and uses those values when identifying the result set. Changing the values of the variables later does not change the result set.

Example 6–28 opens a cursor variable for a query that references the variable `factor`, which has the value 2. Therefore, `sal_multiple` is always 2 times `sal`, despite that `factor` is incremented after every fetch.

Example 6–28 Variable in Cursor Variable Query—No Result Set Change

```
DECLARE
  sal           employees.salary%TYPE;
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  cv SYS_REFCURSOR;

BEGIN
  OPEN cv FOR
    SELECT salary, salary*factor
    FROM employees
    WHERE job_id LIKE 'AD_%';   -- PL/SQL evaluates factor

  LOOP
    FETCH cv INTO sal, sal_multiple;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('factor = ' || factor);
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
    factor := factor + 1;  -- Does not affect sal_multiple
  END LOOP;

  CLOSE cv;
END;
/
```

Result:

```
factor = 2
sal          = 4451
sal_multiple = 8902
factor = 3
sal          = 26460
sal_multiple = 52920
factor = 4
sal          = 18742.5
sal_multiple = 37485
factor = 5
sal          = 18742.5
sal_multiple = 37485
```

To change the result set, you must change the value of the variable and then open the cursor variable again for the same query, as in Example 6–29.

Example 6–29 Variable in Cursor Variable Query—Result Set Change

```
DECLARE
  sal           employees.salary%TYPE;
```

```
  sal_multiple  employees.salary%TYPE;
  factor        INTEGER := 2;

  cv SYS_REFCURSOR;

BEGIN
  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);

  OPEN cv FOR
    SELECT salary, salary*factor
    FROM employees
    WHERE job_id LIKE 'AD_%';   -- PL/SQL evaluates factor

  LOOP
    FETCH cv INTO sal, sal_multiple;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;

  factor := factor + 1;

  DBMS_OUTPUT.PUT_LINE('factor = ' || factor);

  OPEN cv FOR
    SELECT salary, salary*factor
    FROM employees
    WHERE job_id LIKE 'AD_%';   -- PL/SQL evaluates factor

  LOOP
    FETCH cv INTO sal, sal_multiple;
    EXIT WHEN cv%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('sal          = ' || sal);
    DBMS_OUTPUT.PUT_LINE('sal_multiple = ' || sal_multiple);
  END LOOP;

  CLOSE cv;
END;
/
```

Result:

```
factor = 2
sal          = 4451
sal_multiple = 8902
sal          = 26460
sal_multiple = 52920
sal          = 18742.5
sal_multiple = 37485
sal          = 18742.5
sal_multiple = 37485
factor = 3
sal          = 4451
sal_multiple = 13353
sal          = 26460
sal_multiple = 79380
sal          = 18742.5
sal_multiple = 56227.5
sal          = 18742.5
sal_multiple = 56227.5
```

Cursor Variable Attributes

A cursor variable has the same attributes as an explicit cursor (see "Explicit Cursor Attributes" on page 6-19). The syntax for the value of a cursor variable attribute is *cursor_variable_name* immediately followed by *attribute* (for example, `cv%ISOPEN`). If a cursor variable is not open, referencing any attribute except `%ISOPEN` raises the predefined exception `INVALID_CURSOR`.

Cursor Variables as Subprogram Parameters

You can use a cursor variable as a subprogram parameter, which makes it useful for passing query results between subprograms. For example:

- You can open a cursor variable in one subprogram and process it in a different subprogram.
- In a multilanguage application, a PL/SQL subprogram can use a cursor variable to return a result set to a subprogram written in a different language.

> **Note:** The invoking and invoked subprograms must be in the same database instance. You cannot pass or return cursor variables to subprograms invoked through database links.

> **Caution:** Because cursor variables are pointers, using them as subprogram parameters increases the likelihood of subprogram parameter aliasing, which can have unintended results. For more information, see "Subprogram Parameter Aliasing with Cursor Variable Parameters" on page 8-18.

When declaring a cursor variable as the formal parameter of a subprogram:

- If the subprogram opens or assigns a value to the cursor variable, then the parameter mode must be `IN OUT`.
- If the subprogram only fetches from, or closes, the cursor variable, then the parameter mode can be either `IN` or `IN OUT`.

Corresponding formal and actual cursor variable parameters must have compatible return types. Otherwise, PL/SQL raises the predefined exception `ROWTYPE_MISMATCH`.

To pass a cursor variable parameter between subprograms in different PL/SQL units, define the `REF CURSOR` type of the parameter in a package. When the type is in a package, multiple subprograms can use it. One subprogram can declare a formal parameter of that type, and other subprograms can declare variables of that type and pass them to the first subprogram.

Example 6–30 defines, in a package, a `REF CURSOR` type and a procedure that opens a cursor variable parameter of that type.

Example 6–30 Procedure to Open Cursor Variable for One Query

```
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;
/
```

In Example 6–31,the stored procedure opens its cursor variable parameter for a chosen query. The queries have the same return type.

Example 6–31 Opening Cursor Variable for Chosen Query (Same Return Type)

```
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;
/
```

In Example 6–32,the stored procedure opens its cursor variable parameter for a chosen query. The queries have the different return types.

Example 6–32 Opening Cursor Variable for Chosen Query (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;
```

```
/
```

See Also:

- "Subprogram Parameters" on page 8-9 for more information about subprogram parameters
- "CURSOR Expressions" on page 6-38 for information about `CURSOR` expressions, which can be actual parameters for formal cursor variable parameters
- Chapter 10, "PL/SQL Packages," for more information about packages

Cursor Variables as Host Variables

You can use a cursor variable as a host variable, which makes it useful for passing query results between PL/SQL stored subprograms and their clients. When a cursor variable is a host variable, PL/SQL and the client (the host environment) share a pointer to the SQL work area that stores the result set.

To use a cursor variable as a host variable, declare the cursor variable in the host environment and then pass it as an input host variable (bind variable) to PL/SQL. Host cursor variables are compatible with any query return type (like weak PL/SQL cursor variables).

In Example 6–33, a Pro*C client program declares a cursor variable and a selector and passes them as host variables to a PL/SQL anonymous block, which opens the cursor variable for the selected query.

Example 6–33 Cursor Variable as Host Variable in Pro*C Client Program

```
EXEC SQL BEGIN DECLARE SECTION;
  SQL_CURSOR  generic_cv;  -- Declare host cursor variable.
  int         choice;      -- Declare selector.
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;
```

A SQL 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, in Example 6–33, the Pro*C program passes a host cursor variable to an embedded PL/SQL anonymous block. After the block runs, the cursor variable still points to the SQL work area.

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 on the client side. You can

also reduce network traffic with a PL/SQL anonymous block that opens or closes several host cursor variables in a single round trip. For example:

```
/* PL/SQL anonymous 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;
/
```

Because the cursor variables still point to the SQL work areas after the PL/SQL anonymous block runs, the client program can use them. When the client program no longer needs the cursors, it can use a PL/SQL anonymous block to close them. For example:

```
/* PL/SQL anonymous block in host environment */
BEGIN
  CLOSE :emp_cv;
  CLOSE :dept_cv;
  CLOSE :loc_cv;
END;
/
```

This technique is useful for populating a multiblock form, as in Oracle Forms. For example, you can open several SQL work areas in a single round trip, like this:

```
/* PL/SQL anonymous block in host environment */
BEGIN
  OPEN :c1 FOR SELECT 1 FROM DUAL;
  OPEN :c2 FOR SELECT 1 FROM DUAL;
  OPEN :c3 FOR SELECT 1 FROM DUAL;
END;
/
```

Note: If you bind a host cursor variable into PL/SQL from an Oracle Call Interface (OCI) client, then you cannot fetch from it on the server side unless you also open it there on the same server call.

CURSOR Expressions

A `CURSOR` expression returns a nested cursor. It has this syntax:

```
CURSOR ( subquery )
```

You can use a `CURSOR` expression in a `SELECT` statement that is not a subquery (as in Example 6–34) or pass it to a function that accepts a cursor variable parameter (see "Passing CURSOR Expressions to Pipelined Table Functions" on page 12-44). You cannot use a cursor expression with an implicit cursor.

See Also: *Oracle Database SQL Language Reference* for more information about `CURSOR` expressions, including restrictions

Example 6–34 declares and defines an explicit cursor for 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 6–34 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_name,
      CURSOR ( SELECT e.last_name
               FROM employees e
               WHERE e.department_id = d.department_id
               ORDER BY e.last_name
             ) employees
    FROM departments d
    WHERE department_name LIKE 'A%'
    ORDER BY department_name;
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: Accounting
-- Employee: Gietz
-- Employee: Higgins
Department: Administration
-- Employee: Whalen
```

Transaction Processing and Control

A **transaction** is a sequence of one or more SQL statements that Oracle Database treats as a unit: either all of the statements are performed, or none of them are. For more information about transactions, see *Oracle Database Concepts*.

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. For more information about transaction processing, see *Oracle Database Concepts*.

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. For more information about the Oracle Database locking mechanism, see *Oracle Database Concepts*.

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 manually override the Oracle Database default locking mechanisms. For more information about manual data locks, see *Oracle Database Concepts*.

Topics

- COMMIT Statement
- ROLLBACK Statement
- SAVEPOINT Statement
- Implicit Rollbacks
- SET TRANSACTION Statement
- Overriding Default Locking

COMMIT Statement

The `COMMIT` statement ends the current transaction, making its changes permanent and visible to other users.

> **Note:** A transaction can span multiple blocks, and a block can contain multiple transactions.

The `WRITE` clause of the `COMMIT` statement specifies the priority with which Oracle Database writes to the redo log the information that the commit operation generates.

In Example 6–35, 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 6–35 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 a SQL statement fails or PL/SQL raises an exception, a rollback lets you take corrective action and perhaps start over.

Example 6–36 inserts information about an employee into three different tables. If an INSERT statement tries to store a duplicate employee number, PL/SQL raises the predefined exception DUP_VAL_ON_INDEX. To ensure that changes to all three tables are undone, the exception handler runs a ROLLBACK.

Example 6–36 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 6–37 marks a savepoint before doing an insert. If the INSERT statement tries to store a duplicate value in the employee_id column, PL/SQL raises the predefined exception DUP_VAL_ON_INDEX and the transaction rolls back to the savepoint, undoing only the INSERT statement.

Example 6–37 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 6–38 Reusing 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 6–39 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 6–39 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 appear only 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 statement) 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 9–11.

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 6–40, a FOR UPDATE cursor appears in the CURRENT OF clause of an UPDATE statement.

Example 6–40 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 6–41, 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 6–41 SELECT FOR UPDATE Statement for 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 6–42 shows (the result is the same if you substitute ROLLBACK for COMMIT).

Example 6–42 FETCH with FOR UPDATE Cursor After COMMIT Statement

```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

DECLARE
  CURSOR c1 IS
    SELECT * FROM emp
    FOR UPDATE OF salary
    ORDER BY employee_id;

  emp_rec  emp%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)
    );

    UPDATE emp
    SET salary = salary * 1.05
    WHERE employee_id = 105;

    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 11
```

The workaround is to simulate the CURRENT OF clause with the ROWID pseudocolumn (described in *Oracle Database SQL Language Reference*). 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 6–43. (To print the value of a UROWID variable, convert it to VARCHAR2, using the ROWIDTOCHAR function described in *Oracle Database SQL Language Reference*.)

> **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 6–43 Simulating CURRENT OF Clause with ROWID Pseudocolumn

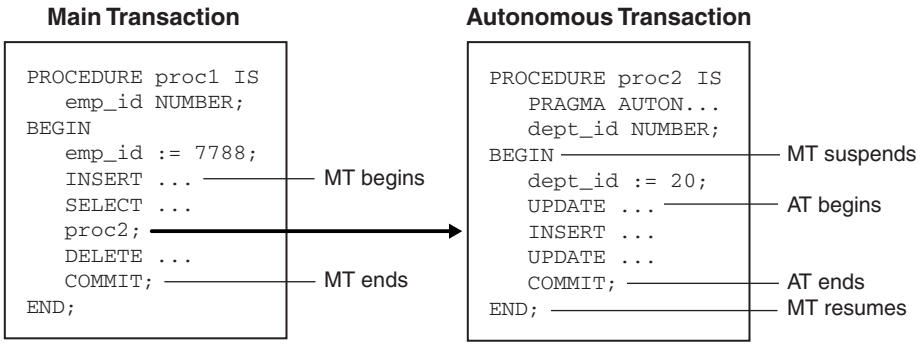
```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

DECLARE
  CURSOR c1 IS
    SELECT last_name, job_id, rowid
    FROM emp;  -- 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 emp
    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.

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

After starting, 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. An invoking 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 or ADT, but you can apply it to each subprogram in a package or each method of an ADT.

Example 6–44 marks a package function as autonomous.

Example 6–44 Declaring Autonomous Function in 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 6–45 marks a standalone stored subprogram as autonomous.

Example 6–45 Declaring 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 6–46 marks a schema-level PL/SQL block as autonomous. (A nested PL/SQL block cannot be autonomous.)

Example 6–46 Declaring Autonomous PL/SQL Block

```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

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.

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.

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, PL/SQL raises an exception 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

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.

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 trigger must be autonomous to run TCL or DDL statements. To run DDL statements, the trigger must use native dynamic SQL.

See Also:

- Chapter 9, "PL/SQL Triggers," for general information about triggers
- "Description of Static SQL" on page 6-1 for general information about TCL statements
- *Oracle Database SQL Language Reference* for information about DDL statements
- "Native Dynamic SQL" on page 7-2 for information about native dynamic SQL

One use of triggers is to log events transparently—for example, to log all inserts into a table, even those that roll back. In Example 6–47, whenever a row is inserted into the `EMPLOYEES` table, a trigger inserts the same row into a log table. Because the trigger is autonomous, it can commit changes to the log table regardless of whether they are committed to the main table.

Example 6–47 Autonomous Trigger Logs INSERT Statements

```
DROP TABLE emp;
CREATE TABLE emp AS SELECT * FROM employees;

-- Log table:

DROP TABLE log;
CREATE TABLE log (
  log_id   NUMBER(6),
  up_date  DATE,
  new_sal  NUMBER(8,2),
  old_sal  NUMBER(8,2)
);

-- Autonomous trigger on emp table:

CREATE OR REPLACE TRIGGER log_sal
  BEFORE UPDATE OF salary ON emp FOR EACH ROW
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO log (
    log_id,
    up_date,
    new_sal,
    old_sal
  )
  VALUES (
    :old.employee_id,
    SYSDATE,
    :new.salary,
    :old.salary
  );
  COMMIT;
END;
/
UPDATE emp
SET salary = salary * 1.05
WHERE employee_id = 115;
```

```
COMMIT;

UPDATE emp
SET salary = salary * 1.05
WHERE employee_id = 116;

ROLLBACK;

-- Show that both committed and rolled-back updates
-- add rows to log table

SELECT * FROM log
WHERE log_id = 115 OR log_id = 116;
```

Result:

```
    LOG_ID UP_DATE      NEW_SAL    OLD_SAL
---------- --------- ---------- ----------
       115 28-APR-10    3417.75       3255
       116 28-APR-10    3197.25       3045

2 rows selected.
```

In Example 6–48, an autonomous trigger uses native dynamic SQL (an `EXECUTE IMMEDIATE` statement) to drop a temporary table after a row is inserted into the table `log`.

Example 6–48 Autonomous Trigger Uses Native Dynamic SQL for DDL

```
DROP TABLE temp;
CREATE TABLE temp (
  temp_id NUMBER(6),
  up_date DATE
);

CREATE OR REPLACE TRIGGER drop_temp_table
  AFTER INSERT ON log
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE temp';
  COMMIT;
END;
/
-- Show how trigger works
SELECT * FROM temp;
```

Result:

```
no rows selected

INSERT INTO log (log_id, up_date, new_sal, old_sal)
VALUES (999, SYSDATE, 5000, 4500);

1 row created.

SELECT * FROM temp;
```

Result:

```
SELECT * FROM temp
```

```
              *
ERROR at line 1:
ORA-00942: table or view does not exist
```

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 8-31). 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 package function `log_msg` in Example 6–49 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 6–49 Invoking 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 package 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 database definition language (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 6-5.

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 8-9.)

  – 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 13-52.

Example 7–1 Invoking Subprogram from 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 Represents 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 13-103.

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 13-71.

3. Use the CLOSE statement to close the cursor variable.

   For syntax details, see "CLOSE Statement" on page 13-23.

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 weak cursor variable, 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 strong or weak) 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.

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:

```
Block1: 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 ORDER BY user_name;
```

Result:

```
USER_NAME SERVICE_TYPE VALUE                          DATE_CREATED
--------- ------------ ------------------------------ ------------
Andy      Waiter       Serve dinner at Cafe Pete      28-APR-10
Chuck     Merger       Buy company XYZ                28-APR-10
```

Example of statement modification:

```
BEGIN
  p('Anybody', 'Anything'');
  DELETE FROM secret_records WHERE service_type=INITCAP(''Merger');
END;
/
```

Result:

```
Block1: 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      28-APR-2010
Chuck     Merger       Buy company XYZ                28-APR-2010
```

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>'29-MAR-2010'
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 '2010-03-29'
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

A PL/SQL **subprogram** is a named PL/SQL block that can be invoked repeatedly. If the subprogram has parameters, their values can differ for each invocation.

A subprogram is either a procedure or a function. Typically, you use a procedure to perform an action and a function to compute and return a value.

Topics

- Reasons to Use Subprograms
- Nested, Package, and Standalone Stored Subprograms
- Subprogram Invocations
- Subprogram Parts
- Forward Declaration
- Subprogram Parameters
- Subprogram Invocation Resolution
- Overloaded Subprograms
- Recursive Subprograms
- Subprogram Side Effects
- PL/SQL Function Result Cache
- PL/SQL Functions that SQL Statements Can Invoke
- Invoker's Rights and Definer's Rights (AUTHID Property)
- External Subprograms

Reasons to Use Subprograms

Subprograms support the development and maintenance of reliable, reusable code with the following features:

- **Modularity**

  Subprograms let you break a program into manageable, well-defined modules.

- **Easier Application Design**

  When designing an application, you can defer the implementation details of the subprograms until you have tested the main program, and then refine them one step at a time. (To define a subprogram without implementation details, use the `NULL` statement, as in Example 4–34.)

- **Maintainability**

  You can change the implementation details of a subprogram without changing its invokers.

- **Packageability**

  Subprograms can be grouped into packages, whose advantages are explained in "Reasons to Use Packages" on page 10-2.

- **Reusability**

  Any number of applications, in many different environments, can use the same package subprogram or standalone stored subprogram.

- **Better Performance**

  Each subprogram is compiled and stored in executable form, which can be invoked repeatedly. Because stored subprograms run in the database server, a single invocation 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 invocation overhead.

Subprograms are an important component of other maintainability features, such as packages (explained in Chapter 10, "PL/SQL Packages") and Abstract Data Types (explained in "Abstract Data Types" on page 1-8).

Nested, Package, and Standalone Stored Subprograms

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

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 and then define it later in the same block (see "Forward Declaration" on page 8-8). A nested subprogram is stored in the database only if it is nested in a standalone or package subprogram.

A subprogram created inside a package is a **package subprogram**. You declare it in the package specification and define it in the package body. It is stored in the database until you drop the package. (Packages are described in Chapter 10, "PL/SQL Packages.")

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. (These statements are described in Chapter 14, "SQL Statements for Stored PL/SQL Units.")

A **stored subprogram** is either a package subprogram or a standalone stored subprogram.

When you create a standalone stored subprogram or package, you can specify the `AUTHID` property, which affects the name resolution and privilege checking of SQL statements that the subprogram issues at run time. For more information, see "Invoker's Rights and Definer's Rights (AUTHID Property)" on page 8-43.

Subprogram Invocations

A subprogram invocation has this form:

```
subprogram_name [ ( [ parameter [, parameter]... ] ) ]
```

If the subprogram has no parameters, or specifies a default value for every parameter, you can either omit the parameter list or specify an empty parameter list.

A procedure invocation is a PL/SQL statement. For example:

```
raise_salary(employee_id, amount);
```

A function invocation is an expression. For example:

```
new_salary := get_salary(employee_id);
IF salary_ok(new_salary, new_title) THEN ...
```

> **See Also:** "Subprogram Parameters" on page 8-9 for more information about specifying parameters in subprogram invocations

Subprogram Parts

A subprogram begins with a **subprogram heading**, which specifies its name and (optionally) its parameter list.

Like an anonymous block, a subprogram has these parts:

- **Declarative part (optional)**

  This part declares and defines local types, cursors, constants, variables, exceptions, and nested subprograms. These items cease to exist when the subprogram completes execution.

  This part can also specify pragmas (described in "Pragmas" on page 2-41).

  > **Note:** The declarative part of a subprogram does not begin with the keyword `DECLARE`, as the declarative part of an anonymous block does.

- **Executable part (required)**

  This part contains one or more statements that assign values, control execution, and manipulate data. (Early in the application design process, this part might contain only a `NULL` statement, as in Example 4–34.)

- **Exception-handling part (optional)**

  This part contains code that handles run-time errors.

In Example 8–1, an anonymous block simultaneously declares and defines a procedure and invokes it three times. The third invocation raises the exception that the exception-handling part of the procedure handles.

Example 8–1 Declaring, Defining, and Invoking a Simple PL/SQL Procedure

```
DECLARE
  first_name employees.first_name%TYPE;
  last_name  employees.last_name%TYPE;
  email      employees.email%TYPE;
  employer   VARCHAR2(8) := 'AcmeCorp';

  -- Declare and define procedure

  PROCEDURE create_email (  -- Subprogram heading begins
    name1   VARCHAR2,
```

```
    name2   VARCHAR2,
    company VARCHAR2
  )                          -- Subprogram heading ends
  IS
                             -- Declarative part begins
    error_message VARCHAR2(30) := 'Email address is too long.';
  BEGIN                      -- Executable part begins
    email := name1 || '.' || name2 || '@' || company;
  EXCEPTION                        -- Exception-handling part begins
    WHEN VALUE_ERROR THEN
      DBMS_OUTPUT.PUT_LINE(error_message);
  END create_email;

BEGIN
  first_name := 'John';
  last_name  := 'Doe';

  create_email(first_name, last_name, employer);  -- invocation
  DBMS_OUTPUT.PUT_LINE ('With first name first, email is: ' || email);

  create_email(last_name, first_name, employer);  -- invocation
  DBMS_OUTPUT.PUT_LINE ('With last name first, email is: ' || email);

  first_name := 'Elizabeth';
  last_name  := 'MacDonald';
  create_email(first_name, last_name, employer);  -- invocation
END;
/
```

Result:

```
With first name first, email is: John.Doe@AcmeCorp
With last name first, email is: Doe.John@AcmeCorp
Email address is too long.
```

Topics

- Additional Parts for Functions
- RETURN Statement

See Also:

- "Procedure Declaration and Definition" on page 13-108 for the syntax of procedure declarations and definitions
- "Subprogram Parameters" on page 8-9 for more information about subprogram parameters

Additional Parts for Functions

A function has the same structure as a procedure, except that:

- A function heading must include a **`RETURN` clause**, which specifies the data type of the value that the function returns. (A procedure heading cannot have a `RETURN` clause.)
- In the executable part of a function, every execution path must lead to a **`RETURN` statement**. Otherwise, the PL/SQL compiler issues a compile-time warning. (In a procedure, the `RETURN` statement is optional and not recommended. For details, see "RETURN Statement" on page 8-5.)

- Only a function heading can include these options:

Option	Description
DETERMINISTIC option	Helps the optimizer avoid redundant function invocations.
PARALLEL_ENABLE option	Enables the function for parallel execution, making it safe for use in slave sessions of parallel DML evaluations.
PIPELINED option	Makes a table function pipelined, for use as a row source.
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:

- "Function Declaration and Definition" on page 13-83 for the syntax of function declarations and definitions, including descriptions of the items in the preceding table
- "PL/SQL Function Result Cache" on page 8-31 for more information about the RESULT_CACHE option and clause

In Example 8–2, an anonymous block simultaneously declares and defines a function and invokes it.

Example 8–2 Declaring, Defining, and Invoking a Simple PL/SQL Function

```
DECLARE
  -- Declare and define function

  FUNCTION square (original NUMBER)   -- parameter list
    RETURN NUMBER                     -- RETURN clause
  AS
                                      -- Declarative part begins
    original_squared NUMBER;
  BEGIN                               -- Executable part begins
    original_squared := original * original;
    RETURN original_squared;          -- RETURN statement
  END;
BEGIN
  DBMS_OUTPUT.PUT_LINE(square(100));  -- invocation
END;
/
```

Result:

```
10000
```

RETURN Statement

The RETURN statement immediately ends the execution of the subprogram or anonymous block that contains it. A subprogram or anonymous block can contain multiple RETURN statements.

Topics

- RETURN Statement in Function

- RETURN Statement in Procedure
- RETURN Statement in Anonymous Block

> **See Also:** "RETURN Statement" on page 13-116 for the syntax of the RETURN statement

RETURN Statement in Function

In a function, every execution path must lead to a RETURN statement and every RETURN statement must specify an expression. The RETURN statement assigns the value of the expression to the function identifier and returns control to the invoker, where execution resumes immediately after the invocation.

> **Note:** In a pipelined table function, a RETURN statement need not specify an expression. For information about the parts of a pipelined table function, see "Creating Pipelined Table Functions" on page 12-40.

In Example 8–3, the anonymous block invokes the same function twice. The first time, the RETURN statement returns control to the inside of the invoking statement. The second time, the RETURN statement returns control to the statement immediately after the invoking statement.

Example 8–3 Execution Resumes After RETURN Statement in Function

```
DECLARE
  x INTEGER;

  FUNCTION f (n INTEGER)
  RETURN INTEGER
  IS
  BEGIN
    RETURN (n*n);
  END;

BEGIN
  DBMS_OUTPUT.PUT_LINE (
    'f returns ' || f(2) || '. Execution returns here (1).'
  );

  x := f(2);
  DBMS_OUTPUT.PUT_LINE('Execution returns here (2).');
END;
/
```

Result:

```
f returns 4. Execution returns here (1).
Execution returns here (2).
```

In Example 8–4, the function has multiple RETURN statements, but if the parameter is not 0 or 1, then no execution path leads to a RETURN statement. The function compiles with warning PLW-05005: subprogram F returns without value at line 10.

Example 8–4 Function Where Not Every Execution Path Leads to RETURN Statement

```
CREATE OR REPLACE FUNCTION f (n INTEGER)
  RETURN INTEGER
IS
```

```
BEGIN
  IF n = 0 THEN
    RETURN 1;
  ELSIF n = 1 THEN
    RETURN n;
  END IF;
END;
/
```

Example 8–5 is like Example 8–4, except for the addition of the ELSE clause. Every execution path leads to a RETURN statement, and the function compiles without warning PLW-05005.

Example 8–5 Function Where Every Execution Path Leads to RETURN Statement

```
CREATE OR REPLACE FUNCTION f (n INTEGER)
  RETURN INTEGER
IS
BEGIN
  IF n = 0 THEN
    RETURN 1;
  ELSIF n = 1 THEN
    RETURN n;
  ELSE
    RETURN n*n;
  END IF;
END;
/
BEGIN
  FOR i IN 0 .. 3 LOOP
    DBMS_OUTPUT.PUT_LINE('f(' || i || ') = ' || f(i));
  END LOOP;
END;
/
```

Result:

```
f(0) = 1
f(1) = 1
f(2) = 4
f(3) = 9
```

RETURN Statement in Procedure

In a procedure, the RETURN statement returns control to the invoker, where execution resumes immediately after the invocation. The RETURN statement cannot specify an expression.

In Example 8–6, the RETURN statement returns control to the statement immediately after the invoking statement.

Example 8–6 Execution Resumes After RETURN Statement in Procedure

```
DECLARE
  PROCEDURE p IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Inside p');
    RETURN;
    DBMS_OUTPUT.PUT_LINE('Unreachable statement.');
  END;
BEGIN
```

```
  p;
  DBMS_OUTPUT.PUT_LINE('Control returns here.');
END;
/
```

Result:

```
Inside p
Control returns here.
```

RETURN Statement in Anonymous Block

In an anonymous block, the RETURN statement exits its own block and all enclosing blocks. The RETURN statement cannot specify an expression.

In Example 8–7, the RETURN statement exits both the inner and outer block.

Example 8–7 Execution Resumes After RETURN Statement in Anonymous Block

```
BEGIN
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Inside inner block.');
    RETURN;
    DBMS_OUTPUT.PUT_LINE('Unreachable statement.');
  END;
  DBMS_OUTPUT.PUT_LINE('Inside outer block. Unreachable statement.');
END;
/
```

Result:

```
Inside inner block.
```

Forward Declaration

If nested subprograms in the same PL/SQL block invoke each other, then one requires a forward declaration, because a subprogram must be declared before it can be invoked.

A **forward declaration** declares a nested subprogram but does not define it. You must define it later in the same block. The forward declaration and the definition must have the same subprogram heading.

In Example 8–8, an anonymous block creates two procedures that invoke each other.

Example 8–8 Nested Subprograms Invoke Each Other

```
DECLARE
  -- Declare proc1 (forward declaration):
  PROCEDURE proc1(number1 NUMBER);

  -- Declare and define proc2:
  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

If a subprogram has parameters, their values can differ for each invocation.

Topics

- Formal and Actual Subprogram Parameters
- Subprogram Parameter Passing Methods
- Subprogram Parameter Modes
- Subprogram Parameter Aliasing
- Default Values for IN Subprogram Parameters
- Positional, Named, and Mixed Notation for Actual Parameters

Formal and Actual Subprogram Parameters

If you want a subprogram to have parameters, declare **formal parameters** in the subprogram heading. In each formal parameter declaration, specify the name and data type of the parameter, and (optionally) its mode and default value. In the execution part of the subprogram, reference the formal parameters by their names.

When invoking the subprogram, specify the **actual parameters** whose values are to be assigned to the formal parameters. Corresponding actual and formal parameters must have compatible data types.

> **Tip:** To avoid confusion, use different names for formal and actual parameters.

In Example 8–9, the procedure has formal parameters `emp_id` and `amount`. In the first procedure invocation, the corresponding actual parameters are `emp_num` and `bonus`, whose value are 120 and 100, respectively. In the second procedure invocation, the actual parameters are `emp_num` and `merit + bonus`, whose value are 120 and 150, respectively.

Example 8–9 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  -- reference to formal parameter
    WHERE employee_id = emp_id;   -- reference to formal parameter
  END raise_salary;
```

```
BEGIN
  raise_salary(emp_num, bonus);          -- actual parameters

  /* raise_salary runs this statement:
       UPDATE employees
       SET salary = salary + 100
       WHERE employee_id = 120;       */

  raise_salary(emp_num, merit + bonus);  -- actual parameters

  /* raise_salary runs this statement:
       UPDATE employees
       SET salary = salary + 150
       WHERE employee_id = 120;       */
END;
/
```

If the data type of a formal parameter is a constrained subtype, then:

- If the subtype has the NOT NULL constraint, then the actual parameter inherits it.
- If the subtype has the base type VARCHAR2, then the actual parameter does not inherit the size of the subtype.
- If the subtype has a numeric base type, then the actual parameter inherits the range of the subtype, but not the size, scale, or precision.

Example 8–10 shows that a subprogram parameter inherits the NOT NULL constraint but not the size of a VARCHAR2 subtype.

Example 8–10 Parameter Inherits Only NOT NULL from Subtype

```
DECLARE
  SUBTYPE License IS VARCHAR2(7) NOT NULL;
  n  License := 'DLLLDDD';

  PROCEDURE p (x License) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE(x);
  END;

BEGIN
  p('1ABC123456789');  -- Succeeds; size is not inherited
  p(NULL);             -- Raises error; NOT NULL is inherited
END;
/
```

Result:

```
  p(NULL);             -- Raises error; NOT NULL is inherited
    *
ERROR at line 12:
ORA-06550: line 12, column 5:
PLS-00567: cannot pass NULL to a NOT NULL constrained formal parameter
ORA-06550: line 12, column 3:
PL/SQL: Statement ignored
```

See Also:

- "Formal Parameter Declaration" on page 13-80 for the syntax and semantics of a formal parameter declaration
- "function_call ::=" on page 13-65 and "function_call" on page 13-68 for the syntax and semantics of a function invocation
- "procedure_call ::=" on page 13-13 and "procedure_call" on page 13-17 for the syntax and semantics of a procedure invocation
- "Constrained Subtypes" on page 3-12 for general information about constrained subtypes

Subprogram Parameter Passing Methods

The PL/SQL compiler has two ways of passing an actual parameter to a subprogram:

- **By reference**

  The compiler passes the subprogram a pointer to the actual parameter. The actual and formal parameters refer to the same memory location.

- **By value**

  The compiler assigns the value of the actual parameter to the corresponding formal parameter. The actual and formal parameters refer to different memory locations.

  If necessary, the compiler implicitly converts the data type of the actual parameter to the data type of the formal parameter. For information about implicit data conversion, see *Oracle Database SQL Language Reference*.

  Tip: Avoid implicit data conversion (for the reasons in *Oracle Database SQL Language Reference*), in either of these ways:

  - Declare the variables that you intend to use as actual parameters with the same data types as their corresponding formal parameters (as in the declaration of variable `x` in Example 8–11).
  - Explicitly convert actual parameters to the data types of their corresponding formal parameters, using the SQL conversion functions described in *Oracle Database SQL Language Reference* (as in the third invocation of the procedure in Example 8–11).

In Example 8–11, the procedure `p` has one parameter, `n`, which is passed by value. The anonymous block invokes `p` three times, avoiding implicit conversion twice.

Example 8–11 Avoiding Implicit Conversion of Actual Parameters

```
CREATE OR REPLACE PROCEDURE p (
  n NUMBER
) IS
BEGIN
  NULL;
END;
/
DECLARE
  x NUMBER      :=  1;
  y VARCHAR2(1) := '1';
BEGIN
  p(x);             -- No conversion needed
```

```
  p(y);             -- z implicitly converted from VARCHAR2 to NUMBER
  p(TO_NUMBER(y));  -- z explicitly converted from VARCHAR2 to NUMBER
END;
/
```

The method by which the compiler passes a specific actual parameter depends on its mode, as explained in "Subprogram Parameter Modes" on page 8-12.

Subprogram Parameter Modes

The **mode** of a formal parameter determines its behavior.

Table 8–1 summarizes and compares the characteristics of the subprogram parameter modes.

Table 8–1 PL/SQL Subprogram Parameter Modes

IN	OUT	IN OUT
Default mode	Must be specified.	Must be specified.
Passes a value to the subprogram.	Returns a value to the invoker.	Passes an initial value to the subprogram and returns an updated value to the invoker.
Formal parameter acts like a constant: When the subprogram begins, its value is that of either its actual parameter or default value, and the subprogram cannot change this value.	Formal parameter acts like an uninitialized variable: When the subprogram begins, its value is `NULL`, regardless of the value of its actual parameter. Oracle recommends that the subprogram assign it a value.	Formal parameter acts like an initialized variable: When the subprogram begins, its value is that of its actual parameter. Oracle recommends that the subprogram update its value.
Actual parameter can be a constant, initialized variable, literal, or expression.	Actual parameter must be a variable whose data type is not defined as `NOT NULL`.	Actual parameter must be a variable (typically, it is a string buffer or numeric accumulator).
Actual parameter is passed by reference.	By default, actual parameter is passed by value; if you specify `NOCOPY`, it might be passed by reference.	By default, actual parameter is passed by value (in both directions); if you specify `NOCOPY`, it might be passed by reference.

Tip: Do not use `OUT` and `IN OUT` for function parameters. Ideally, a function takes zero or more parameters and returns a single value. A function with `IN OUT` parameters returns multiple values and has side effects.

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.

When an OUT or IN OUT parameter is passed by value (the default):

- If the subprogram is exited successfully, then the value of the formal parameter is assigned to the actual parameter (see Example 8–12).
- If the subprogram ends with an unhandled exception, then the value of the formal parameter is not assigned to the actual parameter, and the actual parameter retains the value that it had before the subprogram invocation (see Example 8–13).

When an OUT or IN OUT parameter is passed by reference, the actual and formal parameters refer to the same memory location. Therefore, if the subprogram changes the value of the formal parameter, the change shows immediately in the actual parameter (see "Subprogram Parameter Aliasing with Parameters Passed by Reference" on page 8-16).

In Example 8–12, the procedure p has two IN parameters, one OUT parameter, and one IN OUT parameter. The OUT and IN OUT parameters are passed by value (the default). The anonymous block invokes p twice, with different actual parameters. Before each invocation, the anonymous block prints the values of the actual parameters. The procedure p prints the initial values of its formal parameters. After each invocation, the anonymous block prints the values of the actual parameters again. (Both the anonymous block and p invoke the procedure print, which is created first.)

Example 8–12 Parameter Values Before, During, and After Procedure Invocation

```
CREATE OR REPLACE PROCEDURE print (x PLS_INTEGER) IS
BEGIN
  IF x IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE(x);
  ELSE
    DBMS_OUTPUT.PUT_LINE('NULL');
  END IF;
END print;
/
CREATE OR REPLACE PROCEDURE p (
  a        PLS_INTEGER,  -- IN by default
  b     IN PLS_INTEGER,
  c    OUT PLS_INTEGER,
  d IN OUT BINARY_FLOAT
) IS
BEGIN
  -- Print values of parameters:

  DBMS_OUTPUT.PUT_LINE('Inside procedure p:');
  DBMS_OUTPUT.PUT('IN a = '); print(a);
  DBMS_OUTPUT.PUT('IN b = '); print(b);
  DBMS_OUTPUT.PUT('OUT c = '); print(c);
  DBMS_OUTPUT.PUT_LINE('IN OUT d = ' || TO_CHAR(d));

  -- Can reference IN parameters a and b,
  -- but cannot assign values to them.

  c := a+10;  -- Assign value to OUT parameter
  d := 10/b;  -- Assign value to IN OUT parameter
END;
/
DECLARE
  aa  CONSTANT PLS_INTEGER := 1;
  bb  PLS_INTEGER  := 2;
  cc  PLS_INTEGER  := 3;
  dd  BINARY_FLOAT := 4;
```

```
  ee  PLS_INTEGER;
  ff  BINARY_FLOAT := 5;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');
  DBMS_OUTPUT.PUT('aa = '); print(aa);
  DBMS_OUTPUT.PUT('bb = '); print(bb);
  DBMS_OUTPUT.PUT('cc = '); print(cc);
  DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));

  p (aa, -- constant
     bb, -- initialized variable
     cc, -- initialized variable
     dd  -- initialized variable
  );

  DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');
  DBMS_OUTPUT.PUT('aa = '); print(aa);
  DBMS_OUTPUT.PUT('bb = '); print(bb);
  DBMS_OUTPUT.PUT('cc = '); print(cc);
  DBMS_OUTPUT.PUT_LINE('dd = ' || TO_CHAR(dd));

  DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');
  DBMS_OUTPUT.PUT('ee = '); print(ee);
  DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));

  p (1,        -- literal
     (bb+3)*4, -- expression
     ee,       -- uninitialized variable
     ff        -- initialized variable
   );

  DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');
  DBMS_OUTPUT.PUT('ee = '); print(ee);
  DBMS_OUTPUT.PUT_LINE('ff = ' || TO_CHAR(ff));
END;
/
```

Result:

```
Before invoking procedure p:
aa = 1
bb = 2
cc = 3
dd = 4.0E+000
Inside procedure p:
IN a = 1
IN b = 2
OUT c = NULL
IN OUT d = 4.0E+000
After invoking procedure p:
aa = 1
bb = 2
cc = 11
dd = 5.0E+000
Before invoking procedure p:
ee = NULL
ff = 5.0E+000
Inside procedure p:
IN a = 1
IN b = 20
OUT c = NULL
```

```
IN OUT d = 5.0E+000
After invoking procedure p:
ee = 11
ff = 5.0E-001

PL/SQL procedure successfully completed.
```

In Example 8–13, the anonymous block invokes procedure p (from Example 8–12) with an actual parameter that causes p to raise the predefined exception ZERO_DIVIDE, which p does not handle. The exception propagates to the anonymous block, which handles ZERO_DIVIDE and shows that the actual parameters for the IN and IN OUT parameters of p have retained the values that they had before the invocation. (Exception propagation is explained in "Exception Propagation" on page 11-17.)

Example 8–13 OUT and IN OUT Parameter Values After Unhandled Exception

```
DECLARE
  j  PLS_INTEGER  := 10;
  k  BINARY_FLOAT := 15;
BEGIN
  DBMS_OUTPUT.PUT_LINE('Before invoking procedure p:');
  DBMS_OUTPUT.PUT('j = '); print(j);
  DBMS_OUTPUT.PUT_LINE('k = ' || TO_CHAR(k));

  p(4, 0, j, k);  -- causes p to exit with exception ZERO_DIVIDE

EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('After invoking procedure p:');
    DBMS_OUTPUT.PUT('j = '); print(j);
    DBMS_OUTPUT.PUT_LINE('k = ' || TO_CHAR(k));
END;
/
```

Result:

```
Before invoking procedure p:
j = 10
k = 1.5E+001
Inside procedure p:
IN a = 4
IN b = 0
OUT c = NULL
d = 1.5E+001
After invoking procedure p:
j = 10
k = 1.5E+001

PL/SQL procedure successfully completed.
```

Subprogram Parameter Aliasing

Aliasing is having two different names for the same memory location. Subprogram parameter aliasing always occurs when the compiler passes an actual parameter by reference, and can also occur when a subprogram has cursor variable parameters.

Topics

- Subprogram Parameter Aliasing with Parameters Passed by Reference

- Subprogram Parameter Aliasing with Cursor Variable Parameters

Subprogram Parameter Aliasing with Parameters Passed by Reference

When the compiler passes an actual parameter by reference, the actual and formal parameters refer to the same memory location. Therefore, if the subprogram changes the value of the formal parameter, the change shows immediately in the actual parameter.

The compiler always passes `IN` parameters by reference, but the resulting aliasing cannot cause problems, because subprograms cannot assign values to `IN` parameters.

The compiler *might* pass an `OUT` or `IN OUT` parameter by reference, if you specify `NOCOPY` for that parameter. `NOCOPY` is only a hint—each time the subprogram is invoked, the compiler decides, silently, whether to obey or ignore `NOCOPY`. Therefore, aliasing can occur for one invocation but not another, making subprogram results indeterminate. For example:

- If the actual parameter is a global variable, then an assignment to the formal parameter *might* show in the global parameter (see Example 8–14).
- If the same variable is the actual parameter for two formal parameters, then an assignment to either formal parameter *might* show immediately in both formal parameters (see Example 8–15).
- If the actual parameter is a package variable, then an assignment to either the formal parameter or the package variable *might* show immediately in both the formal parameter and the package variable.
- If the subprogram is exited with an unhandled exception, then an assignment to the formal parameter *might* show in the actual parameter.

> **See Also:** "NOCOPY" on page 13-81 for the cases in which the compiler always ignores `NOCOPY`

In Example 8–14, the procedure has an `IN OUT NOCOPY` formal parameter, to which it assigns the value `'aardvark'`. The anonymous block assigns the value `'aardwolf'` to a global variable and then passes the global variable to the procedure. If the compiler obeys the `NOCOPY` hint, then the final value of the global variable is `'aardvark'`. If the compiler ignores the `NOCOPY` hint, then the final value of the global variable is `'aardwolf'`.

Example 8–14 Aliasing from Global Variable as Actual Parameter

```
DECLARE
  TYPE Definition IS RECORD (
    word     VARCHAR2(20),
    meaning  VARCHAR2(200)
  );

  TYPE Dictionary IS VARRAY(2000) OF Definition;

  lexicon  Dictionary := Dictionary();  -- global variable

  PROCEDURE add_entry (
    word_list IN OUT NOCOPY Dictionary  -- formal NOCOPY parameter
  ) IS
  BEGIN
    word_list(1).word := 'aardvark';
  END;
```

```
BEGIN
  lexicon.EXTEND;
  lexicon(1).word := 'aardwolf';
  add_entry(lexicon);  -- global variable is actual parameter
  DBMS_OUTPUT.PUT_LINE(lexicon(1).word);
END;
/
```

Result:

```
aardvark
```

In Example 8–15, the procedure has an IN parameter, an IN OUT parameter, and an IN OUT NOCOPY parameter. The anonymous block invokes the procedure, using the same actual parameter, a global variable, for all three formal parameters. The procedure changes the value of the IN OUT parameter before it changes the value of the IN OUT NOCOPY parameter. However, if the compiler obeys the NOCOPY hint, then the latter change shows in the actual parameter immediately. The former change shows in the actual parameter after the procedure is exited successfully and control returns to the anonymous block.

Example 8–15 Aliasing from Same Actual Parameter for Multiple Formal Parameters

```
DECLARE
  n NUMBER := 10;

  PROCEDURE p (
    n1 IN NUMBER,
    n2 IN OUT NUMBER,
    n3 IN OUT NOCOPY NUMBER
  ) IS
  BEGIN
    n2 := 20;  -- actual parameter is 20 only after procedure succeeds
    DBMS_OUTPUT.put_line(n1);  -- actual parameter value is still 10
    n3 := 30;  -- might change actual parameter immediately
    DBMS_OUTPUT.put_line(n1);  -- actual parameter value is either 10 or 30
  END;

BEGIN
  p(n, n, n);
  DBMS_OUTPUT.put_line(n);
END;
/
```

Result if the compiler obeys the NOCOPY hint:

```
10
30
20
```

Result if the compiler ignores the NOCOPY hint:

```
10
10
30
```

Subprogram Parameter Aliasing with Cursor Variable Parameters

Cursor variable parameters are pointers. Therefore, if a subprogram assigns one cursor variable parameter to another, they refer to the same memory location. This aliasing can have unintended results.

In Example 8–16, the procedure has two cursor variable parameters, `emp_cv1` and `emp_cv2`. The procedure opens `emp_cv1` and assigns its value (which is a pointer) to `emp_cv2`. Now `emp_cv1` and `emp_cv2` refer to the same memory location. When the procedure closes `emp_cv1`, it also closes `emp_cv2`. Therefore, when the procedure tries to fetch from `emp_cv2`, PL/SQL raises an exception.

Example 8–16 Aliasing from Cursor Variable Subprogram Parameters

```
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;  -- now both variables refer to same location
    FETCH emp_cv1 INTO emp_rec;  -- fetches first row of employees
    FETCH emp_cv1 INTO emp_rec;  -- fetches second row of employees
    FETCH emp_cv2 INTO emp_rec;  -- fetches third row of employees
    CLOSE emp_cv1;  -- closes both variables
    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 19
ORA-06512: at line 22
```

Default Values for IN Subprogram Parameters

When you declare a formal `IN` parameter, you can specify a default value for it. A formal parameter with a default value is called an **optional parameter**, because its corresponding actual parameter is optional in a subprogram invocation. If the actual parameter is omitted, then the invocation assigns the default value to the formal parameter. A formal parameter with no default value is called a **required parameter**, because its corresponding actual parameter is required in a subprogram invocation.

Omitting an actual parameter does not make the value of the corresponding formal parameter `NULL`. To make the value of a formal parameter `NULL`, specify `NULL` as either the default value or the actual parameter.

In Example 8–17, the procedure has one required parameter and two optional parameters.

Example 8–17 Procedure with Default Parameter Values

```
DECLARE
  PROCEDURE raise_salary (
    emp_id IN employees.employee_id%TYPE,
    amount IN employees.salary%TYPE := 100,
    extra  IN employees.salary%TYPE := 50
  ) IS
  BEGIN
    UPDATE employees
    SET salary = salary + amount + extra
    WHERE employee_id = emp_id;
  END raise_salary;

BEGIN
  raise_salary(120);       -- same as raise_salary(120, 100, 50)
  raise_salary(121, 200);  -- same as raise_salary(121, 200, 50)
END;
/
```

In Example 8–17, the procedure invocations specify the actual parameters in the same order as their corresponding formal parameters are declared—that is, the invocations use positional notation. Positional notation does not let you omit the second parameter of `raise_salary` but specify the third; to do that, you must use either named or mixed notation. For more information, see "Positional, Named, and Mixed Notation for Actual Parameters" on page 8-21.

The default value of a formal parameter can be any expression whose value can be assigned to the parameter; that is, the value and parameter must have compatible data types. If a subprogram invocation specifies an actual parameter for the formal parameter, then that invocation does not evaluate the default value.

In Example 8–18, the procedure `p` has a parameter whose default value is an invocation of the function `f`. The function `f` increments the value of a global variable. When `p` is invoked without an actual parameter, `p` invokes `f`, and `f` increments the global variable. When `p` is invoked with an actual parameter, `p` does not invoke `f`, and value of the global variable does not change.

Example 8–18 Function Provides Default Parameter Value

```
DECLARE
  global PLS_INTEGER := 0;

  FUNCTION f RETURN PLS_INTEGER IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Inside f.');
    global := global + 1;
    RETURN global * 2;
  END f;

  PROCEDURE p (
    x IN PLS_INTEGER := f()
  ) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE (
      'Inside p. ' ||
      '  global = ' || global ||
```

```
      ', x = ' || x || '.'
    );
    DBMS_OUTPUT.PUT_LINE('--------------------------------');
  END p;

  PROCEDURE pre_p IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE (
     'Before invoking p,  global = ' || global || '.'
    );
    DBMS_OUTPUT.PUT_LINE('Invoking p.');
  END pre_p;

BEGIN
  pre_p;
  p();     -- default expression is evaluated

  pre_p;
  p(100);  -- default expression is not evaluated

  pre_p;
  p();     -- default expression is evaluated
END;
/
```

Result:

```
Before invoking p,  global = 0.
Invoking p.
Inside f.
Inside p.   global = 1, x = 2.
--------------------------------
Before invoking p,  global = 1.
Invoking p.
Inside p.   global = 1, x = 100.
--------------------------------
Before invoking p,  global = 1.
Invoking p.
Inside f.
Inside p.   global = 2, x = 4.
--------------------------------
```

Example 8–19 creates a procedure with two required parameters, invokes it, and then adds a third, optional parameter. Because the third parameter is optional, the original invocation remains valid.

Example 8–19 Adding Subprogram Parameter Without Changing Existing Invocations

Create procedure:

```
CREATE OR REPLACE PROCEDURE print_name (
  first VARCHAR2,
  last VARCHAR2
) IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(first || ' ' || last);
END print_name;
/
```

Invoke procedure:

```
BEGIN
  print_name('John', 'Doe');
END;
/
```

Result:

```
John Doe
```

Add third parameter with default value:

```
CREATE OR REPLACE PROCEDURE print_name (
  first VARCHAR2,
  last VARCHAR2,
  mi   VARCHAR2 := NULL
) IS
BEGIN
  IF mi IS NULL THEN
    DBMS_OUTPUT.PUT_LINE(first || ' ' || last);
  ELSE
    DBMS_OUTPUT.PUT_LINE(first || ' ' || mi || '. ' || last);
  END IF;
END print_name;
/
```

Invoke procedure:

```
BEGIN
  print_name('John', 'Doe');          -- original invocation
  print_name('John', 'Public', 'Q');  -- new invocation
END;
/
```

Result:

```
John Doe
John Q. Public
```

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 summarizes and compares these notations.

Table 8–2 PL/SQL Actual Parameter Notations

Positional	Named	Mixed
Specify the actual parameters in the same order as the formal parameters are declared.	Specify the actual parameters in any order, using this syntax: *formal => actual* *formal* is the name of the formal parameter and *actual* is the actual parameter.	Start with positional notation, then use named notation for the remaining parameters.
You can omit trailing optional parameters.	You can omit any optional parameters.	In the positional notation, you can omit trailing optional parameters; in the named notation, you can omit any optional parameters.

Table 8–2 (Cont.) PL/SQL Actual Parameter Notations

Positional	Named	Mixed
Specifying actual parameters in the wrong order can cause problems that are hard to detect, especially if the actual parameters are literals.	There is no wrong order for specifying actual parameters.	In the positional notation, the wrong order can cause problems that are hard to detect, especially if the actual parameters are literals.
Subprogram invocations must change if the formal parameter list changes, unless the list only acquires new trailing optional parameters (as in Example 8–19).	Subprogram invocations must change only if the formal parameter list acquires new required parameters.	Changes to the formal parameter list might require changes in the positional notation.
	Recommended when you invoke a subprogram defined or maintained by someone else.	Convenient when you invoke a subprogram that has required parameters followed by optional parameters, and you must specify only a few of the optional parameters.

In Example 8–20, the procedure invocations use different notations, but are equivalent.

Example 8–20 Equivalent Invocations with Different Notations in Anonymous Block

```
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
  -- Equivalent invocations:

  raise_salary(emp_num, bonus);                      -- positional notation
  raise_salary(amount => bonus, emp_id => emp_num);  -- named notation
  raise_salary(emp_id => emp_num, amount => bonus);  -- named notation
  raise_salary(emp_num, amount => bonus);            -- mixed notation
END;
/
```

In Example 8–21, the SQL `SELECT` statements invoke the PL/SQL function `compute_bonus`, using equivalent invocations with different notations.

Example 8–21 Equivalent Invocations with Different Notations in SELECT Statements

```
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
```

Subprogram Invocation Resolution

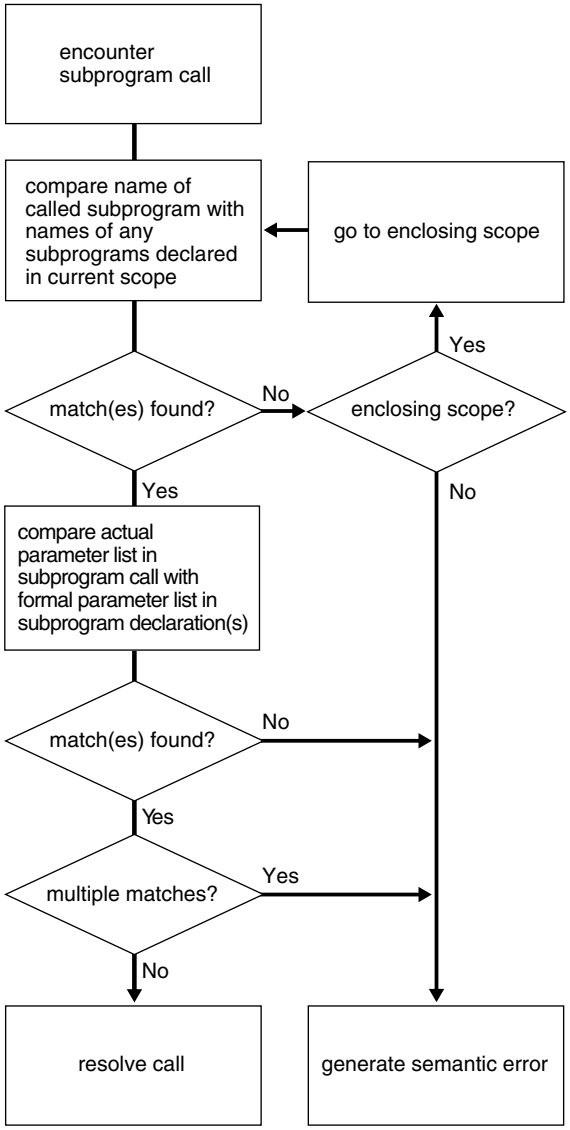
When the PL/SQL compiler encounters a subprogram invocation, it searches for a matching subprogram declaration—first in the current scope and then, if necessary, in successive enclosing scopes.

A declaration and invocation match if their subprogram names and parameter lists match. The parameter lists match if each required formal parameter in the declaration has a corresponding actual parameter in the invocation.

If the compiler finds no matching declaration for an invocation, then it generates a semantic error.

Figure 8–1 shows how the PL/SQL compiler resolves a subprogram invocation.

Figure 8–1 How PL/SQL Compiler Resolves Invocations



In Example 8–22, the function `balance` tries to invoke the enclosing procedure `swap`, using appropriate actual parameters. However, `balance` contains two nested 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 8–22 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);
      RETURN x;
    END balance;

  BEGIN  -- enclosing procedure swap
    NULL;
  END swap;

BEGIN  -- anonymous block
  NULL;
END;   -- anonymous block
/
```

Result:

```
      swap(num1, num2);
      *
ERROR at line 33:
ORA-06550: line 33, column 7:
PLS-00306: wrong number or types of arguments in call to 'SWAP'
ORA-06550: line 33, column 7:
PL/SQL: Statement ignored
```

Overloaded Subprograms

PL/SQL lets you overload nested subprograms, package subprograms, and type methods. You can use the same name for several different subprograms if their formal parameters differ in name, number, order, or data type family. (A **data type family** is a data type and its subtypes. For the data type families of predefined PL/SQL data types, see Appendix E, "PL/SQL Predefined Data Types". For information about user-defined PL/SQL subtypes, see "User-Defined PL/SQL Subtypes" on page 3-11.) If formal parameters differ only in name, then you must use named notation to specify the corresponding actual parameters. (For information about named notation, see "Positional, Named, and Mixed Notation for Actual Parameters" on page 8-21.)

Example 8–23 defines two subprograms with the same name, `initialize`. The procedures initialize different types of collections. Because the processing in the 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 type body. 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 8–23 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
    DBMS_OUTPUT.PUT_LINE('Invoked first version');
    FOR i IN 1..n LOOP
      tab(i) := SYSDATE;
    END LOOP;
  END initialize;

  PROCEDURE initialize (tab OUT num_tab_typ, n INTEGER) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE('Invoked second version');
    FOR i IN 1..n LOOP
      tab(i) := 0.0;
    END LOOP;
  END initialize;

BEGIN
  initialize(hiredate_tab, 50);
  initialize(sal_tab, 100);
END;
/
```

Result:

```
Invoked first version
Invoked second version
```

For an example of an overloaded procedure in a package, see Example 10–8 on page 10-12.

Topics

- Formal Parameters that Differ Only in Numeric Data Type
- Subprograms that You Cannot Overload
- Subprogram Overload Errors

Formal Parameters that Differ Only in Numeric Data Type

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 be more precise.

To avoid problems or unexpected results when 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 like '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:

```
PROCEDURE s (p IN  VARCHAR2) IS ...
PROCEDURE s (p OUT VARCHAR2) IS ...
```

- Subprograms whose formal parameters differ only in subtype; for example:

```
PROCEDURE s (p INTEGER) IS ...
PROCEDURE s (p REAL) IS ...
```

  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:

```
FUNCTION f (p INTEGER) RETURN BOOLEAN IS ...
FUNCTION f (p INTEGER) RETURN INTEGER IS ...
```

Subprogram Overload Errors

The PL/SQL compiler catches overload errors as soon as it determines that it cannot 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 nested) or when you try to compile the package specification that declares them. Otherwise, the compiler catches the error when you try to compile an ambiguous invocation of a subprogram.

When you try to compile the package specification in Example 8–24, which declares subprograms with identical headings, you get compile-time error PLS-00305.

Example 8–24 Overload Error 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 8–25 violates the rule that you cannot overload subprograms whose formal parameters differ only in subtype, you can compile it without error.

Example 8–25 Overload Error 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 8–26, you get compile-time error PLS-00307.

Example 8–26 Invoking Subprogram in Example 8–25 Causes Compile-Time Error

```
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 8–25 by giving the formal parameters of the overloaded subprograms different names, as in Example 8–27.

Example 8–27 Correcting Overload Error in Example 8–25

```
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 8–28. (If you specify the actual parameter with positional notation, as in Example 8–26, you still get compile-time error PLS-00307.)

Example 8–28 Invoking Subprogram in Example 8–27

```
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 8–29 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 8–30.

Example 8–29 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 8–30 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;
/
```

Recursive Subprograms

A **recursive subprogram** invokes itself. Recursion is a powerful technique for simplifying an algorithm.

A recursive subprogram must have at least two execution paths—one leading to the recursive invocation and one leading to a terminating condition. Without the latter, recursion continues until PL/SQL runs out of memory and raises the predefined exception STORAGE_ERROR.

In Example 8–31, the function implements the following recursive definition of *n* factorial (*n!*), the product of all integers from 1 to *n*:

```
n! = n * (n - 1)!
```

Example 8–31 Recursive Function Returns n Factorial (n!)

```
CREATE OR REPLACE FUNCTION factorial (
  n POSITIVE
) RETURN POSITIVE
IS
BEGIN
  IF n = 1 THEN                 -- terminating condition
    RETURN n;
  ELSE
    RETURN n * factorial(n-1);  -- recursive invocation
  END IF;
END;
/
BEGIN
  FOR i IN 1..5 LOOP
    DBMS_OUTPUT.PUT_LINE(i || '! = ' || factorial(i));
  END LOOP;
END;
/
```

Result:

```
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
```

In Example 8–32, the function returns the *n*th Fibonacci number, which is the sum of the *n*-1st and *n*-2nd Fibonacci numbers. The first and second Fibonacci numbers are zero and one, respectively.

Example 8–32 Recursive Function Returns nth Fibonacci Number

```
CREATE OR REPLACE FUNCTION fibonacci (
  n PLS_INTEGER
) RETURN PLS_INTEGER
IS
  fib_1 PLS_INTEGER := 0;
  fib_2 PLS_INTEGER := 1;
BEGIN
  IF n = 1 THEN                              -- terminating condition
    RETURN fib_1;
  ELSIF n = 2 THEN
    RETURN fib_2;                           -- terminating condition
  ELSE
    RETURN fibonacci(n-2) + fibonacci(n-1);  -- recursive invocations
  END IF;
END;
/
```

```
BEGIN
  FOR i IN 1..10 LOOP
    DBMS_OUTPUT.PUT(fibonacci(i));
    IF i < 10 THEN
      DBMS_OUTPUT.PUT(', ');
    END IF;
  END LOOP;

  DBMS_OUTPUT.PUT_LINE(' ...');
END;
/
```

Result:

```
0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ...
```

Note: The function in Example 8–32 is a good candidate for result caching. For more information, see "Result-Cached Recursive Function" on page 8-36.

Each recursive invocation of a subprogram creates an instance of each item that the subprogram declares and each SQL statement that it executes.

A recursive invocation inside a cursor `FOR LOOP` statement, or between an `OPEN` or `OPEN FOR` statement and a `CLOSE` statement, opens another cursor at each invocation, which might cause the number of open cursors to exceed the limit set by the database initialization parameter `OPEN_CURSORS`.

Subprogram Side Effects

A subprogram has side effects if it changes anything except the values of its own local variables. For example, a subprogram that changes any of the following has side effects:

- Its own `OUT` or `IN OUT` parameter
- A global variable
- A public variable in a package
- A database table
- The database
- The external state (by invoking `DBMS_OUTPUT` or sending e-mail, for example)

Minimizing side effects is especially important when defining a result-cached function or a stored function for SQL statements to invoke.

PL/SQL Function Result Cache

The PL/SQL function result caching mechanism provides a language-supported and system-managed way to cache 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 invocation of 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 Declaration and Definition" on page 13-83.

In Example 8–33, the package `department_pkg` declares and then defines a result-cached function, `get_dept_info`, which returns a record of information about a given department. The function depends on the database tables `DEPARTMENTS` and `EMPLOYEES`.

Example 8–33 Declaring and Defining Result-Cached Function

```
CREATE OR REPLACE PACKAGE department_pkg IS

  TYPE dept_info_record IS RECORD (
    dept_name  departments.department_name%TYPE,
    mgr_name   employees.last_name%TYPE,
    dept_size  PLS_INTEGER
  );

  -- Function declaration

  FUNCTION get_dept_info (dept_id PLS_INTEGER)
    RETURN dept_info_record
    RESULT_CACHE;
```

```
END department_pkg;
/
CREATE OR REPLACE PACKAGE BODY department_pkg IS
  -- Function definition
  FUNCTION get_dept_info (dept_id PLS_INTEGER)
    RETURN dept_info_record
    RESULT_CACHE RELIES_ON (DEPARTMENTS, EMPLOYEES)
  IS
    rec  dept_info_record;
  BEGIN
    SELECT department_name INTO rec.dept_name
    FROM departments
    WHERE department_id = dept_id;

    SELECT e.last_name INTO rec.mgr_name
    FROM departments d, employees e
    WHERE d.department_id = dept_id
    AND d.manager_id = e.employee_id;

    SELECT COUNT(*) INTO rec.dept_size
    FROM EMPLOYEES
    WHERE department_id = dept_id;

    RETURN rec;
  END get_dept_info;
END department_pkg;
/
```

You invoke the function `get_dept_info` as you invoke any function. For example, this invocation returns a record of information about department number 10:

```
department_pkg.get_dept_info(10);
```

This invocation returns only the name of department number 10:

```
department_pkg.get_dept_info(10).department_name;
```

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 `DEPARTMENTS` and `EMPLOYEES` tables, any committed change to `DEPARTMENTS` or `EMPLOYEES` invalidates all cached results for `get_dept_info`, relieving you of programming cache invalidation logic everywhere that `DEPARTMENTS` or `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 8-37)

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 information about side effects, see "Subprogram Side Effects" on page 8-31.
- It does not depend on session-specific settings.

  For more information, see "Making Result-Cached Functions Handle Session-Specific Settings" on page 8-38.
- It does not depend on session-specific application contexts.

  For more information, see "Making Result-Cached Functions Handle Session-Specific Application Contexts" on page 8-39.

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
   val   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
   val     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
   val      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 8–34 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 8–34 Result-Cached Function 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 val 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 val 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 val 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 invocations 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 8-42).

- 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 8–35 Result-Cached Function Handles 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 invoke `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 SQL 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 a default value, so that not every user must specify it.

In Example 8–36, 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 8–36 Result-Cached Function Handles Session-Specific Application Context

```
CREATE OR REPLACE FUNCTION get_param_value (
  param_name VARCHAR,
  appctx     VARCHAR  DEFAULT SYS_CONTEXT('Config', 'App_ID')
) RETURN VARCHAR
  RESULT_CACHE
IS
  rec VARCHAR(2000);
BEGIN
  SELECT val INTO rec
  FROM config_tab
  WHERE name = param_name;

  RETURN rec;
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 must 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 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 8–37 and Example 8–38, 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 8–37, `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 invocation of `get_product_name_1` adds at most one `TRANSLATED_NAME` to the cache.

Example 8–37 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 8–38, `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 invocation of `all_product_names` adds every `TRANSLATED_NAME` of at most one `LANGUAGE_ID` to the cache.

Example 8–38 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 package 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.

PL/SQL Functions that SQL Statements Can Invoke

To be invocable 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 subprogram cannot modify any database tables.
- When invoked from an `INSERT`, `UPDATE`, or `DELETE` statement, the subprogram cannot query or modify any database tables modified by that statement.

  If a function either queries or modifies a table, and a DML statement on that table invokes the function, then ORA-04091 (mutating-table error) occurs. There is one exception: ORA-04091 does not occur if a single-row `INSERT` statement that is not in a `FORALL` statement invokes the function in a `VALUES` clause.
- When invoked from a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, the subprogram cannot execute any of the following SQL statements (unless `PRAGMA AUTONOMOUS_TRANSACTION` was specified):
  - Transaction control statements (such as `COMMIT`)
  - Session control statements (such as `SET ROLE`)
  - System control statements (such as `ALTER SYSTEM`)
  - Database definition language (DDL) statements (such as `CREATE`), which are committed automatically

  (For the description of `PRAGMA AUTONOMOUS_TRANSACTION`, see "AUTONOMOUS_TRANSACTION Pragma" on page 13-6.)

If any SQL statement in the execution part of the function violates a rule, then a run-time error occurs when that statement is parsed.

The fewer side effects a function has, the better it can be optimized in a `SELECT` statement, especially if the function is declared with the option `DETERMINISTIC` or `PARALLEL_ENABLE` (for descriptions of these options, see "DETERMINISTIC" on page 13-84 and "PARALLEL_ENABLE" on page 13-85).

See Also:

- *Oracle Database Advanced Application Developer's Guide* for information about restrictions on PL/SQL functions that SQL statements can invoke
- "Tune Function Invocations in Queries" on page 12-5

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 14-43
- "CREATE PROCEDURE Statement" on page 14-50
- "CREATE TYPE Statement" on page 14-68
- "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 must 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 must 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 check for the criterion that a user must never see source text 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

If a view expression invokes an IR subprogram, then the user who created the view, not the user who is querying the view, is considered to be the current user.

If a trigger invokes an IR subprogram, then the user who created the trigger, not the user who is running the triggering statement, is considered to be the current user.

> **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 8–39.

Example 8–39 ADT for Use in Any Schema

```
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 invocations 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 8–39 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 8–40 shows.

Example 8–40 Invoking 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;
/
```

External Subprograms

If a C procedure or Java method is stored in the database, you can publish it as an external subprogram and then invoke it from PL/SQL.

To publish an external subprogram, define a stored PL/SQL subprogram with a call specification. The call specification maps the name, parameter types, and return type of the external subprogram to PL/SQL equivalents. Invoke the published external subprogram by its PL/SQL name.

For example, suppose that this Java class, `Adjuster`, is stored in the database:

```
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 Java class `Adjuster` has one method, `raiseSalary`, which raises the salary of a specified employee by a specified percentage. Because `raiseSalary` is a `void` method, you publish it as a PL/SQL procedure (rather than a function).

Example 8–41 publishes the stored Java method `Adjuster.raiseSalary` as a PL/SQL standalone stored procedure, mapping the Java method name `Adjuster.raiseSalary` to the PL/SQL procedure name `raise_salary` and the Java data types `int` and `float` to the PL/SQL data type `NUMBER`. Then the anonymous block invokes `raise_salary`.

Example 8–41 PL/SQL Anonymous Block Invokes External Procedure

```
-- Publish Adjuster.raiseSalary as standalone stored PL/SQL procedure:

CREATE OR REPLACE PROCEDURE raise_salary (
  empid NUMBER,
  pct   NUMBER
) AS
  LANGUAGE JAVA NAME 'Adjuster.raiseSalary (int, float)';  -- call specification
/

BEGIN
  raise_salary(120, 10);  -- invoke Adjuster.raiseSalary by PL/SQL name
END;
/
```

Example 8–42 publishes the stored Java method `java.lang.Thread.sleep` as a PL/SQL standalone stored procedure, mapping the Java method name to the PL/SQL procedure name `java_sleep` and the Java data type `long` to the PL/SQL data type `NUMBER`. The PL/SQL standalone stored procedure `sleep` invokes `java_sleep`.

Example 8–42 PL/SQL Standalone Stored Procedure Invokes External Procedure

```
-- 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;
/
```

Call specifications can appear in PL/SQL standalone stored subprograms, package specifications and bodies, and type specifications and bodies. They cannot appear in inside PL/SQL blocks.

> **See Also:** *Oracle Database Advanced Application Developer's Guide* for more information about calling external programs

9

PL/SQL Triggers

A trigger is like a stored procedure that Oracle Database invokes automatically whenever a specified event occurs.

> **Note:** The database can detect only system-defined events. You cannot define your own events.

Topics

- Overview of Triggers
- Reasons to Use Triggers
- DML Triggers
- System Triggers
- Subprograms Invoked by Triggers
- Trigger Compilation, Invalidation, and Recompilation
- Exception Handling in Triggers
- Trigger Design Guidelines
- Trigger Restrictions
- Order in Which Triggers Fire
- Trigger Enabling and Disabling
- Trigger Changing and Debugging
- Triggers and Oracle Database Data Transfer Utilities
- Triggers for Publishing Events
- Views for Information About Triggers

Overview of Triggers

Like a stored procedure, a trigger is a named PL/SQL unit that is stored in the database and can be invoked repeatedly. Unlike a stored procedure, you can enable and disable a trigger, but you cannot explicitly invoke it. While a trigger is **enabled**, the database automatically invokes it—that is, the trigger **fires**—whenever its triggering event occurs. While a trigger is **disabled**, it does not fire.

You create a trigger with the `CREATE TRIGGER` statement. You specify the **triggering event** in terms of **triggering statements** and the item on which they act. The trigger is

said to be **created on** or **defined on** the item, which is either a table, a view, a schema, or the database. You also specify the **timing point**, which determines whether the trigger fires before or after the triggering statement runs and whether it fires for each row that the triggering statement affects. By default, a trigger is created in the enabled state. For more information about the `CREATE TRIGGER` statement, see "CREATE TRIGGER Statement" on page 14-54.

If the trigger is created on a table or view, then the triggering event is composed of DML statements, and the trigger is called a **DML trigger**. For more information, see "DML Triggers" on page 9-3.

If the trigger is created on a schema or the database, then the triggering event is composed of either DDL or database operation statements, and the trigger is called a **system trigger**. For more information, see "System Triggers" on page 9-30.

A **conditional trigger** has a `WHEN` clause that specifies a SQL condition that the database evaluates for each row that the triggering statement affects. For more information about the `WHEN` clause, see "WHEN (condition)" on page 14-65.

When a trigger fires, tables that the trigger references might be undergoing changes made by SQL statements in other users' transactions. SQL statements running in triggers follow the same rules that standalone SQL statements do. Specifically:

- Queries in the trigger see the current read-consistent materialized view of referenced tables and any data changed in the same transaction.
- Updates in the trigger wait for existing data locks to be released before proceeding.

Note: A trigger is often called by the name of its triggering statement (for example, *`DELETE` trigger* or *`LOGON` trigger*), the name of the item on which it is defined (for example, *`DATABASE` trigger* or *`SCHEMA` trigger*), or its timing point (for example, *`BEFORE` statement trigger* or *`AFTER` each row trigger*).

Reasons to Use Triggers

Triggers let you customize your database management system. For example, you can use triggers to:

- Automatically generate calculated column values
- Log events
- Gather statistics on table access
- Modify table data when DML statements are issued against views
- Enforce referential integrity when child and parent tables are on different nodes of a distributed database
- Publish information about database events, user events, and SQL statements to subscribing applications
- Prevent DML operations on a table after regular business hours
- Prevent invalid transactions
- Enforce complex business or referential integrity rules that you cannot define with constraints (see "How Triggers and Constraints Differ" on page 9-3)

Caution: Triggers are not reliable security mechanisms, because they are programmatic and easy to disable. For high-assurance security, use Oracle Database Vault, described in *Oracle Database Vault Administrator's Guide*.

How Triggers and Constraints Differ

Both triggers and constraints can constrain data input, but they differ significantly.

A constraint applies to both existing and new data. For example, if a database column has a `NOT NULL` constraint, then its existing data is `NOT NULL` and no DML statement can violate the `NOT NULL` constraint.

A trigger applies only to new data. For example, a trigger can prevent a DML statement from inserting a `NULL` value into a database column, but the column might contain `NULL` values that were inserted into the column before the trigger was defined or while the trigger was disabled.

Constraints are easier to write and less error-prone than triggers that enforce the same rules. However, triggers can enforce some complex business rules that constraints cannot. 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 or referential integrity rules that you cannot define with constraints

See Also:

- *Oracle Database Advanced Application Developer's Guide* for information about using constraints to enforce business rules and prevent the entry of invalid information into tables
- "Triggers for Ensuring Referential Integrity" on page 9-19 for information about using triggers and constraints to maintain referential integrity between parent and child tables

DML Triggers

A **DML trigger** is created on either a table or view, and its triggering event is composed of the DML statements `DELETE`, `INSERT`, and `UPDATE`. To create a trigger that fires in response to a `MERGE` statement, create triggers on the `INSERT` and `UPDATE` statements to which the `MERGE` operation decomposes.

A DML trigger is either simple or compound.

A **simple DML trigger** fires at exactly one of these timing points:

- Before the triggering statement runs

  (The trigger is called a *`BEFORE` statement trigger* or *statement-level `BEFORE` trigger*.)
- After the triggering statement runs

  (The trigger is called an *`AFTER` statement trigger* or *statement-level `AFTER` trigger*.)
- Before each row that the triggering statement affects

  (The trigger is called a *`BEFORE` each row trigger* or *row-level `BEFORE` trigger*.)

- After each row that the triggering statement affects

  (The trigger is called an *AFTER each row trigger* or *row-level AFTER trigger*.)

A **compound DML trigger** created on a table or editioning view can fire at one, some, or all of the preceding timing points. Compound DML triggers help program an approach where you want the actions that you implement for the various timing points to share common data. For more information, see "Compound DML Triggers" on page 9-14.

A simple or compound DML trigger that fires at row level can access the data in the row that it is processing. For details, see "Correlation Names and Pseudorecords" on page 9-5.

An **INSTEAD OF trigger** is a DML trigger created on a noneditioning view, or on a nested table column of a noneditioning view. The database fires the INSTEAD OF trigger instead of running the triggering DML statement. For more information, see "INSTEAD OF Triggers" on page 9-10.

A **crossedition trigger** is a simple or compound DML trigger for use only in edition-based redefinition. For information about crossedition triggers, see *Oracle Database Advanced Application Developer's Guide*.

Except in an INSTEAD OF trigger, a triggering UPDATE statement can include a column list. With a column list, the trigger fires only when a specified column is updated. Without a column list, the trigger fires when any column of the associated table is updated. For more information about the column list, see "dml_event_clause" on page 14-60.

Topics

- Conditional Predicates for Detecting Triggering DML Statement
- Correlation Names and Pseudorecords
- OBJECT_VALUE Pseudocolumn
- INSTEAD OF Triggers
- Compound DML Triggers
- Triggers for Ensuring Referential Integrity

Conditional Predicates for Detecting Triggering DML Statement

The triggering event of a DML trigger can be composed of multiple triggering statements. When one of them fires the trigger, the trigger can determine which one by using these **conditional predicates**:

Conditional Predicate	TRUE if and only if:
INSERTING	An INSERT statement fired the trigger.
UPDATING	An UPDATE statement fired the trigger.
UPDATING ('*column*')	An UPDATE statement statement that affected the specified column fired the trigger.
DELETING	A DELETE statement fired the trigger.

A conditional predicate can appear wherever a BOOLEAN expression can appear.

Example 9–1 creates a DML trigger that uses conditional predicates to determine which of its four possible triggering statements fired it.

Example 9–1 Trigger Uses Conditional Predicates to Detect Triggering Statement

```
CREATE OR REPLACE TRIGGER t
  BEFORE
    INSERT OR
    UPDATE OF salary OR
    UPDATE OF department_id OR
    DELETE
  ON employees
BEGIN
  CASE
    WHEN INSERTING THEN
      DBMS_OUTPUT.PUT_LINE('Inserting');
    WHEN UPDATING('salary') THEN
      DBMS_OUTPUT.PUT_LINE('Updating salary');
    WHEN UPDATING('department_id') THEN
      DBMS_OUTPUT.PUT_LINE('Updating department ID');
    WHEN DELETING THEN
      DBMS_OUTPUT.PUT_LINE('Deleting');
  END CASE;
END;
/
```

Correlation Names and Pseudorecords

> **Note:** This topic applies only to triggers that fire at row level—that is, row-level simple DML triggers and compound DML triggers with row-level timing point sections.

A trigger that fires at row level can access the data in the row that it is processing by using **correlation names**. The default correlation names are `OLD`, `NEW`, and `PARENT`. To change the correlation names, use the `REFERENCING` clause of the `CREATE TRIGGER` statement (see "referencing_clause ::=" on page 14-56).

If the trigger is created on a nested table in a view (see "dml_event_clause ::=" on page 14-56), then `OLD` and `NEW` refer to the current row of the nested table, and `PARENT` refers to the current row of the parent table. If the trigger is created on a table or view, then `OLD` and `NEW` refer to the current row of the table or view, and `PARENT` is undefined.

`OLD`, `NEW`, and `PARENT` are also called **pseudorecords**, because they have record structure, but are allowed in fewer contexts than records are. The structure of a pseudorecord is *`table_name`*`%ROWTYPE`, where *`table_name`* is the name of the table on which the trigger is created (for `OLD` and `NEW`) or the name of the parent table (for `PARENT`).

In the *`trigger_body`* of a simple trigger or the *`tps_body`* of a compound trigger, a correlation name is a placeholder for a bind argument. Reference the field of a pseudorecord with this syntax:

```
:pseudorecord_name.field_name
```

In the `WHEN` clause of a conditional trigger, a correlation name is not a placeholder for a bind argument. Therefore, omit the colon in the preceding syntax.

Table 9–1 shows the values of `OLD` and `NEW` fields for the row that the triggering statement is processing.

Table 9–1 OLD and NEW Pseudorecord Field Values

Triggering Statement	OLD.*field* Value	NEW.*field* Value
INSERT	NULL	Post-insert value
UPDATE	Pre-update value	Post-update value
DELETE	Pre-delete value	NULL

The restrictions on pseudorecords are:

- A pseudorecord cannot appear in a record-level operation.

  For example, the trigger cannot include this statement:

  ```
  :NEW := NULL;
  ```

- A pseudorecord cannot be an actual subprogram parameter.

  (A pseudorecord field can be an actual subprogram parameter.)

- The trigger cannot change OLD field values.

  Trying to do so raises ORA-04085.

- If the triggering statement is DELETE, then the trigger cannot change NEW field values.

  Trying to do so raises ORA-04084.

- An AFTER trigger cannot change NEW field values, because the triggering statement runs before the trigger fires.

  Trying to do so raises ORA-04084.

A BEFORE trigger can change NEW field values before a triggering INSERT or UPDATE statement puts them in the table.

If a statement triggers both a BEFORE trigger and an AFTER trigger, and the BEFORE trigger changes a NEW field value, then the AFTER trigger "sees" that change.

Example 9–2 creates a log table and a trigger that inserts a row in the log table after any UPDATE statement affects the SALARY column of the EMPLOYEES table, and then updates EMPLOYEES.SALARY and shows the log table.

Example 9–2 Trigger Logs Changes to EMPLOYEES.SALARY

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 that inserts row in log table after EMPLOYEES.SALARY is updated:

```
CREATE OR REPLACE TRIGGER log_salary_increase
  AFTER UPDATE OF salary ON employees
  FOR EACH ROW
BEGIN
  INSERT INTO Emp_log (Emp_id, Log_date, New_salary, Action)
  VALUES (:NEW.employee_id, SYSDATE, :NEW.salary, 'New Salary');
END;
```

```
/
```

Update EMPLOYEES.SALARY:

```
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 28-APR-10   15049.13 New Salary
       202 28-APR-10    6945.75 New Salary

2 rows selected.
```

Example 9–3 creates a conditional trigger that prints salary change information whenever a DELETE, INSERT, or UPDATE statement affects the EMPLOYEES table—unless that information is about the President. The database evaluates the WHEN condition for each affected row. If the WHEN condition is TRUE for an affected row, then the trigger fires for that row before the triggering statement runs. If the WHEN condition is not TRUE for an affected row, then trigger does not fire for that row, but the triggering statement still runs.

Example 9–3 Conditional Trigger Prints Salary Change Information

```
CREATE OR REPLACE TRIGGER print_salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON employees
  FOR EACH ROW
  WHEN (NEW.job_id <> 'AD_PRES')  -- do not print information about President
DECLARE
  sal_diff  NUMBER;
BEGIN
  sal_diff  := :NEW.salary  - :OLD.salary;
  DBMS_OUTPUT.PUT(:NEW.last_name || ': ');
  DBMS_OUTPUT.PUT('Old salary = ' || :OLD.salary || ', ');
  DBMS_OUTPUT.PUT('New salary = ' || :NEW.salary || ', ');
  DBMS_OUTPUT.PUT_LINE('Difference: ' || sal_diff);
END;
/
```

Query:

```
SELECT last_name, department_id, salary, job_id
FROM employees
WHERE department_id IN (10, 20, 90)
ORDER BY department_id, last_name;
```

Result:

```
LAST_NAME                 DEPARTMENT_ID     SALARY JOB_ID
------------------------- ------------- ---------- ----------
Whalen                               10       2800 AD_ASST
```

```
Fay                                 20       6000 MK_REP
Hartstein                           20      13000 MK_MAN
De Haan                             90      17000 AD_VP
King                                90      24000 AD_PRES
Kochhar                             90      17000 AD_VP

6 rows selected.
```

Triggering statement:

```
UPDATE employees
SET salary = salary * 1.05
WHERE department_id IN (10, 20, 90);
```

Result:

```
Whalen: Old salary = 2800, New salary = 2940, Difference: 140
Hartstein: Old salary = 13000, New salary = 13650, Difference: 650
Fay: Old salary = 6000, New salary = 6300, Difference: 300
Kochhar: Old salary = 17000, New salary = 17850, Difference: 850
De Haan: Old salary = 17000, New salary = 17850, Difference: 850

6 rows updated.
```

Query:

```
SELECT salary FROM employees WHERE job_id = 'AD_PRES';
```

Result:

```
    SALARY
----------
     25200

1 row selected.
```

Example 9–4 creates a trigger that modifies CLOB columns. (For information about TO_CLOB and other conversion functions, see *Oracle Database SQL Language Reference*.)

Example 9–4 Trigger 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;
```

Example 9–5 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 9–5 Trigger with REFERENCING Clause

```
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;
/
```

OBJECT_VALUE Pseudocolumn

A trigger on an object table can reference the SQL pseudocolumn `OBJECT_VALUE`, which returns system-generated names for the columns of the object table. The trigger can also invoke a PL/SQL subprogram that has a formal `IN` parameter whose data type is `OBJECT_VALUE`.

> **See Also:** *Oracle Database SQL Language Reference* for more information about `OBJECT_VALUE`

Example 9–6 creates object table `tbl`, table `tbl_history` for logging updates to `tbl`, and trigger `Tbl_Trg`. The trigger runs for each row of `tb1` that is affected by a DML statement, causing the old and new values of the object `t` in `tbl` to be written in `tbl_history`. The old and new values are :`OLD.OBJECT_VALUE` and :`NEW.OBJECT_VALUE`.

Example 9–6 Trigger References 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 ORDER BY n;
```

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:

```
28-APR-10 -- old: 1 0 -- new: 2 0
28-APR-10 -- old: 2 0 -- new: 3 0
28-APR-10 -- old: 3 0 -- new: 4 0
28-APR-10 -- old: 4 0 -- new: 5 0
28-APR-10 -- old: 5 0 -- new: 6 0
```

All values of column `n` were increased by 1. The value of `m` remains 0.

INSTEAD OF Triggers

An **`INSTEAD OF` trigger** is a DML trigger created on a noneditioning view, or on a nested table column of a noneditioning view. The database fires the `INSTEAD OF` trigger instead of running the triggering DML statement. An `INSTEAD OF` trigger cannot be conditional.

An `INSTEAD OF` trigger is the only way to update a view that is not inherently updatable. (For information about inherently updatable views, see *Oracle Database SQL Language Reference*.) Design the `INSTEAD OF` trigger to determine what operation was intended and do the appropriate DML operations on the underlying tables.

An INSTEAD OF trigger is always a row-level trigger. An INSTEAD OF trigger can read OLD and NEW values, but cannot change them.

Example 9–7 creates the view oe.order_info to display information about customers and their orders. The view is not inherently updatable (because the primary key of the orders table, order_id, is not unique in the result set of the join view). The example creates an INSTEAD OF trigger to process INSERT statements directed to the view. The trigger inserts rows into the base tables of the view, customers and orders.

Example 9–7 INSTEAD OF Trigger

```
CREATE OR REPLACE 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;

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;
/
```

Query to show that row to be inserted does not exist:

```
SELECT COUNT(*) FROM order_info WHERE customer_id = 999;
```

Result:

```
  COUNT(*)
----------
         0

1 row selected.
```

Insert row into view:

```
INSERT INTO order_info VALUES
   (999, 'Smith', 'John', 2500, '13-MAR-2001', 0);
```

Result:

```
1 row created.
```

Query to show that row has been inserted in view:

```
SELECT COUNT(*) FROM order_info WHERE customer_id = 999;
```

Result:

```
  COUNT(*)
----------
         1

1 row selected.
```

Query to show that row has been inserted in `customers` table:

```
SELECT COUNT(*) FROM customers WHERE customer_id = 999;
```

Result:

```
  COUNT(*)
----------
         1

1 row selected.
```

Query to show that row has been inserted in `orders` table:

```
SELECT COUNT(*) FROM orders WHERE customer_id = 999;
```

Result:

```
  COUNT(*)
----------
         1

1 row selected.
```

INSTEAD OF Triggers on Nested Table Columns of Views

An `INSTEAD OF` trigger with the `NESTED TABLE` clause fires only if the triggering statement operates on the elements of the specified nested table column of the view. The trigger fires for each modified nested table element.

In Example 9–8, the view `dept_view` contains a nested table of employees, `emplist`, created by the `CAST` function (described in *Oracle Database SQL Language Reference*). To modify the `emplist` column, the example creates an `INSTEAD OF` trigger on the column.

Example 9–8 INSTEAD OF Trigger on Nested Table Column of View

```
-- 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;

-- Create trigger:

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;
/
```

Query view before inserting row into nested table:

```
SELECT emplist FROM dept_view WHERE department_id=10;
```

Result:

```
EMPLIST(EMP_ID, LASTNAME, JOB, SAL)
-----------------------------------------------

EMP_LIST_(NTE(200, 'Whalen', 'AD_ASST', 2800))

1 row selected.
```

Query table before inserting row into nested table:

```
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          2800

1 row selected.
```

Insert a row into nested table:

```
INSERT INTO TABLE (
  SELECT d.emplist
  FROM dept_view d
  WHERE department_id = 10
)
VALUES (1001, 'Glenn', 'AC_MGR', 10000);
```

Query view after inserting row into nested table:

```
SELECT emplist FROM dept_view WHERE department_id=10;
```

Result (formatted to fit page):

```
EMPLIST(EMP_ID, LASTNAME, JOB, SAL)
-------------------------------------------------------------------------------

EMP_LIST_(NTE(200, 'Whalen', 'AD_ASST', 2800),
          NTE(1001, 'Glenn', 'AC_MGR', 10000))

1 row selected.
```

Query table after inserting row into nested table:

```
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          2800
       1001 Glenn                     AC_MGR          10000

2 rows selected.
```

Compound DML Triggers

A compound DML trigger created on a table or editioning view can fire at multiple timing points. Each timing point section has its own executable part and optional exception-handling part, but all of these parts 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.

A compound DML trigger created on a noneditioning view is not really compound, because it has only one timing point section.

A compound trigger can be conditional, but not autonomous.

Two common uses of compound triggers are:

- To accumulate rows destined for a second table so that you can periodically bulk-insert them
- To avoid the mutating-table error (ORA-04091)

Topics

- Compound DML Trigger Structure
- Compound DML Trigger Restrictions
- Performance Benefit of Compound DML Triggers
- Using Compound DML Triggers with Bulk Insertion
- Using Compound DML Triggers to Avoid Mutating-Table Error

Compound DML Trigger Structure

The optional declarative part of a compound trigger declares variables and subprograms that all of its timing-point sections can use. When the trigger fires, the declarative part runs before any timing-point sections run. The variables and subprograms exist for the duration of the triggering statement.

A compound DML trigger created on a noneditioning view is not really compound, because it has only one timing point section. The syntax for creating the simplest compound DML trigger on a noneditioning view is:

```
CREATE trigger FOR dml_event_clause ON view
COMPOUND TRIGGER
INSTEAD OF EACH ROW IS BEGIN
  statement;
END INSTEAD OF EACH ROW;
```

A compound DML trigger created on a table or editioning view has at least one timing-point section in Table 9–2. If the trigger has multiple timing-point sections, they can be in any order, but no timing-point section can be repeated. If a timing-point section is absent, then nothing happens at its timing point.

Table 9–2 Compound Trigger Timing-Point Sections

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`

> **See Also:** "CREATE TRIGGER Statement" on page 14-54 for more information about the syntax of compound triggers

A compound DML trigger does not have an initialization section, but the `BEFORE STATEMENT` section, which runs before any other timing-point section, can do any necessary initializations.

If a compound DML trigger has neither a BEFORE STATEMENT section nor an AFTER STATEMENT section, and its triggering statement affects no rows, then the trigger never fires.

Compound DML Trigger Restrictions

In addition to the "Trigger Restrictions" on page 9-36), compound DML triggers have these restrictions:

- 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.
- A timing-point section cannot handle exceptions raised in another timing-point section.
- If a timing-point section includes a GOTO statement, the target of the GOTO statement must be in the same timing-point section.

Performance Benefit of Compound DML Triggers

A compound DML trigger has a performance benefit when the triggering statement affects many rows.

For example, suppose that this statement triggers a compound DML trigger that has all four timing-point sections in Table 9–2:

```
INSERT INTO Target
  SELECT c1, c2, c3
  FROM Source
  WHERE Source.c1 > 0
```

Although the BEFORE EACH ROW and AFTER EACH ROW sections of the trigger run for each row of Source whose column c1 is greater than zero, the BEFORE STATEMENT section runs only before the INSERT statement runs and the AFTER STATEMENT section runs only after the INSERT statement runs.

A compound DML trigger has a greater performance benefit when it uses bulk SQL, described in "Bulk SQL and Bulk Binding" on page 12-11.

Using Compound DML Triggers with Bulk Insertion

A compound DML trigger is useful for accumulating rows destined for a second table so that you can periodically bulk-insert them. To get the performance benefit from the compound trigger, you must specify BULK COLLECT INTO in the FORALL statement (otherwise, the FORALL statement does a single-row DML operation multiple times). For more information about using the BULK COLLECT clause with the FORALL statement, see "Using FORALL Statement and BULK COLLECT Clause Together" on page 12-36.

> **See Also:** "FORALL Statement" on page 12-11

Scenario: You want to log every change to hr.employees.salary in a new table, employee_salaries. A single UPDATE statement updates 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 9–9. 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 9–9, you must have the EXECUTE privilege on the package DBMS_LOCK.

Example 9–9 Compound Trigger Logs 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 DML Triggers to Avoid Mutating-Table Error

A compound DML trigger is useful for avoiding the mutating-table error (ORA-04091) explained in "Mutating-Table Restriction" on page 9-37.

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 9–10. The state variables are initialized each time the trigger fires (even when the triggering statement is interrupted and restarted).

Example 9–10 Compound Trigger Avoids 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;

  -- Declare collection type and variable:

  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;
```

Triggers for Ensuring Referential Integrity

You can use triggers and constraints to maintain referential integrity between parent and child tables, as Table 9–3 shows. (For more information about constraints, see *Oracle Database SQL Language Reference*.)

Table 9–3 Constraints and Triggers for Ensuring Referential Integrity

Table	Constraint to Declare on Table	Triggers to Create on Table
Parent	`PRIMARY KEY` or `UNIQUE`	One or more triggers that ensure that when `PRIMARY KEY` or `UNIQUE` values are updated or deleted, the desired action (`RESTRICT`, `CASCADE`, or `SET NULL`) occurs on corresponding `FOREIGN KEY` values. No action is required for inserts into the parent table, because no dependent foreign keys exist.
Child	`FOREIGN KEY`, if parent and child are in the same database. (The database does not support declarative referential constraints between tables on different nodes of a distributed database.) Disable this foreign key constraint to prevent the corresponding `PRIMARY KEY` or `UNIQUE` constraint from being dropped (except explicitly with the `CASCADE` option).	One trigger that ensures that values inserted or updated in the `FOREIGN KEY` correspond to `PRIMARY KEY` or `UNIQUE` values in the parent table.

Topics

- Foreign Key Trigger for Child Table
- UPDATE and DELETE RESTRICT Trigger for Parent Table
- UPDATE and DELETE SET NULL Trigger for Parent Table
- DELETE CASCADE Trigger for Parent Table
- UPDATE CASCADE Trigger for Parent Table
- Triggers for Complex Constraint Checking
- 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

Note: The examples in the following topics use these tables, which share the column `Deptno`:

```
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);

CREATE TABLE dept (
  Deptno    NUMBER(2) NOT NULL,
  Dname     VARCHAR2(14),
  Loc       VARCHAR2(13),
  Mgr_no    NUMBER,
  Dept_type NUMBER);
```

Several triggers include statements that lock rows (`SELECT FOR UPDATE`). This operation is necessary to maintain concurrency while the rows are being processed.

These examples are not meant to be used exactly as written. They are provided to assist you in designing your own triggers.

Foreign Key Trigger for Child Table

The trigger in Example 9–11 ensures that before an `INSERT` or `UPDATE` statement affects a foreign key value, the corresponding value exists in the parent key. The exception ORA-04091 (mutating-table error) allows the trigger `emp_dept_check` to be used with the `UPDATE_SET_DEFAULT` and `UPDATE_CASCADE` triggers. This exception is unnecessary if the trigger `emp_dept_check` is used alone.

Example 9–11 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 code and 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 Trigger for Parent Table

The trigger in Example 9–12 enforces the `UPDATE` and `DELETE RESTRICT` referential action on the primary key of the `dept` table.

> **Caution:** The trigger in Example 9–12 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 9–12 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 code 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 Trigger for Parent Table

The trigger in Example 9–13 enforces the `UPDATE` and `DELETE SET NULL` referential action on the primary key of the `dept` table.

Example 9–13 UPDATE and DELETE SET NULL Trigger 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 9–14 enforces the `DELETE CASCADE` referential action on the primary key of the `dept` table.

Example 9–14 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 9–15 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 9–15 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 9–11, 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 Constraint Checking

Triggers can enforce integrity rules other than referential integrity. The trigger in Example 9–16 does a complex check before allowing the triggering statement to run.

Note: Example 9–16 needs this data structure:

```
CREATE TABLE Salgrade (
  Grade               NUMBER,
  Losal               NUMBER,
  Hisal               NUMBER,
  Job_classification  NUMBER);
```

Example 9–16 Trigger Checks Complex 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 done only for the triggering statement, not for each row affected by the triggering statement.

The trigger in Example 9–17 enforces security.

Note: Example 9–17 needs this data structure:

```
CREATE TABLE Company_holidays (Day DATE);
```

Example 9–17 Trigger Enforces 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 do 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 9–18 derives new column values for a table whenever a row is inserted or updated.

Note: Example 9–18 needs this change to this data structure:

```
ALTER TABLE Emp ADD(
   Uppername   VARCHAR2(20),
   Soundexname VARCHAR2(20));
```

Example 9–18 Trigger Derives New Column Values

```
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 ORDER BY Booknum;
```

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 ORDER BY Section;
```

Result:

```
SECTION
--------------------
Classic
Novel

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 ORDER BY Section;
```

Result:

```
SECTION
--------------------
BOOKLIST(BOOKNUM, TITLE, AUTHOR, AVAILABLE)
--------------------------------------------------------------------

Classic
BOOK_LIST_T(BOOK_T(121001, 'Iliad', 'Homer', 'Y'))

History
BOOK_LIST_T(BOOK_T(121330, 'Alexander', 'Mirth', 'Y'))

Novel
BOOK_LIST_T(BOOK_T(121002, 'Gone with the Wind', 'Mitchell M', 'N'))


3 rows selected.
```

See the effect on `Book_table`:

```
SELECT * FROM Book_table ORDER BY Booknum;
```

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 ORDER BY Section;
```

Result:

```
SECTION
--------------------
Classic
History
Novel

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

System Triggers

A **system trigger** is created on either a schema or the database. Its triggering event is composed of either DDL statements (listed in "ddl_event" on page 14-62) or database operation statements (listed in "database_event" on page 14-63).

A system trigger fires at exactly one of these timing points:

- Before the triggering statement runs

  (The trigger is called a *`BEFORE` statement trigger* or *statement-level `BEFORE` trigger*.)

- After the triggering statement runs

  (The trigger is called a *`AFTER` statement trigger* or *statement-level `AFTER` trigger*.)

Topics

- SCHEMA Triggers
- DATABASE Triggers

SCHEMA Triggers

A **SCHEMA trigger** is created on a schema and fires whenever any user connected as that schema owner initiates the triggering event.

Example 9–19 creates a BEFORE statement trigger on the sample schema HR. When a user connected as HR tries to drop a database object, the database fires the trigger before dropping the object.

Example 9–19 BEFORE Statement Trigger on Sample Schema HR

```
CREATE OR REPLACE TRIGGER drop_trigger
  BEFORE DROP ON hr.SCHEMA
  BEGIN
    RAISE_APPLICATION_ERROR (
      num => -20000,
      msg => 'Cannot drop object');
  END;
/
```

DATABASE Triggers

A **DATABASE trigger** is created on the database and fires whenever any database user initiates the triggering event.

Example 9–20 shows the basic syntax for a trigger to log all errors. This trigger fires after an unsuccessful statement execution, such as unsuccessful logon.

Example 9–20 AFTER Statement Trigger on Database

```
CREATE TRIGGER log_errors
  AFTER SERVERERROR ON DATABASE
  BEGIN
    IF (IS_SERVERERROR (1017)) THEN
      NULL;  -- (substitute code that processes logon error)
    ELSE
      NULL;  -- (substitute code that logs error code)
    END IF;
  END;
/
```

The trigger in Example 9–21 runs the procedure check_user after a user logs onto the database.

Example 9–21 Trigger Monitors 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;
```

```
/
```

Subprograms Invoked by Triggers

Triggers can invoke subprograms written in PL/SQL, C, and Java. The trigger in Example 9–9 invokes a PL/SQL subprogram. The trigger in Example 9–22 invokes a Java subprogram.

Example 9–22 Trigger 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;
   }
}
```

A subprogram invoked by a trigger cannot run transaction control statements, because the subprogram runs in the context of the trigger body.

If a trigger invokes an invoker rights (IR) subprogram, then the user who created the trigger, not the user who ran the triggering statement, is considered to be the current user. For information about IR subprograms, see "Invoker's Rights and Definer's Rights (AUTHID Property)" on page 8-43.

If a trigger invokes a remote subprogram, and a timestamp or signature mismatch is found during execution of the trigger, then the remote subprogram does not run and the trigger is invalidated.

Trigger Compilation, Invalidation, and Recompilation

The `CREATE TRIGGER` statement compiles the trigger and stores its code in the database. If a compilation error occurs, the trigger is still created, but its triggering statement fails, except in these cases:

- The trigger was created in the disabled state.
- The triggering event is `AFTER STARTUP ON DATABASE`.
- The triggering event is either `AFTER LOGON ON DATABASE` or `AFTER LOGON ON SCHEMA`, and someone logs on as `SYSTEM`.

To see trigger compilation errors, either use the `SHOW ERRORS` command in SQL*Plus or Enterprise Manager, or query the static data dictionary view `*_ERRORS` (described in *Oracle Database Reference*).

If a trigger does not compile successfully, then its exception handler cannot run. For an example, see "Remote Exception Handling" on page 9-34.

If a trigger references another object, such as a subprogram or package, and that object is modified or dropped, then the trigger becomes invalid. The next time the triggering event occurs, the compiler tries to revalidate the trigger (for details, see *Oracle Database Advanced Application Developer's Guide*).

> **Note:** Because the `DBMS_AQ` package is used to enqueue a message, dependency between triggers and queues cannot be maintained.

To recompile a trigger manually, use the `ALTER TRIGGER` statement, described in "ALTER TRIGGER Statement" on page 14-14.

Exception Handling in Triggers

In most cases, if a trigger runs a statement that raises an exception, and the exception is not handled by an exception handler, then the database rolls back the effects of both the trigger and its triggering statement.

In the following cases, the database rolls back only the effects of the trigger, not the effects of the triggering statement (and logs the error in trace files and the alert log):

- The triggering event is either `AFTER STARTUP ON DATABASE` or `BEFORE SHUTDOWN ON DATABASE`.
- The triggering event is `AFTER LOGON ON DATABASE` and the user has the `ADMINISTER DATABASE TRIGGER` privilege.
- The triggering event is `AFTER LOGON ON SCHEMA` and the user either owns the schema or has the `ALTER ANY TRIGGER` privilege.

In the case of a compound DML trigger, the database rolls back only the effects of the triggering statement, not the effects of the trigger. However, variables declared in the trigger are re-initialized, and any values computed before the triggering statement was rolled back are lost.

> **Note:** Triggers that enforce complex security authorizations or constraints typically raise user-defined exceptions, which are explained in "User-Defined Exceptions" on page 11-11.

> **See Also:** Chapter 11, "PL/SQL Error Handling," for general information about exception handling

Remote Exception Handling

A trigger that accesses a remote database can do remote exception handling only if the remote database is available. If the remote database is unavailable when the local database must compile the trigger, then the local database cannot validate the statement that accesses the remote database, and the compilation fails. If the trigger cannot be compiled, then its exception handler cannot run.

The trigger in Example 9–23 has an `INSERT` statement that accesses a remote database. The trigger also has an exception handler. However, if the remote database is unavailable when the local database tries to compile the trigger, then the compilation fails and the exception handler cannot run.

Example 9–23 Trigger Cannot Handle Exception if Remote Database is Unavailable

```
CREATE OR REPLACE TRIGGER employees_tr
  AFTER INSERT ON employees
  FOR EACH ROW
BEGIN
  -- When remote database is unavailable, compilation fails here:
  INSERT INTO employees@remote (
    employee_id, first_name, last_name, email, hire_date, job_id
  )
  VALUES (
    99, 'Jane', 'Doe', 'jane.doe@example.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;
/
```

Example 9–24 shows the workaround for the problem in Example 9–23: Put the remote `INSERT` statement and exception handler in a stored subprogram and have the trigger invoke the stored subprogram. The subprogram is stored in the local database in compiled form, with a validated statement for accessing the remote database. Therefore, when the remote `INSERT` statement fails because the remote database is unavailable, the exception handler in the subprogram can handle it.

Example 9–24 Workaround for Example 9–23

```
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@example.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 Design Guidelines

- Use triggers to ensure that whenever a specific event occurs, any necessary actions are done (regardless of which user or application issues the triggering statement).

  For example, use a trigger to ensure that whenever anyone updates a table, its log file is updated.

- Do not create triggers that duplicate database features.

  For example, do not create a trigger to reject invalid data if you can do the same with constraints (see "How Triggers and Constraints Differ" on page 9-3).

- Do not create triggers that depend on the order in which a SQL statement processes rows (which can vary).

  For example, do not assign a value to a global package variable in a row trigger if the current value of the variable depends on the row being processed by the row trigger. If a trigger updates global package variables, initialize those variables in a `BEFORE` statement trigger.

- Use `BEFORE` row triggers to modify the row before writing the row data to disk.
- Use `AFTER` row triggers to obtain the row ID and use it in operations.

  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 are read first for the trigger and then for the triggering statement. With `AFTER` row triggers, affected data blocks are read only for the trigger.

- If the triggering statement of a `BEFORE` statement trigger is an `UPDATE` or `DELETE` statement that conflicts with an `UPDATE` statement that is running, then the database does a transparent `ROLLBACK` to `SAVEPOINT` and restarts the triggering statement. The database can do this many times before the triggering statement completes successfully. Each time the database restarts the triggering statement, the trigger fires. The `ROLLBACK` to `SAVEPOINT` does not undo changes to package variables that the trigger references. To detect this situation, include a counter variable in the package.

- Do not create recursive triggers.

  For example, do not create an `AFTER UPDATE` trigger that issues an `UPDATE` statement on the table on which the trigger is defined. The trigger fires recursively until it runs out of memory.

- If you create a trigger that includes a statement that accesses a remote database, then put the exception handler for that statement in a stored subprogram and invoke the subprogram from the trigger.

  For more information, see "Remote Exception Handling" on page 9-34.

- Use `DATABASE` triggers judiciously. They fire every time any database user initiates a triggering event.

- If a trigger runs the following statement, the statement returns the owner of the trigger, not the user who is updating the table:

  ```
  SELECT Username FROM USER_USERS;
  ```

- Only committed triggers fire.

  A trigger is committed, implicitly, after the `CREATE TRIGGER` statement that creates it succeeds. Therefore, the following statement cannot fire the trigger that it creates:

  ```
  CREATE OR REPLACE TRIGGER my_trigger
    AFTER CREATE ON DATABASE
  BEGIN
    NULL;
  END;
  /
  ```

- To allow the modular installation of applications that have triggers on the same tables, create multiple triggers of the same type, rather than a single trigger that runs a sequence of operations.

  Each trigger sees the changes made by the previously fired triggers. Each trigger can see `OLD` and `NEW` values.

Trigger Restrictions

In addition to the restrictions that apply to all PL/SQL units (see Table C–1), triggers have these restrictions:

- Trigger Size Restriction
- Trigger LONG and LONG RAW Data Type Restrictions
- Mutating-Table Restriction
- Only an autonomous trigger can run TCL or DDL statements.

  For information about autonomous triggers, see "Autonomous Triggers" on page 6-53.

- A trigger cannot invoke a subprogram that runs transaction control statements, because the subprogram runs in the context of the trigger body.

  For more information about subprograms invoked by triggers, see "Subprograms Invoked by Triggers" on page 9-32.

- A trigger cannot access a `SERIALLY_REUSABLE` package.

  For information about `SERIALLY_REUSABLE` packages, see "SERIALLY_REUSABLE Packages" on page 10-7.

See Also: "Compound DML Trigger Restrictions" on page 9-16

Trigger Size Restriction

The size of the trigger cannot exceed 32K.

If the logic for your trigger requires much more than 60 lines of PL/SQL source text, then put most of the source text in a stored subprogram and invoke the subprogram from the trigger. For information about subprograms invoked by triggers, see "Subprograms Invoked by Triggers" on page 9-32.

Trigger LONG and LONG RAW Data Type Restrictions

> **Note:** Oracle supports the LONG and LONG RAW data types only for backward compatibility with existing applications.

In addition to the restrictions that apply to all PL/SQL units (see "LONG and LONG RAW Variables" on page 3-6), triggers have these restrictions:

- A trigger cannot declare a variable of the LONG or LONG RAW data type.
- A SQL statement in a trigger can reference a LONG or LONG RAW column only if the column data can be converted to the data type CHAR or VARCHAR2.
- A trigger cannot use the correlation name NEW or PARENT with a LONG or LONG RAW column.

Mutating-Table Restriction

> **Note:** This topic applies only to row-level simple DML triggers.

A **mutating table** is a table that is being modified by a DML statement (possibly by the effects of a DELETE CASCADE constraint). (A view being modified by an INSTEAD OF trigger is not considered to be mutating.)

The mutating-table restriction prevents the trigger from querying or modifying the table that the triggering statement is modifying. When a row-level trigger encounters a mutating table, ORA-04091 occurs, the effects of the trigger and triggering statement are rolled back, and control returns to the user or application that issued the triggering statement, as Example 9–25 shows.

> **Caution:** Oracle Database does not enforce the mutating-table restriction for a trigger that accesses remote nodes, because the database does not support declarative referential constraints between tables on different nodes of a distributed database.
>
> Similarly, the database does not enforce the mutating-table restriction for 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.

Example 9–25 Trigger Causes Mutating-Table Error

```
-- Create log table

DROP TABLE log;
CREATE TABLE log (
  emp_id  NUMBER(6),
  l_name  VARCHAR2(25),
  f_name  VARCHAR2(20)
);

-- Create trigger that updates log and then reads employees

CREATE OR REPLACE TRIGGER log_deletions
```

```
  AFTER DELETE ON employees
  FOR EACH ROW
DECLARE
  n INTEGER;
BEGIN
  INSERT INTO log VALUES (
    :OLD.employee_id,
    :OLD.last_name,
    :OLD.first_name
  );

  SELECT COUNT(*) INTO n FROM employees;
  DBMS_OUTPUT.PUT_LINE('There are now ' || n || ' employees.');
END;
/

-- Issue triggering statement:

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 might not see it
ORA-06512: at "HR.LOG_DELETIONS", line 10
ORA-04088: error during execution of trigger 'HR.LOG_DELETIONS'
```

Show that effect of trigger was rolled back:

```
SELECT count(*) FROM log;
```

Result:

```
  COUNT(*)
----------
         0

1 row selected.
```

Show that effect of triggering statement was rolled back:

```
SELECT employee_id, last_name FROM employees WHERE employee_id = 197;
```

Result:

```
EMPLOYEE_ID LAST_NAME
----------- -------------------------
        197 Feeney

1 row selected.
```

If you must use a trigger to update a mutating table, you can avoid the mutating-table error in either of these ways:

- Use a compound DML trigger (see "Using Compound DML Triggers to Avoid Mutating-Table Error" on page 9-18).
- Use a temporary table.

  For example, instead of using one `AFTER` each row trigger that updates the mutating table, use two triggers—an `AFTER` each row trigger that updates the

temporary table and an AFTER statement trigger that updates the mutating table with the values from the temporary table.

Mutating-Table Restriction Relaxed

As of Oracle Database 8*g* Release 1 (8.1), a deletion from the parent table causes BEFORE and AFTER triggers to fire once. Therefore, you can create row-level and statement-level triggers that query and modify the parent and child tables. This allows most foreign key constraint actions to be implemented through their after-row triggers (unless the constraint is 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—see "Triggers for Ensuring Referential Integrity" on page 9-19.

However, cascades require care for multiple-row 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.

In Example 9–26, the triggering statement updates p correctly but causes problems when the trigger updates f. First, the triggering statement changes (1) to (2) in p, and the trigger updates (1) to (2) in f, leaving two rows of value (2) in f. Next, the triggering 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 between the data items in p and f is lost.

Example 9–26 Update Cascade

```
DROP TABLE p;
CREATE TABLE p (p1 NUMBER CONSTRAINT pk_p_p1 PRIMARY KEY);
INSERT INTO p VALUES (1);
INSERT INTO p VALUES (2);
INSERT INTO p VALUES (3);

DROP TABLE f;
CREATE TABLE f (f1 NUMBER CONSTRAINT fk_f_f1 REFERENCES p);
INSERT INTO f VALUES (1);
INSERT INTO f VALUES (2);
INSERT INTO f VALUES (3);

CREATE TRIGGER pt
  AFTER UPDATE ON p
  FOR EACH ROW
BEGIN
  UPDATE f SET f1 = :NEW.p1 WHERE f1 = :OLD.p1;
END;
/
```

Query:

```
SELECT * FROM p;
```

Result:

```
        P1
----------
         1
         2
         3
```

Query:

```
SELECT * FROM f;
```

Result:

```
        F1
----------
         1
         2
         3
```

Issue triggering statement:

```
UPDATE p SET p1 = p1+1;
```

Query:

```
SELECT * FROM p;
```

Result:

```
        P1
----------
         2
         3
         4
```

Query:

```
SELECT * FROM f;
```

Result:

```
        F1
----------
         4
         4
         4
```

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.

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. For information about compound triggers, see "Compound DML Triggers" on page 9-14.

If you are creating two or more triggers *with the same timing point*, and the order in which they fire is important, then you can control their firing order using the `FOLLOWS` and `PRECEDES` clauses (see "FOLLOWS | PRECEDES" on page 14-64).

If multiple compound triggers are created 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 the `FOLLOWS` clause determines the order of execution of compound trigger sections. If `FOLLOWS` is specified for some but not all triggers, then the order of execution of triggers is guaranteed only for those that are related using the `FOLLOWS` clause.

- 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 the `PRECEDES` clause determines the order of execution of compound trigger sections . If `PRECEDES` is specified for some 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, which are described in *Oracle Database Advanced Application Developer's Guide*.

The firing of compound triggers can be interleaved with the firing of simple triggers.

When one trigger causes another trigger to fire, the triggers are said to be **cascading**. The database allows up to 32 triggers to cascade simultaneously. To limit the number of trigger cascades, use the initialization parameter `OPEN_CURSORS` (described in *Oracle Database Reference*), because a cursor opens every time a trigger fires.

Trigger Enabling and Disabling

By default, the `CREATE TRIGGER` statement creates a trigger in the enabled state. To create a trigger in the disabled state, specify `DISABLE`. Creating a trigger in the disabled state lets you ensure that it compiles without errors before you enable it.

Some reasons to temporarily disable a trigger are:

- The trigger refers to an unavailable object.
- You must do a large data load, and you want it to proceed quickly without firing triggers.
- You are reloading data.

To enable or disable a single trigger, use this statement:

```
ALTER TRIGGER [schema.]trigger_name { ENABLE | DISABLE };
```

To enable or disable all triggers created on a specific table, use this statement:

```
ALTER TABLE table_name { ENABLE | DISABLE } ALL TRIGGERS;
```

In both of the preceding statements, *`schema`* is the name of the schema containing the trigger, and the default is your schema.

See Also:

- "ALTER TRIGGER Statement" on page 14-14 for more information about the `ALTER TRIGGER` statement
- *Oracle Database SQL Language Reference* for more information about the `ALTER TABLE` statement

Trigger Changing and Debugging

To change a trigger, you must either replace or re-create it. (The `ALTER TRIGGER` statement only enables, disables, compiles, or renames a trigger.)

To replace a trigger, use the `CREATE TRIGGER` statement with the `OR REPLACE` clause.

To re-create a trigger, first drop it with the `DROP TRIGGER` statement and then create it again with the `CREATE TRIGGER` statement.

To debug a trigger, you can use the facilities available for stored subprograms. For information about these facilities, see *Oracle Database Advanced Application Developer's Guide*.

See Also:

- "CREATE TRIGGER Statement" on page 14-54 for more information about the `CREATE TRIGGER` statement
- "DROP TRIGGER Statement" on page 14-97 for more information about the `DROP TRIGGER` statement
- "ALTER TRIGGER Statement" on page 14-14 for more information about the `ALTER TRIGGER` statement

Triggers and Oracle Database Data Transfer Utilities

The Oracle database utilities that transfer data to your database, possibly firing triggers, are:

- **SQL*Loader (`sqlldr`)**

  SQL*Loader loads data from external files into tables of an Oracle database.

  During a SQL*Loader conventional load, `INSERT` triggers fire.

  Before a SQL*Loader direct load, triggers are disabled.

  See Also: *Oracle Database Utilities* for more information about SQL*Loader

- **Data Pump Import (`impdp`)**

  Data Pump Import (`impdp`) reads an export dump file set created by Data Pump Export (`expdp`) and writes it to an Oracle database.

  If a table to be imported does not exist on the target database, or if you specify `TABLE_EXISTS_ACTION=REPLACE`, then `impdp` creates and loads the table before creating any triggers, so no triggers fire.

  If a table to be imported exists on the target database, and you specify either `TABLE_EXISTS_ACTION=APPEND` or `TABLE_EXISTS_ACTION=TRUNCATE`, then `impdp` loads rows into the existing table, and `INSERT` triggers created on the table fire.

See Also: *Oracle Database Utilities* for more information about Data Pump Import

- **Original Import (`imp`)**

  Original Import (the original Import utility, `imp`) reads object definitions and table data from dump files created by original Export (the original Export utility, `exp`) and writes them to the target database.

  Note: To import files that original Export created, you must use original Import. In all other cases, Oracle recommends that you use Data Pump Import instead of original Import.

  If a table to be imported does not exist on the target database, then `imp` creates and loads the table before creating any triggers, so no triggers fire.

  If a table to be imported exists on the target database, then the Import `IGNORE` parameter determines whether triggers fire during import operations. The `IGNORE` parameter specifies whether object creation errors are ignored or not, resulting in the following behavior:

  - If `IGNORE=n` (default), then `imp` does not change the table and no triggers fire.
  - If `IGNORE=y`, then `imp` loads rows into the existing table, and `INSERT` triggers created on the table fire.

  See Also:

  - *Oracle Database Utilities* for more information about the original Import utility
  - *Oracle Database Utilities* for more information about the original Export utility
  - *Oracle Database Utilities* for more information about `IGNORE`

Triggers for Publishing Events

To use a trigger to publish an event, create a trigger that:

- Has the event as its triggering event
- Invokes the appropriate subprograms in the `DBMS_AQ` package, which provides an interface to Oracle Streams Advanced Queuing (AQ)

  For information about the `DBMS_AQ` package, see *Oracle Database PL/SQL Packages and Types Reference*.

  For information about AQ, see *Oracle Streams Advanced Queuing User's Guide*.

By enabling and disabling such triggers, you can turn event notification on and off. For information about enabling and disabling triggers, see "Trigger Enabling and Disabling" on page 9-41.

How Triggers Publish Events

When the database detects an event, it fires all enabled triggers that are defined 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, if a recursive DDL statement in a system trigger modifies another trigger, then events in the same transaction cannot fire the modified trigger.

When a trigger fires and invokes AQ, AQ publishes the event and passes to the trigger the publication context and specified attributes. The trigger can access the attributes by invoking event attribute functions.

The attributes that a trigger can specify to AQ (by passing them to AQ as IN parameters) and then access with event attribute functions depends on the triggering event, which is either a database event or a client event.

Note:

- A trigger always behaves like a definer rights (DR) unit. 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. For information about DR units, see "Invoker's Rights and Definer's Rights (AUTHID Property)" on page 8-43.
- The database ignores the return status from callback functions for all events. For example, the database does nothing with the return status from a SHUTDOWN event.

Topics

- Event Attribute Functions
- Event Attribute Functions for Database Event Triggers
- Event Attribute Functions for Client Event Triggers

Event Attribute Functions

By invoking system-defined event attribute functions in Table 9–4, a trigger can retrieve certain attributes of the triggering event. Not all triggers can invoke all event attribute functions—for details, see "Event Attribute Functions for Database Event Triggers" on page 9-47 and "Event Attribute Functions for Client Event Triggers" on page 9-48.

Note:

- In earlier releases, you had to access these functions through the SYS package. Now Oracle recommends accessing them with their public synonyms (the names starting with ora_ in the first column of Table 9–4).
- The function parameter ora_name_list_t is defined in package DBMS_STANDARD as:

  ```
  TYPE ora_name_list_t IS TABLE OF VARCHAR2(64);
  ```

Table 9–4 System-Defined Event Attributes

Attribute	Return Type and Value	Example

ora_client_ip_address	VARCHAR2: IP address of client in LOGON event when underlying protocol is TCP/IP	DECLARE
		v_addr VARCHAR2(11);
		BEGIN
		IF (ora_sysevent = 'LOGON') THEN
		v_addr := **ora_client_ip_address**;
		END IF;
		END;
		/

ora_database_name	VARCHAR2(50): Database name	DECLARE
		v_db_name VARCHAR2(50);
		BEGIN
		v_db_name := **ora_database_name**;
		END;
		/

ora_des_encrypted_password	VARCHAR2: DES-encrypted password of user being created or altered	IF (ora_dict_obj_type = 'USER') THEN
		INSERT INTO event_table
		VALUES (**ora_des_encrypted_password**);
		END IF;

ora_dict_obj_name	VARCHAR2(30): Name of dictionary object on which DDL operation occurred	INSERT INTO event_table VALUES ('Changed object is ' || **ora_dict_obj_name**);

ora_dict_obj_name_list (
name_list OUT ora_name_list_t
)
	PLS_INTEGER: Number of object names modified in event
	OUT parameter: List of object names modified in event
		DECLARE
		name_list DBMS_STANDARD.ora_name_list_t;
		number_modified PLS_INTEGER;
		BEGIN
		IF (ora_sysevent='ASSOCIATE STATISTICS') THEN
		number_modified :=
		ora_dict_obj_name_list(name_list);
		END IF;
		END;

ora_dict_obj_owner	VARCHAR2(30): Owner of dictionary object on which DDL operation occurred	INSERT INTO event_table VALUES ('object owner is' || **ora_dict_obj_owner**);

ora_dict_obj_owner_list (
owner_list OUT ora_name_list_t
)
	PLS_INTEGER: Number of owners of objects modified in event
	OUT parameter: List of owners of objects modified in event
		DECLARE
		owner_list DBMS_STANDARD.ora_name_list_t;
		DBMS_STANDARD.ora_name_list_t;
		number_modified PLS_INTEGER;
		BEGIN
		IF (ora_sysevent='ASSOCIATE STATISTICS') THEN
		number_modified :=
		ora_dict_obj_name_list(owner_list);
		END IF;
		END;

ora_dict_obj_type	VARCHAR2(20): Type of dictionary object on which DDL operation occurred	INSERT INTO event_table VALUES ('This object is a ' || **ora_dict_obj_type**);

ora_grantee (
user_list OUT ora_name_list_t
)
	PLS_INTEGER: Number of grantees in grant event
	OUT parameter: List of grantees in grant event
		DECLARE
		user_list DBMS_STANDARD.ora_name_list_t;
		number_of_grantees PLS_INTEGER;
		BEGIN
		IF (ora_sysevent = 'GRANT') THEN
		number_of_grantees :=
		ora_grantee(user_list);
		END IF;
		END;

Table 9–4 (Cont.) System-Defined Event Attributes

Attribute	Return Type and Value	Example

`ora_instance_num`	`NUMBER`: Instance number	`IF (ora_instance_num = 1) THEN`
		`  INSERT INTO event_table VALUES ('1');`
		`END IF;`

`ora_is_alter_column (`
`column_name IN VARCHAR2`
`)`
	`BOOLEAN`: `TRUE` if specified column is altered, `FALSE` otherwise	`IF (ora_sysevent = 'ALTER' AND`
		`  ora_dict_obj_type = 'TABLE') THEN`
		`  alter_column := ora_is_alter_column('C');`
		`END IF;`

`ora_is_creating_nested_table`	`BOOLEAN`: `TRUE` if current event is creating nested table, `FALSE` otherwise	`IF (ora_sysevent = 'CREATE' AND`
		`  ora_dict_obj_type = 'TABLE' AND`
		`  ora_is_creating_nested_table) THEN`
		`  INSERT INTO event_table`
		`  VALUES ('A nested table is created');`
		`END IF;`

`ora_is_drop_column (`
`column_name IN VARCHAR2`
`)`
	`BOOLEAN`: `TRUE` if specified column is dropped, `FALSE` otherwise	`IF (ora_sysevent = 'ALTER' AND`
		`  ora_dict_obj_type = 'TABLE') THEN`
		`  drop_column := ora_is_drop_column('C');`
		`END IF;`

`ora_is_servererror (`
`error_number IN VARCHAR2`
`)`
	`BOOLEAN`: `TRUE` if given error is on error stack, `FALSE` otherwise	`IF ora_is_servererror(error_number) THEN`
		`  INSERT INTO event_table`
		`  VALUES ('Server error!!');`
		`END IF;`

`ora_login_user`	`VARCHAR2(30)`: Login user name	`SELECT ora_login_user FROM DUAL;`

`ora_partition_pos`	`PLS_INTEGER`: In `INSTEAD OF` trigger for `CREATE TABLE`, position in SQL text where you can insert `PARTITION` clause	`-- Retrieve ora_sql_txt into  sql_text variable`
		`v_n := ora_partition_pos;`
		`v_new_stmt := SUBSTR(sql_text,1,v_n - 1)`
		`  || ' ' || my_partition_clause`
		`  || ' ' || SUBSTR(sql_text, v_n));`

`ora_privilege_list (`
`privilege_list OUT ora_name_list_t`
`)`
	`PLS_INTEGER`: Number of privileges in grant or revoke event
	`OUT` parameter: List of privileges granted or revoked in event
		`DECLARE`
		`  privilege_list DBMS_STANDARD.ora_name_list_t;`
		`  number_of_privileges PLS_INTEGER;`
		`BEGIN`
		`  IF (ora_sysevent = 'GRANT' OR`
		`  ora_sysevent = 'REVOKE') THEN`
		`  number_of_privileges :=`
		`  ora_privilege_list(privilege_list);`
		`  END IF;`
		`END;`

`ora_revokee (`
`user_list OUT ora_name_list_t`
`)`
	`PLS_INTEGER`: Number of revokees in revoke event
	`OUT` parameter: List of revokees in event
		`DECLARE`
		`  user_list DBMS_STANDARD.ora_name_list_t;`
		`  number_of_users PLS_INTEGER;`
		`BEGIN`
		`  IF (ora_sysevent = 'REVOKE') THEN`
		`  number_of_users := ora_revokee(user_list);`
		`  END IF;`
		`END;`

`ora_server_error (`
`position IN PLS_INTEGER`
`)`
	`NUMBER`: Error code at given position on error stack[1]	`INSERT INTO event_table`
		`VALUES ('top stack error ' ||`
		`  ora_server_error(1));`

`ora_server_error_depth`	`PLS_INTEGER`: Number of error messages on error stack	`n := ora_server_error_depth;` `-- Use n with functions such as ora_server_error`

`ora_server_error_msg (`
`position IN PLS_INTEGER`
`)`
	`VARCHAR2`: Error message at given position on error stack[1]	`INSERT INTO event_table`
		`VALUES ('top stack error message' ||`
		`  ora_server_error_msg(1));`

Table 9–4 (Cont.) System-Defined Event Attributes

Attribute	Return Type and Value	Example
`ora_server_error_num_params ( position IN PLS_INTEGER )`	`PLS_INTEGER`: Number of strings substituted into error message (using format like `%s`) at given position on error stack[1]	`n := ora_server_error_num_params(1);`
`ora_server_error_param ( position IN PLS_INTEGER, param IN PLS_INTEGER )`	`VARCHAR2`: Matching substitution value (`%s`, `%d`, and so on) in error message at given position and parameter number[1]	`-- Second %s in "Expected %s, found %s":` `param := ora_server_error_param(1,2);`

`ora_sql_txt ( sql_text OUT ora_name_list_t )`	`PLS_INTEGER`: Number of elements in PL/SQL table
	`OUT` parameter: SQL text of triggering statement (broken into multiple collection elements if statement is long)
		`CREATE TABLE event_table (col VARCHAR2(2030));`
		
		`DECLARE`
		`  sql_text DBMS_STANDARD.ora_name_list_t;`
		`  n PLS_INTEGER;`
		`  v_stmt VARCHAR2(2000);`
		`BEGIN`
		`  n := ora_sql_txt(sql_text);`
		
		`  FOR i IN 1..n LOOP`
		`  v_stmt := v_stmt || sql_text(i);`
		`  END LOOP;`
		
		`  INSERT INTO event_table VALUES ('text of`
		`  triggering statement: ' || v_stmt);`
		`END;`

`ora_sysevent`	`VARCHAR2(20)`: Name of triggering event, as given in syntax	`INSERT INTO event_table` `VALUES (ora_sysevent);`

`ora_with_grant_option`	`BOOLEAN`: `TRUE` if privileges are granted with `GRANT` option, `FALSE` otherwise	`IF (ora_sysevent = 'GRANT' AND`
		`  ora_with_grant_option = TRUE) THEN`
		`  INSERT INTO event_table`
		`  VALUES ('with grant option');`
		`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`: `TRUE` if error is related to out-of-space condition, `FALSE` otherwise
	`OUT` parameters: Information about object that caused error
		`IF (space_error_info (`
		`  eno,typ,owner,ts,obj,subobj) = TRUE) THEN`
		`  DBMS_OUTPUT.PUT_LINE('The object '|| obj`
		`  || ' owned by ' || owner ||`
		`  ' has run out of space.');`
		`END IF;`

[1] Position 1 is the top of the stack.

Event Attribute Functions for Database Event Triggers

Table 9–5 summarizes the database event triggers that can invoke event attribute functions. For more information about the triggering events in Table 9–5, see "database_event" on page 14-63.

Table 9–5 Database Event Triggers

Triggering Event	When Trigger Fires	WHEN Conditions	Restrictions	Transaction	Attribute Functions

`AFTER STARTUP`	When database is opened.	None allowed	Trigger cannot do database operations.	Starts a separate transaction and commits it after firing the triggers.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`

`BEFORE SHUTDOWN`	Just before server starts shutdown of an instance.
	This lets the cartridge shutdown completely. For abnormal instance shutdown, this trigger might not fire.
		None allowed	Trigger cannot do database operations.	Starts separate transaction and commits it after firing triggers.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`

`AFTER DB_ROLE_CHANGE`	When database is opened for first time after role change.	None allowed	None	Starts separate transaction and commits it after firing triggers.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`

`AFTER SERVERERROR`	With condition, whenever specified error occurs. Without condition, whenever any error occurs.
	Trigger does not fire for errors listed in "database_event" on page 14-63.
		`ERRNO = eno`	Depends on error.	Starts separate transaction and commits it after firing triggers.	`ora_sysevent`
					`ora_login_user`
					`ora_instance_num`
					`ora_database_name`
					`ora_server_error`
					`ora_is_servererror`
					`space_error_info`

Event Attribute Functions for Client Event Triggers

Table 9–6 summarizes the client event triggers that can invoke event attribute functions. For more information about the triggering events in Table 9–6, see "ddl_event" on page 14-62 and "database_event" on page 14-63.

Note: If a client event trigger becomes the target of a DDL operation (such as `CREATE OR REPLACE TRIGGER`), then it cannot fire later during the same transaction.

Table 9–6 Client Event Triggers

Triggering Event	When Trigger Fires	WHEN Conditions	Restrictions	Transaction	Attribute Functions
BEFORE ALTER AFTER ALTER	When catalog object is altered	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_des_encrypted_password (for ALTER USER events) ora_is_alter_column (for ALTER TABLE events) ora_is_drop_column (for ALTER TABLE events)
BEFORE DROP AFTER DROP	When catalog object is dropped	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner
BEFORE ANALYZE AFTER ANALYZE	When ANALYZE statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE ASSOCIATE STATISTICS AFTER ASSOCIATE STATISTICS	When ASSOCIATE STATISTICS statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list

Table 9–6 (Cont.) Client Event Triggers

Triggering Event	When Trigger Fires	WHEN Conditions	Restrictions	Transaction	Attribute Functions
BEFORE AUDIT AFTER AUDIT BEFORE NOAUDIT AFTER NOAUDIT	When AUDIT or NOAUDIT statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name
BEFORE COMMENT AFTER COMMENT	When object is commented	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE CREATE AFTER CREATE	When catalog object is created	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_type ora_dict_obj_name ora_dict_obj_owner ora_is_creating_nested_table (for CREATE TABLE events)
BEFORE DDL 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.	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner

Table 9–6 (Cont.) Client Event Triggers

Triggering Event	When Trigger Fires	WHEN Conditions	Restrictions	Transaction	Attribute Functions
BEFORE DISASSOCIATE STATISTICS AFTER DISASSOCIATE STATISTICS	When DISASSOCIATE STATISTICS statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_dict_obj_name_list ora_dict_obj_owner_list
BEFORE GRANT AFTER GRANT	When GRANT statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_grantee ora_with_grant_option ora_privileges
BEFORE LOGOFF	At start of user logoff	Simple conditions on UID and USER	DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name
AFTER LOGON	After successful user logon	Simple conditions on UID and USER	DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Starts separate transaction and commits it after firing triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_client_ip_address
BEFORE RENAME AFTER RENAME	When RENAME statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_owner ora_dict_obj_type

Table 9–6 (Cont.) Client Event Triggers

Triggering Event	When Trigger Fires	WHEN Conditions	Restrictions	Transaction	Attribute Functions
BEFORE REVOKE AFTER REVOKE	When REVOKE statement is issued	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner ora_revokee ora_privileges
AFTER SUSPEND	After SQL statement is suspended because of out-of-space condition. (Trigger must correct condition so statement can be resumed.)	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror space_error_info
BEFORE TRUNCATE AFTER TRUNCATE	When object is truncated	Simple conditions on type and name of object, UID, and USER	Trigger cannot do DDL operations on object that caused event to be generated. DDL on other objects is limited to compiling an object, creating a trigger, and creating, altering, and dropping a table.	Fires triggers in current transaction.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner

Views for Information About Triggers

The *_TRIGGERS static data dictionary views reveal information about triggers. For information about these views, see *Oracle Database Reference*.

Example 9–27 creates a trigger and queries the static data dictionary view USER_TRIGGERS twice—first to show its type, triggering event, and the name of the table on which it is created, and then to show its body.

Note: The query results in Example 9–27 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 9–27 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;
/

COLUMN Trigger_type FORMAT A15
COLUMN Triggering_event FORMAT A16
COLUMN Table_name FORMAT A11
COLUMN Trigger_body FORMAT A50
```

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.
```

10

PL/SQL Packages

This chapter explains how to bundle related PL/SQL code and data into a package, whose contents are available to many applications.

Topics

- What is a Package?
- Reasons to Use Packages
- Package Specification
- Package Body
- Package Instantiation and Initialization
- Package State
- SERIALLY_REUSABLE Packages
- Package Writing Guidelines
- Package Example
- How STANDARD Package Defines the PL/SQL Environment

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for information about the many product-specific packages that Oracle Database supplies
- "DROP PACKAGE Statement" on page 14-93, which drops a stored package from the database

What is a Package?

A **package** is a schema object that groups logically related PL/SQL types, variables, constants, subprograms, cursors, and exceptions. A package is compiled and stored in the database, where many applications can share its contents. You can think of a package as an application.

A package always has a **specification**, which declares the **public items** that can be referenced from outside the package. You can think of the package specification as the application programming interface (API). For more information about the package specification, see "Package Specification" on page 10-3.

If the public items include cursors or subprograms, then the package must also have a **body**. The body must define queries for public cursors and code for public subprograms. The body can also declare and define **private items** that cannot be

referenced from outside the package, but are necessary for the internal workings of the package. Finally, the body can have an **initialization part**, whose statements initialize variables and do other one-time setup steps. You can change the body without changing the specification or the references to the public items; therefore, you can think of the package body as a black box. For more information about the package body, see "Package Body" on page 10-5.

In either the package specification or package body, you can map a package subprogram to an external Java or C subprogram by using a **call specification**, which maps the external subprogram name, parameter types, and return type to their SQL counterparts. For details, see "Function Declaration and Definition" on page 13-83 and "Procedure Declaration and Definition" on page 13-108.

The **`AUTHID` clause** of the package specification 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 8-43.

Reasons to Use Packages

Packages support the development and maintenance of reliable, reusable code with the following features:

- **Modularity**

  Packages let you encapsulate logically related types, variables, constants, subprograms, cursors, and exceptions in named PL/SQL modules. You can make each package easy to understand, and make the interfaces between packages simple, clear, and well defined. This practice aids application development.

- **Easier Application Design**

  When designing an application, all you need initially is the interface information in the package specifications. You can code and compile specifications without their bodies. Next, you can compile standalone stored subprograms that reference the packages. You need not fully define the package bodies until you are ready to complete the application.

- **Information Hiding**

  Packages let you share the your interface information in the package specification, and hide the implementation details in the package body. Hiding the implementation details in the body has these advantages:

  - You can change the implementation details without affecting the application interface.
  - Application users cannot develop code that depends on implementation details that you might want to change.

- **Added Functionality**

  Package public variables and cursors can persist for the life 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. (For the situations in which package public variables and cursors do not persist for the life of a session, see "Package State" on page 10-7.)

- **Better Performance**

The first time you invoke a package subprogram, Oracle Database loads the whole package into memory. Subsequent invocations of other subprograms in same the package require no disk I/O.

Packages prevent cascading dependencies and unnecessary recompiling. For example, if you change the body of a package function, Oracle Database does not recompile other subprograms that invoke the function, because these subprograms depend only on the parameters and return value that are declared in the specification.

Note: You cannot reference host variables from inside a package.

Package Specification

A **package specification** declares **public items**. The scope of a public item is the schema of the package. A public item is visible everywhere in the schema. To reference a public item that is in scope but not visible, qualify it with the package name. (For information about scope, visibility, and qualification, see "Scope and Visibility of Identifiers" on page 2-16.)

Each public item declaration has all information that you need to use the item. For example, suppose that a package specification declares the function `factorial` this way:

```
FUNCTION factorial (n INTEGER) RETURN INTEGER; -- returns n!
```

The declaration shows that `factorial` needs one argument of type `INTEGER` and returns a value of type `INTEGER`, which is all you must know to invoke `factorial`. You need not know how `factorial` is implemented (for example, whether it is iterative or recursive).

Topics

- Appropriate Public Items
- Creating Package Specifications

Appropriate Public Items

Appropriate public items are:

- Types, variables, constants, subprograms, cursors, and exceptions used by multiple subprograms

  A type defined in a package specification is either a PL/SQL user-defined subtype (described in "User-Defined PL/SQL Subtypes" on page 3-11) or a PL/SQL composite type (described in Chapter 5, "PL/SQL Collections and Records").

  Note: A PL/SQL composite type defined in a package specification is incompatible with an identically defined local or standalone stored type (see Example 5–30, Example 5–31, and Example 5–35).

- Associative array types of standalone stored subprogram parameters

  You cannot declare an associative array type at schema level. Therefore, to pass an associative array variable as a parameter to a standalone stored subprogram, you must declare the type of that variable in a package specification. Doing so makes

the type available to both the invoked subprogram (which declares a formal parameter of that type) and to the invoking subprogram or anonymous block (which declares a variable of that type). See Example 10–2.

- Variables that must remain available between subprogram invocations in the same session
- Subprograms that read and write public variables ("get" and "set" subprograms)

  Provide these subprograms to discourage package users from reading and writing public variables directly.
- Subprograms that invoke each other

  You need not worry about compilation order for package subprograms, as you must for standalone stored subprograms that invoke each other.
- Overloaded subprograms

  Overloaded subprograms are variations of the same subprogram. That is, they have the same name but different formal parameters. For more information about them, see "Overloaded Subprograms" on page 8-25.

Note: You cannot reference remote package public variables, even indirectly. For example, if a subprogram refers to a package public variable, you cannot invoke the subprogram through a database link.

Creating Package Specifications

To create a package specification, use the "CREATE PACKAGE Statement" on page 14-43.

In Example 10–1, the specification for the package `trans_data` declares two public types and three public variables.

Example 10–1 Simple Package Specification

```
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;
/
```

In Example 10–2, the specification for the package `aa_pkg` declares an associative array type, `aa_type`. Then, the standalone stored procedure `print_aa` declares a formal parameter of type `aa_type`. Next, the anonymous block declares a variable of type `aa_type`, populates it, and passes it to the procedure `print_aa`, which prints it.

Example 10–2 Passing Associative Array to Standalone Subprogram

```
CREATE OR REPLACE PACKAGE aa_pkg IS
  TYPE aa_type IS TABLE OF INTEGER INDEX BY VARCHAR2(15);
```

```
END;
/
CREATE OR REPLACE PROCEDURE print_aa (
  aa aa_pkg.aa_type
) IS
  i  VARCHAR2(15);
BEGIN
  i := aa.FIRST;

  WHILE i IS NOT NULL LOOP
    DBMS_OUTPUT.PUT_LINE (aa(i) || '  ' || i);
    i := aa.NEXT(i);
  END LOOP;
END;
/
DECLARE
  aa_var  aa_pkg.aa_type;
BEGIN
  aa_var('zero') := 0;
  aa_var('one') := 1;
  aa_var('two') := 2;
  print_aa(aa_var);
END;
/
```

Result:

```
1  one
2  two
0  zero
```

Because the package specifications in Example 10–1 and Example 10–2 do not declare cursors or subprograms, the packages `trans_data` and `aa_pkg` do not need bodies.

Package Body

If a package specification declares cursors or subprograms, then a package body is required; otherwise, it is optional. The package body and package specification must be in the same schema.

Every cursor or subprogram declaration in the package specification must have a corresponding definition in the package body. The headings of corresponding subprogram declarations and definitions must match word for word, except for white space.

To create a package body, use the "CREATE PACKAGE BODY Statement" on page 14-46.

In Example 10–3, the headings of the corresponding subprogram declaration and definition do not match word for word; therefore, PL/SQL raises an exception, even though `employees.hire_date%TYPE` is `DATE`.

Example 10–3 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 OR REPLACE 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 cursors and subprograms declared in the package specification and defined in the package body are public items that can be referenced from outside the package. The package body can also declare and define **private items** that cannot be referenced from outside the package, but are necessary for the internal workings of the package.

Finally, the body can have an **initialization part**, whose statements initialize public variables and do other one-time setup steps. The initialization part runs only the first time the package is referenced. The initialization part can include an exception handler.

You can change the package body without changing the specification or the references to the public items.

Package Instantiation and Initialization

When a session references a package item, Oracle Database instantiates the package for that session. Every session that references a package has its own instantiation of that package.

When Oracle Database instantiates a package, it initializes it. Initialization includes whichever of the following are applicable:

- Assigning initial values to public constants
- Assigning initial values to public variables whose declarations specify them
- Executing the initialization part of the package body

Package State

The values of the variables, constants, and cursors that a package declares (in either its specification or body) comprise its **package state**. If a PL/SQL package declares at least one variable, constant, or cursor, then the package is **stateful**; otherwise, it is **stateless**.

Each session that references a package item has its own instantiation of that package. If the package is stateful, the instantiation includes its state. The package state persists for the life of a session, except in these situations:

- The package is `SERIALLY_REUSABLE`.

  For details, see "SERIALLY_REUSABLE Packages" on page 10-7.

- The package body is recompiled.

  If the body of an instantiated, stateful package is recompiled (either explicitly, with the "ALTER PACKAGE Statement" on page 14-8, or implicitly), the next invocation of a subprogram in the package causes Oracle Database to discard the existing package state and raise the exception ORA-04068.

  After PL/SQL raises the exception, a reference to the package causes Oracle Database to re-instantiate the package, which re-initializes it. Therefore, previous changes to the package state are lost. (For information about initialization, see "Package Instantiation and Initialization" on page 10-6.)

- Any of the session's instantiated packages are invalidated and revalidated.

  All of a session's package instantiations (including package states) can be lost if any of the session's instantiated packages are invalidated and revalidated. For information about invalidation and revalidation of schema objects, see *Oracle Database Advanced Application Developer's Guide*.

As of Oracle Database 11*g* Release 2 (11.2.0.2), Oracle Database treats a package as stateless if its state is constant for the life of a session (or longer). This is the case for a package whose items are all compile-time constants.

A **compile-time constant** is a constant whose value the PL/SQL compiler can determine at compilation time. A constant whose initial value is a literal is always a compile-time constant. A constant whose initial value is not a literal, but which the optimizer reduces to a literal, is also a compile-time constant. Whether the PL/SQL optimizer can reduce a nonliteral expression to a literal depends on optimization level. Therefore, a package that is stateless when compiled at one optimization level might be stateful when compiled at a different optimization level. For information about the optimizer, see "PL/SQL Optimizer" on page 12-1.

SERIALLY_REUSABLE Packages

`SERIALLY_REUSABLE` packages let you design applications that manage memory better for scalability.

If a package is not SERIALLY_REUSABLE, its package state is stored in the user global area (UGA) for each user. Therefore, the amount of UGA memory needed increases linearly with the number of users, limiting scalability. The package state can persist for the life of a session, locking UGA memory until the session ends. In some applications, such as Oracle Office, a typical session lasts several days.

If a package is SERIALLY_REUSABLE, its package state is stored in a work area in a small pool in the system global area (SGA). The package state persists only for the life of a server call. After the server call, the work area returns to the pool. If a subsequent server call references the package, then Oracle Database reuses an instantiation from the pool. Reusing an instantiation re-initializes it; therefore, changes made to the package state in previous server calls are invisible. (For information about initialization, see "Package Instantiation and Initialization" on page 10-6.)

Note: Trying to access a SERIALLY_REUSABLE package from a database trigger, or from a PL/SQL subprogram invoked by a SQL statement, raises an error.

Topics

- Creating SERIALLY_REUSABLE Packages
- SERIALLY_REUSABLE Package Work Unit
- Explicit Cursors in SERIALLY_REUSABLE Packages

Creating SERIALLY_REUSABLE Packages

To create a SERIALLY_REUSABLE package, include the SERIALLY_REUSABLE pragma in the package specification and, if it exists, the package body.

Example 10–4 creates two very simple SERIALLY_REUSABLE packages, one with only a specification, and one with both a specification and a body.

See Also: "SERIALLY_REUSABLE Pragma" on page 13-130

Example 10–4 Creating SERIALLY_REUSABLE Packages

```
-- Create bodiless SERIALLY_REUSABLE package:

CREATE OR REPLACE PACKAGE bodiless_pkg IS
  PRAGMA SERIALLY_REUSABLE;
  n NUMBER := 5;
END;
/

-- Create SERIALLY_REUSABLE package with specification and body:

CREATE OR REPLACE PACKAGE pkg IS
  PRAGMA SERIALLY_REUSABLE;
  n NUMBER := 5;
END;
/

CREATE OR REPLACE PACKAGE BODY pkg IS
  PRAGMA SERIALLY_REUSABLE;
BEGIN
  n := 5;
END;
```

```
/
```

SERIALLY_REUSABLE Package Work Unit

For a SERIALLY_REUSABLE package, the work unit is a server call. You must use its public variables only within the work unit.

> **Note:** If you make a mistake and depend on the value of a public variable that was set in a previous work unit, then your program can fail. PL/SQL cannot check for such cases.

In Example 10–5, the bodiless packages pkg and pkg_sr are the same, except that pkg_sr is SERIALLY_REUSABLE and pkg is not. Each package declares public variable n with initial value 5. Then, an anonymous block changes the value of each variable to 10. Next, another anonymous block prints the value of each variable. The value of pkg.n is still 10, because the state of pkg persists for the life of the session. The value of pkg_sr.n is 5, because the state of pkg_sr persists only for the life of the server call.

Example 10–5 Effect of SERIALLY_REUSABLE Pragma

```
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
```

After the work unit (server call) of a SERIALLY_REUSABLE package completes, Oracle Database does the following:

- Closes any open cursors.
- Frees some nonreusable memory (for example, memory for collection and long VARCHAR2 variables)
- Returns the package instantiation to the pool of reusable instantiations kept for this package.

Explicit Cursors in SERIALLY_REUSABLE Packages

An explicit cursor in a SERIALLY_REUSABLE package remains open until either you close it or its work unit (server call) ends. To re-open the cursor, you must make a new server call. A server call can be different from a subprogram invocation, as Example 10–6 shows.

In contrast, an explicit cursor in a package that is not SERIALLY_REUSABLE remains open until you either close it or disconnect from the session.

Example 10–6 Cursor in SERIALLY_REUSABLE Package Open at Call Boundary

```
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 Writing Guidelines

- Become familiar with the packages that Oracle Database supplies, and avoid writing packages that duplicate their features.

  For more information about the packages that Oracle Database supplies, see *Oracle Database PL/SQL Packages and Types Reference*.

- Keep your packages general so that future applications can reuse them.
- Design and define the package specifications before the package bodies.
- In package specifications, declare only items that must be visible to invoking programs.

  This practice prevents other developers from building unsafe dependencies on your implementation details and reduces the need for recompilation.

  If you change the package specification, you must recompile any subprograms that invoke the public subprograms of the package. If you change only the package body, you do not have to recompile those subprograms.

- Declare public cursors in package specifications and define them in package bodies, as in Example 10–7.

  This practice lets you hide cursors' queries from package users and change them without changing cursor declarations.

- Assign initial values in the initialization part of the package body instead of in declarations.

  This practice has these advantages:

  – The code for computing the initial values can be more complex and better documented.

  – If computing an initial value raises an exception, the initialization part can handle it with its own exception handler.

In Example 10–7, the declaration and definition of the cursor `c1` are in the specification and body, respectively, of the package `emp_stuff`. The cursor declaration specifies only the data type of the return value, not the query, which appears in the cursor definition (for complete syntax and semantics, see "Explicit Cursor Declaration and Definition" on page 13-57).

Example 10–7 Separating Cursor Declaration and Definition in Package

```
CREATE PACKAGE emp_stuff AS
  CURSOR c1 RETURN employees%ROWTYPE;  -- Declare cursor
END emp_stuff;
/
CREATE PACKAGE BODY emp_stuff AS
  CURSOR c1 RETURN employees%ROWTYPE IS
    SELECT * FROM employees WHERE salary > 2500;  -- Define cursor
END emp_stuff;
/
```

Package Example

Example 10–8 creates a table, `log`, and a package, `emp_admin`, and then invokes package subprograms from an anonymous block. The package has both specification and body.

The specification declares a public type, cursor, and exception, and three public subprograms. One public subprogram is overloaded (for information about overloaded subprograms, see "Overloaded Subprograms" on page 8-25).

The body declares a private variable, defines the public cursor and subprograms that the specification declares, declares and defines a private function, and has an initialization part.

The initialization part (which runs only the first time the anonymous block references the package) inserts one row into the table `log` and initializes the private variable `number_hired` to zero. Every time the package procedure `hire_employee` is invoked, it updates the private variable `number_hired`.

Example 10–8 Creating emp_admin Package

```
-- Log to track changes (not part of package):

DROP TABLE log;
CREATE TABLE log (
  date_of_action  DATE,
  user_id         VARCHAR2(20),
  package_name    VARCHAR2(30)
);

-- Package specification:

CREATE OR REPLACE PACKAGE emp_admin AS
  -- Declare public type, cursor, and exception:
  TYPE EmpRecTyp IS RECORD (emp_id NUMBER, sal NUMBER);
  CURSOR desc_salary RETURN EmpRecTyp;
  invalid_salary EXCEPTION;

  -- Declare public 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;

  -- Overload preceding public 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;  -- private variable, 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 private 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  -- Invoke private function
    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 log (date_of_action, user_id, package_name)
   VALUES (SYSDATE, USER, 'EMP_ADMIN');
   number_hired := 0;
END emp_admin;
/
-- Invoke packages subprograms in anonymous block:

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 is similar to:

```
The number of employees hired is 1
The employee id is 212
The 10th highest salary is 12075, belonging to employee: 168
```

```
There are now 107 employees.
```

How STANDARD Package Defines the PL/SQL Environment

A package named STANDARD defines the PL/SQL environment. The package specification declares public types, variables, exceptions, 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 public declaration. You can still invoke the SQL function by specifying its full name:

```
abs_diff := STANDARD.ABS(x - y);
```

Most SQL 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 an invocation of TO_CHAR by matching the number and data types of the formal and actual parameters.

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**.

> **Note:** The language of warning and error messages depends on the `NLS_LANGUAGE` parameter. For information about this parameter, see *Oracle Database Globalization Support Guide*.

Topics

- Compile-Time Warnings
- Overview of Exception Handling
- Internally Defined Exceptions
- Predefined Exceptions
- User-Defined Exceptions
- Redeclared Predefined Exceptions
- Raising Exceptions Explicitly
- Exception Propagation
- Unhandled Exceptions
- Error Code and Error Message Retrieval
- Continuing Execution After Handling Exceptions
- Retrying Transactions After Handling Exceptions

See Also:

- "Exception Handling in Triggers" on page 9-33
- "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-19

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 cause errors and prevent compilation—for example, using a deprecated PL/SQL feature.

To see warnings (and errors) generated during compilation, either query the static data dictionary view `*_ERRORS` (described in *Oracle Database Reference*) or, in the SQL*Plus environment, use the command `SHOW ERRORS`.

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 you can compile the PL/SQL unit)

You can set the value of `PLSQL_WARNINGS` for:

- Your Oracle database instance

  Use the `ALTER SYSTEM` statement, described in *Oracle Database SQL Language Reference*.

- Your session

  Use the `ALTER SESSION` statement, described in *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 1-10.

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, described in *Oracle Database Reference*.

DBMS_WARNING Package

If you are writing PL/SQL units in a development environment that compiles them (such as SQL*Plus), you can display and set the value of PLSQL_WARNINGS by invoking subprograms in the DBMS_WARNING package.

Example 11–2 uses an ALTER SESSION statement to disable all warning messages for the session and then compiles a procedure that has unreachable code. The procedure compiles without warnings. Next, the example enables all warnings for the session by invoking DBMS_WARNING.set_warning_setting_string and displays the value of PLSQL_WARNINGS by invoking DBMS_WARNING.get_warning_setting_string. Finally, the example recompiles the procedure, and the compiler generates a warning about the unreachable code.

Note: Unreachable code could represent a mistake or be intentionally hidden by a debug flag.

Example 11–2 Displaying and Setting PLSQL_WARNINGS with DBMS_WARNING Subprograms

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 AUTHID DEFINER 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
-------- -----------------------------------------------------------------
7/5      PLW-06002: Unreachable code
```

`DBMS_WARNING` subprograms are useful when you are compiling a complex application composed of several nested SQL*Plus scripts, where different subprograms need different `PLSQL_WARNINGS` settings. With `DBMS_WARNING` subprograms, you can save the current `PLSQL_WARNINGS` setting, change the setting to compile a particular set of subprograms, and then restore the setting to its original value.

> **See Also:** *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_WARNING` package

Overview of Exception Handling

Exceptions (PL/SQL run-time errors) can arise from design faults, coding mistakes, hardware failures, and many other sources. You cannot anticipate all possible exceptions, but you can write exception handlers that let your program to continue to operate in their presence.

Any PL/SQL block can have an exception-handling part, which can have one or more exception handlers. For example, an exception-handling part could have this syntax:

```
EXCEPTION
  WHEN ex_name_1 THEN statements_1                -- Exception handler
  WHEN ex_name_2 OR ex_name_3 THEN statements_2  -- Exception handler
  WHEN OTHERS THEN statements_3                     -- Exception handler
END;
```

In the preceding syntax example, *ex_name*_n is the name of an exception and *statements*_n is one or more statements. (For complete syntax and semantics, see "Exception Handler" on page 13-50.)

When an exception is raised in the executable part of the block, the executable part stops and control transfers to the exception-handling part. If *ex_name_1* was raised, then *statements_1* run. If either *ex_name_2* or *ex_name_3* was raised, then *statements_2* run. If any other exception was raised, then *statements_3* run.

After an exception handler runs, control transfers to the next statement of the enclosing block. If there is no enclosing block, then:

- If the exception handler is in a subprogram, then control returns to the invoker, at the statement after the invocation.
- If the exception handler is in an anonymous block, then control transfers to the host environment (for example, SQL*Plus)

If an exception is raised in a block that has no exception handler for it, then the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a block has a handler for it or there is no enclosing block (for more information, see "Exception Propagation" on page 11-17). If there is no handler for the exception, then PL/SQL returns an unhandled exception error to the invoker or host environment, which determines the outcome (for more information, see "Unhandled Exceptions" on page 11-24).

Topics

- Exception Categories
- Advantages of Exception Handlers
- Guidelines for Avoiding and Handling Exceptions

Exception Categories

The exception categories are:

- **Internally defined**

  The run-time system raises internally defined exceptions implicitly (automatically). Examples of internally defined exceptions are ORA-00060 (deadlock detected while waiting for resource) and ORA-27102 (out of memory).

  An internally defined exception always has an error code, but does not have a name unless PL/SQL gives it one or you give it one.

  For more information, see "Internally Defined Exceptions" on page 11-9.

- **Predefined**

  A predefined exception is an internally defined exception that PL/SQL has given a name. For example, ORA-06500 (PL/SQL: storage error) has the predefined name `STORAGE_ERROR`.

For more information, see "Predefined Exceptions" on page 11-10.

- **User-defined**

  You can declare your own exceptions in the declarative part of any PL/SQL anonymous block, subprogram, or package. For example, you might declare an exception named `insufficient_funds` to flag overdrawn bank accounts.

  You must raise user-defined exceptions explicitly.

  For more information, see "User-Defined Exceptions" on page 11-11.

Table 11–2 summarizes the exception categories.

Table 11–2 Exception Categories

Category	Definer	Has Error Code	Has Name	Raised Implicitly	Raised Explicitly
Internally defined	Run-time system	Always	Only if you assign one	Yes	Optionally[1]
Predefined	Run-time system	Always	Always	Yes	Optionally[1]
User-defined	User	Only if you assign one	Always	No	Always

[1] For details, see "Raising Internally Defined Exception with RAISE Statement" on page 11-14.

For a named exception, you can write a specific exception handler, instead of handling it with an `OTHERS` exception handler. A specific exception handler is more efficient than an `OTHERS` exception handler, because the latter must invoke a function to determine which exception it is handling. For details, see "Error Code and Error Message Retrieval" on page 11-24.

Advantages of Exception Handlers

Using exception handlers for error-handling makes programs easier to write and understand, and reduces the likelihood of unhandled exceptions.

Without exception handlers, you must check for every possible error, everywhere that it might occur, and then handle it. It is easy to overlook a possible error or a place where it might occur, especially if the error is not immediately detectable (for example, bad data might be undetectable until you use it in a calculation). Error-handling code is scattered throughout the program.

With exception handlers, you need not know every possible error or everywhere that it might occur. You need only include an exception-handling part in each block where errors might occur. In the exception-handling part, you can include exception handlers for both specific and unknown errors. If an error occurs anywhere in the block (including inside a sub-block), then an exception handler handles it. Error-handling code is isolated in the exception-handling parts of the blocks.

In Example 11–3, a procedure uses a single exception handler to handle the predefined exception `NO_DATA_FOUND`, which can occur in either of two `SELECT INTO` statements.

Example 11–3 Single Exception Handler for Multiple Exceptions

```
CREATE OR REPLACE PROCEDURE select_item (
  t_column VARCHAR2,
  t_name   VARCHAR2
```

```
) AUTHID DEFINER
IS
  temp VARCHAR2(30);
BEGIN
  temp := t_column;  -- For error message if next SELECT fails

  -- Fails if table t_name does not have column t_column:

  SELECT COLUMN_NAME INTO temp
  FROM USER_TAB_COLS
  WHERE TABLE_NAME = UPPER(t_name)
  AND COLUMN_NAME = UPPER(t_column);

  temp := t_name;  -- For error message if next SELECT fails

  -- Fails if there is no table named t_name:

  SELECT OBJECT_NAME INTO temp
  FROM USER_OBJECTS
  WHERE OBJECT_NAME = UPPER(t_name)
  AND OBJECT_TYPE = 'TABLE';

EXCEPTION
  WHEN NO_DATA_FOUND THEN
    DBMS_OUTPUT.PUT_LINE ('No Data found for SELECT on ' || temp);
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE ('Unexpected error');
    RAISE;
END;
/
```

Invoke procedure (there is a `DEPARTMENTS` table, but it does not have a `LAST_NAME` column):

```
BEGIN
  select_item('departments', 'last_name');
END;
/
```

Result:

```
No Data found for SELECT on departments
```

Invoke procedure (there is no `EMP` table):

```
BEGIN
  select_item('emp', 'last_name');
END;
/
```

Result:

```
No Data found for SELECT on emp
```

If multiple statements use the same exception handler, and you want to know which statement failed, you can use locator variables, as in Example 11–4.

Example 11–4 Locator Variables for Statements that Share Exception Handler

```
CREATE OR REPLACE PROCEDURE loc_var AUTHID DEFINER IS
  stmt_no  POSITIVE;
  name_    VARCHAR2(100);
```

```
BEGIN
  stmt_no := 1;

  SELECT table_name INTO name_
  FROM user_tables
  WHERE table_name LIKE 'ABC%';

  stmt_no := 2;

  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
```

You determine the precision of your error-handling code. You can have a single exception handler for all division-by-zero errors, bad array subscripts, and so on. You can also check for errors in a single statement by putting that statement inside a block with its own exception handler.

Guidelines for Avoiding and Handling Exceptions

To make your programs as reliable and safe as possible:

- Use both error-checking code and exception handlers.

  Use error-checking code wherever bad input data can cause an error. Examples of bad input data are incorrect or null actual parameters and queries that return no rows or more rows than you expect. Test your code with different combinations of bad input data to see what potential errors arise.

  Sometimes you can use error-checking code to avoid raising an exception, as in Example 11–7.

- Add exception handlers wherever errors can occur.

  Errors are especially likely during arithmetic calculations, string manipulation, and database operations. Errors can also arise from problems that are independent of your code—for example, disk storage or memory hardware failure—but your code still must take corrective action.

- Design your programs to work when the database is not in the state you expect.

  For example, a table you query might have columns added or deleted, or their types might have changed. You can avoid problems by declaring scalar variables with `%TYPE` qualifiers and record variables to hold query results with `%ROWTYPE` qualifiers.

- Whenever possible, write exception handlers for named exceptions instead of using `OTHERS` exception handlers.

  Learn the names and causes of the predefined exceptions. If you know that your database operations might raise specific internally defined exceptions that do not

have names, then give them names so that you can write exception handlers specifically for them.

- Have your exception handlers output debugging information.

  If you store the debugging information in a separate table, do it with an autonomous routine, so that you can commit your debugging information even if you roll back the work that the main subprogram did. For information about autonomous routines, see "AUTONOMOUS_TRANSACTION Pragma" on page 13-6.

- For each exception handler, carefully decide whether to have it commit the transaction, roll it back, or let it continue.

  Regardless of the severity of the error, you want to leave the database in a consistent state and avoid storing bad data.

- Avoid unhandled exceptions by including an `OTHERS` exception handler at the top level of every PL/SQL program.

Internally Defined Exceptions

Internally defined exceptions (ORA-*n* errors) are described in *Oracle Database Error Messages*. The run-time system raises them implicitly (automatically).

An internally defined exception does not have a name unless either PL/SQL gives it one (see "Predefined Exceptions" on page 11-10) or you give it one.

If you know that your database operations might raise specific internally defined exceptions that do not have names, then give them names so that you can write exception handlers specifically for them. Otherwise, you can handle them only with `OTHERS` exception handlers.

To give a name to an internally defined exception, do the following in the declarative part of the appropriate anonymous block, subprogram, or package. (To determine the appropriate block, see "Exception Propagation" on page 11-17.)

1. Declare the name.

   An exception name declaration has this syntax:

   ```
   exception_name EXCEPTION;
   ```

   For semantic information, see "Exception Declaration" on page 13-48.

2. Associate the name with the error code of the internally defined exception.

   The syntax is:

   ```
   PRAGMA EXCEPTION_INIT (exception_name, error_code)
   ```

   For semantic information, see "EXCEPTION_INIT Pragma" on page 13-46.

   > **Note:** An internally defined exception with a user-declared name is still an internally defined exception, not a user-defined exception.

Example 11–5 gives the name `deadlock_detected` to the internally defined exception ORA-00060 (deadlock detected while waiting for resource) and uses the name in an exception handler.

Example 11–5 Naming Internally Defined Exception

```
DECLARE
  deadlock_detected EXCEPTION;
  PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
  ...
EXCEPTION
  WHEN deadlock_detected THEN
    ...
END;
/
```

See Also: "Raising Internally Defined Exception with RAISE Statement" on page 11-14

Predefined Exceptions

Predefined exceptions are internally defined exceptions that have predefined names, which PL/SQL declares globally in the package STANDARD. The run-time system raises predefined exceptions implicitly (automatically). Because predefined exceptions have names, you can write exception handlers specifically for them.

Table 11–3 lists the names and error codes of the predefined exceptions.

Table 11–3 PL/SQL Predefined Exceptions

Exception Name	Error Code
ACCESS_INTO_NULL	-6530
CASE_NOT_FOUND	-6592
COLLECTION_IS_NULL	-6531
CURSOR_ALREADY_OPEN	-6511
DUP_VAL_ON_INDEX	-1
INVALID_CURSOR	-1001
INVALID_NUMBER	-1722
LOGIN_DENIED	-1017
NO_DATA_FOUND	+100
NO_DATA_NEEDED	-6548
NOT_LOGGED_ON	-1012
PROGRAM_ERROR	-6501
ROWTYPE_MISMATCH	-6504
SELF_IS_NULL	-30625
STORAGE_ERROR	-6500
SUBSCRIPT_BEYOND_COUNT	-6533
SUBSCRIPT_OUTSIDE_LIMIT	-6532
SYS_INVALID_ROWID	-1410
TIMEOUT_ON_RESOURCE	-51
TOO_MANY_ROWS	-1422
VALUE_ERROR	-6502

Table 11–3 (Cont.) PL/SQL Predefined Exceptions

Exception Name	Error Code
ZERO_DIVIDE	-1476

Example 11–6 calculates a price-to-earnings ratio for a company. If the company has zero earnings, the division operation raises the predefined exception ZERO_DIVIDE and the executable part of the block transfers control to the exception-handling part.

Example 11–6 Anonymous Block Handles ZERO_DIVIDE

```
DECLARE
  stock_price   NUMBER := 9.73;
  net_earnings  NUMBER := 0;
  pe_ratio      NUMBER;
BEGIN
  pe_ratio := stock_price / net_earnings;  -- raises ZERO_DIVIDE exception
  DBMS_OUTPUT.PUT_LINE('Price/earnings ratio = ' || pe_ratio);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Company had zero earnings.');
    pe_ratio := NULL;
END;
/
```

Result:

```
Company had zero earnings.
```

Example 11–7 uses error-checking code to avoid the exception that Example 11–6 handles.

Example 11–7 Anonymous Block Avoids ZERO_DIVIDE

```
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;
/
```

See Also: "Raising Internally Defined Exception with RAISE Statement" on page 11-14

User-Defined Exceptions

You can declare your own exceptions in the declarative part of any PL/SQL anonymous block, subprogram, or package.

An exception name declaration has this syntax:

```
exception_name EXCEPTION;
```

For semantic information, see "Exception Declaration" on page 13-48.

You must raise a user-defined exception explicitly. For details, see "Raising Exceptions Explicitly" on page 11-13.

Redeclared Predefined Exceptions

Oracle recommends against redeclaring predefined exceptions—that is, declaring a user-defined exception name that is a predefined exception name. (For a list of predefined exception names, see Table 11–3.)

If you redeclare a predefined exception, your local declaration overrides the global declaration in package STANDARD. Exception handlers written for the globally declared exception become unable to handle it—unless you qualify its name with the package name STANDARD.

Example 11–8 shows this.

Example 11–8 Redeclared Predefined Identifier

```
DROP TABLE t;
CREATE TABLE t (c NUMBER);
```

In the following block, the INSERT statement implicitly raises the predefined exception INVALID_NUMBER, which the exception handler handles.

```
DECLARE
  default_number NUMBER := 0;
BEGIN
  INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));
EXCEPTION
  WHEN INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
    INSERT INTO t VALUES(default_number);
END;
/
```

Result:

```
Substituting default value for invalid number.
```

The following block redeclares the predefined exception INVALID_NUMBER. When the INSERT statement implicitly raises the predefined exception INVALID_NUMBER, the exception handler does not handle it.

```
DECLARE
  default_number NUMBER := 0;
  i NUMBER := 5;
  invalid_number EXCEPTION;    -- redeclare predefined exception
BEGIN
  INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));
EXCEPTION
  WHEN INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
    INSERT INTO t VALUES(default_number);
END;
/
```

Result:

```
DECLARE
*
ERROR at line 1:
```

```
ORA-01722: invalid number
ORA-06512: at line 6
```

The exception handler in the preceding block handles the predefined exception INVALID_NUMBER if you qualify the exception name in the exception handler:

```
DECLARE
  default_number NUMBER := 0;
  i NUMBER := 5;
  invalid_number EXCEPTION;    -- redeclare predefined exception
BEGIN
  INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));
EXCEPTION
  WHEN STANDARD.INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
    INSERT INTO t VALUES(default_number);
END;
/
```

Result:

```
Substituting default value for invalid number.
```

Raising Exceptions Explicitly

To raise an exception explicitly, use either the RAISE statement or RAISE_APPLICATION_ERROR procedure.

Topics

- RAISE Statement
- RAISE_APPLICATION_ERROR Procedure

RAISE Statement

The RAISE statement explicitly raises an exception. Outside an exception handler, you must specify the exception name. Inside an exception handler, if you omit the exception name, the RAISE statement reraises the current exception.

Topics

- Raising User-Defined Exception with RAISE Statement
- Raising Internally Defined Exception with RAISE Statement
- Reraising Current Exception with RAISE Statement

Raising User-Defined Exception with RAISE Statement

In Example 11–9, the procedure declares an exception named past_due, raises it explicitly with the RAISE statement, and handles it with an exception handler.

Example 11–9 Declaring, Raising, and Handling User-Defined Exception

```
CREATE PROCEDURE account_status (
  due_date DATE,
  today    DATE
) AUTHID DEFINER
IS
  past_due  EXCEPTION;  -- declare exception
BEGIN
```

```
  IF due_date < today THEN
    RAISE past_due;  -- explicitly raise exception
  END IF;
EXCEPTION
  WHEN past_due THEN  -- handle exception
    DBMS_OUTPUT.PUT_LINE ('Account past due.');
END;
/

BEGIN
  account_status ('1-JUL-10', '9-JUL-10');
END;
/
```

Result:

```
Account past due.
```

Raising Internally Defined Exception with RAISE Statement

Although the run-time system raises internally defined exceptions implicitly, you can raise them explicitly with the `RAISE` statement if they have names. Table 11–3 lists the internally defined exceptions that have predefined names. "Internally Defined Exceptions" on page 11-9 explains how to give user-declared names to internally defined exceptions.

An exception handler for a named internally defined exception handles that exception whether it is raised implicitly or explicitly.

In Example 11–10, the procedure raises the predefined exception `INVALID_NUMBER` either explicitly or implicitly, and the `INVALID_NUMBER` exception handler handles always it.

Example 11–10 Explicitly Raising Predefined Exception

```
DROP TABLE t;
CREATE TABLE t (c NUMBER);

CREATE PROCEDURE p (n NUMBER) AUTHID DEFINER IS
  default_number NUMBER := 0;
BEGIN
  IF n < 0 THEN
    RAISE INVALID_NUMBER;  -- raise explicitly
  ELSE
    INSERT INTO t VALUES(TO_NUMBER('100.00', '9G999'));  -- raise implicitly
  END IF;
EXCEPTION
  WHEN INVALID_NUMBER THEN
    DBMS_OUTPUT.PUT_LINE('Substituting default value for invalid number.');
    INSERT INTO t VALUES(default_number);
END;
/

BEGIN
  p(-1);
END;
/
```

Result:

```
Substituting default value for invalid number.
```

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

Result:

```
Substituting default value for invalid number.
```

Reraising Current Exception with RAISE Statement

In an exception handler, you can use the `RAISE` statement to"reraise" the exception being handled. Reraising the exception passes it to the enclosing block, which can handle it further. (If the enclosing block cannot handle the reraised exception, then the exception propagates—see "Exception Propagation" on page 11-17.) When reraising the current exception, you need not specify an exception name.

In Example 11–11, the handling of the exception starts in the inner block and finishes in the outer block. The outer block declares the exception, so the exception name exists in both blocks, and each block has an exception handler specifically for that exception. The inner block raises the exception, and its exception handler does the initial handling and then reraises the exception, passing it to the outer block for further handling.

Example 11–11 Reraising Exception

```
DECLARE
  salary_too_high   EXCEPTION;
  current_salary    NUMBER := 20000;
  max_salary        NUMBER := 10000;
  erroneous_salary  NUMBER;
BEGIN

  BEGIN
    IF current_salary > max_salary THEN
      RAISE salary_too_high;   -- raise exception
    END IF;
  EXCEPTION
    WHEN salary_too_high THEN  -- start handling exception
      erroneous_salary := current_salary;
      DBMS_OUTPUT.PUT_LINE('Salary ' || erroneous_salary ||' is out of range.');
      DBMS_OUTPUT.PUT_LINE ('Maximum salary is ' || max_salary || '.');
      RAISE;  -- reraise current exception (exception name is optional)
  END;

EXCEPTION
  WHEN salary_too_high THEN    -- finish handling exception
    current_salary := max_salary;

    DBMS_OUTPUT.PUT_LINE (
      'Revising salary from ' || erroneous_salary ||
      ' to ' || current_salary || '.'
    );
END;
/
```

Result:

```
Salary 20000 is out of range.
Maximum salary is 10000.
```

```
Revising salary from 20000 to 10000.
```

RAISE_APPLICATION_ERROR Procedure

You can invoke the RAISE_APPLICATION_ERROR procedure (defined in the DBMS_STANDARD package) only from a stored subprogram or method. Typically, you invoke this procedure to raise a user-defined exception and return its error code and error message to the invoker.

To invoke RAISE_APPLICATION_ERROR, use this syntax:

```
RAISE_APPLICATION_ERROR (error_code, message[, {TRUE | FALSE}]);
```

You must have assigned *error_code* to the user-defined exception with the EXCEPTION_INIT pragma. The syntax is:

```
PRAGMA EXCEPTION_INIT (exception_name, error_code)
```

For semantic information, see "EXCEPTION_INIT Pragma" on page 13-46.

The *message* is a character string of at most 2048 bytes.

If you specify TRUE, PL/SQL puts *error_code* on top of the error stack. Otherwise, PL/SQL replaces the error stack with *error_code*.

In Example 11–12, an anonymous block declares an exception named past_due, assigns the error code -20000 to it, and invokes a stored procedure. The stored procedure invokes the RAISE_APPLICATION_ERROR procedure with the error code -20000 and a message, whereupon control returns to the anonymous block, which handles the exception. To retrieve the message associated with the exception, the exception handler in the anonymous block invokes the SQLERRM function, described in "Error Code and Error Message Retrieval" on page 11-24.

Example 11–12 Raising User-Defined Exception with RAISE_APPLICATION_ERROR

```
CREATE PROCEDURE account_status (
  due_date DATE,
  today    DATE
) AUTHID DEFINER
IS
BEGIN
  IF due_date < today THEN                  -- explicitly raise exception
    RAISE_APPLICATION_ERROR(-20000, 'Account past due.');
  END IF;
END;
/

DECLARE
  past_due  EXCEPTION;                      -- declare exception
  PRAGMA EXCEPTION_INIT (past_due, -20000);  -- assign error code to exception
BEGIN
  account_status ('1-JUL-10', '9-JUL-10');   -- invoke procedure
EXCEPTION
  WHEN past_due THEN                        -- handle exception
    DBMS_OUTPUT.PUT_LINE(TO_CHAR(SQLERRM(-20000)));
END;
/
```

Result:

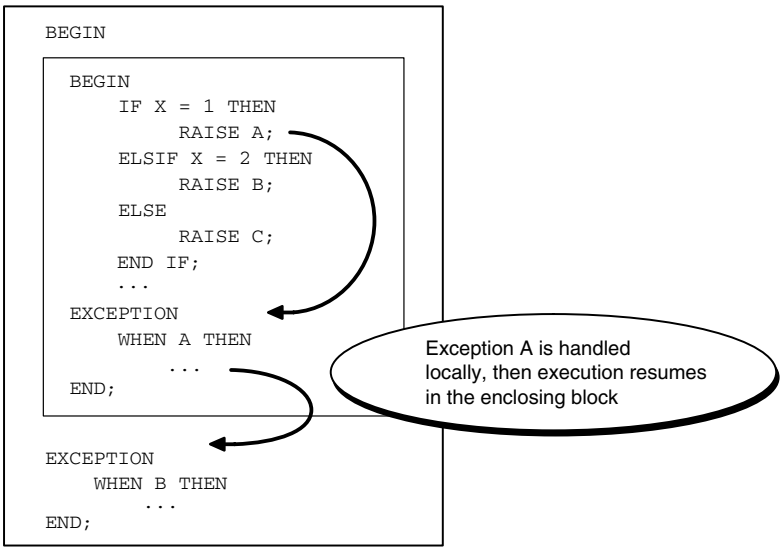
```
ORA-20000: Account past due.
```

Exception Propagation

If an exception is raised in a block that has no exception handler for it, then the exception **propagates**. That is, the exception reproduces itself in successive enclosing blocks until either a block has a handler for it or there is no enclosing block. If there is no handler for the exception, then PL/SQL returns an unhandled exception error to the invoker or host environment, which determines the outcome (for more information, see "Unhandled Exceptions" on page 11-24).

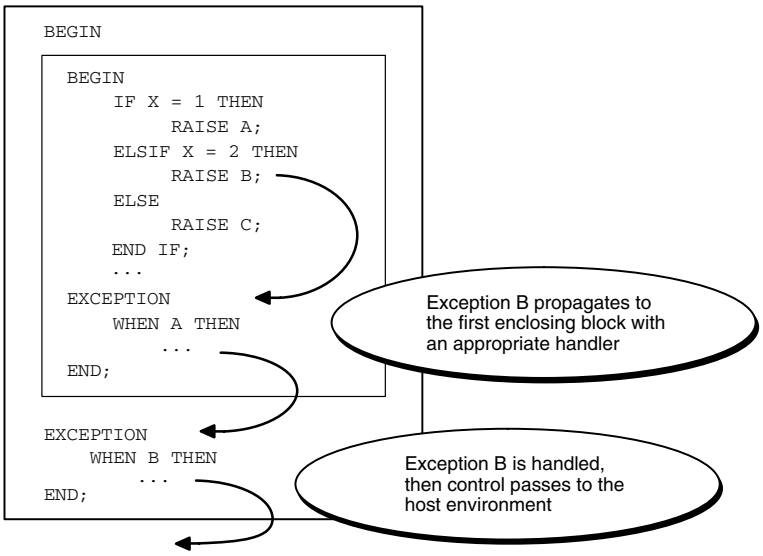
In Figure 11–1, one block is nested inside another. The inner block raises exception A. The inner block has an exception handler for A, so A does not propagate. After the exception handler runs, control transfers to the next statement of the outer block.

Figure 11–1 Exception Does Not Propagate



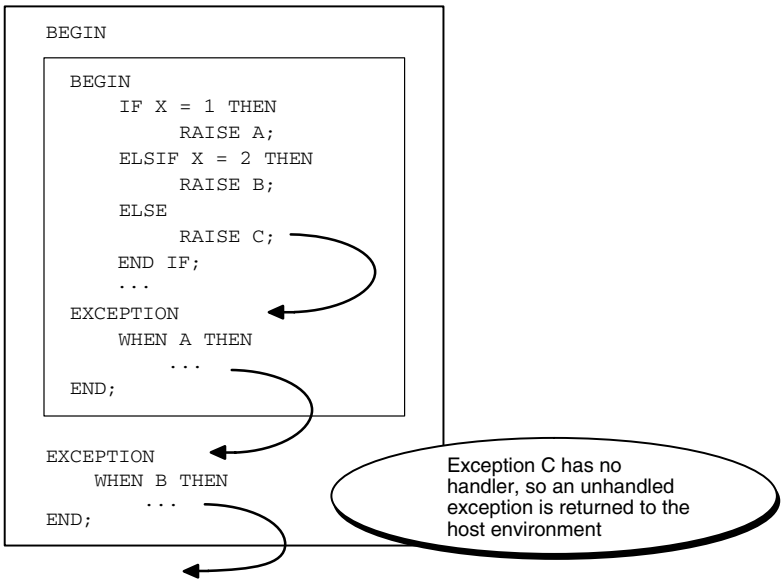
In Figure 11–2, the inner block raises exception B. The inner block does not have an exception handler for exception B, so B propagates to the outer block, which does have an exception handler for it. After the exception handler runs, control transfers to the host environment.

Figure 11–2 Exception Propagates from Inner Block to Outer Block



In Figure 11–3, the inner block raises exception C. The inner block does not have an exception handler for C, so exception C propagates to the outer block. The outer block does not have an exception handler for C, so PL/SQL returns an unhandled exception error to the host environment.

Figure 11–3 PL/SQL Returns Unhandled Exception Error to Host Environment



A user-defined exception can propagate beyond its scope (that is, beyond the block that declares it), but its name does not exist beyond its scope. Therefore, beyond its scope, a user-defined exception can be handled only with an `OTHERS` exception handler.

In Example 11–13, the inner block declares an exception named `past_due`, for which it has no exception handler. When the inner block raises `past_due`, the exception propagates to the outer block, where the name `past_due` does not exist. The outer block handles the exception with an `OTHERS` exception handler.

Example 11–13 Exception that Propagates Beyond Scope is Handled

```
BEGIN

  DECLARE
    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;

EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
/
```

If the outer block does not handle the user-defined exception, then an error occurs, as in Example 11–14.

Example 11–14 Exception that Propagates Beyond Scope is Not Handled

```
BEGIN

  DECLARE
    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;

END;
/
```

Result:

```
BEGIN
*
ERROR at line 1:
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 9
```

Note: Exceptions cannot propagate across remote subprogram invocations. Therefore, a PL/SQL block cannot handle an exception raised by a remote subprogram.

Topics

- Propagation of Exceptions Raised in Declarations
- Propagation of Exceptions Raised in Exception Handlers

Propagation of Exceptions Raised in Declarations

An exception raised in a declaration propagates immediately to the enclosing block (or to the invoker or host environment if there is no enclosing block). Therefore, the exception handler must be in an enclosing or invoking block, not in the same block as the declaration.

In Example 11–15, the VALUE_ERROR exception handler is in the same block as the declaration that raises VALUE_ERROR. Because the exception propagates immediately to the host environment, the exception handler does not handle it.

Example 11–15 Exception Raised in Declaration is Not Handled

```
DECLARE
  credit_limit CONSTANT NUMBER(3) := 5000;  -- Maximum value is 999
BEGIN
  NULL;
EXCEPTION
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('Exception raised in 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
```

Example 11–16 is like Example 11–15 except that an enclosing block handles the VALUE_ERROR exception that the declaration in the inner block raises.

Example 11–16 Exception Raised in Declaration is Handled by Enclosing Block

```
BEGIN

  DECLARE
    credit_limit CONSTANT NUMBER(3) := 5000;
  BEGIN
    NULL;
  END;

EXCEPTION
  WHEN VALUE_ERROR THEN
    DBMS_OUTPUT.PUT_LINE('Exception raised in declaration.');
END;
/
```

Result:

```
Exception raised in declaration.
```

Propagation of Exceptions Raised in Exception Handlers

An exception raised in an exception handler propagates immediately to the enclosing block (or to the invoker or host environment if there is no enclosing block). Therefore, the exception handler must be in an enclosing or invoking block.

In Example 11–17, when n is zero, the calculation 1/n raises the predefined exception ZERO_DIVIDE, and control transfers to the ZERO_DIVIDE exception handler in the same block. When the exception hander raises ZERO_DIVIDE, the exception propagates immediately to the invoker. The invoker does not handle the exception, so PL/SQL returns an unhandled exception error to the host environment.

Example 11–17 Exception Raised in Exception Handler is Not Handled

```
CREATE PROCEDURE print_reciprocal (n NUMBER) AUTHID DEFINER IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(1/n);  -- handled
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error:');
    DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined');  -- not handled
END;
/

BEGIN  -- invoking block
  print_reciprocal(0);
END;
```

Result:

```
Error:
BEGIN
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at "HR.PRINT_RECIPROCAL", line 7
ORA-01476: divisor is equal to zero
ORA-06512: at line 2
```

Example 11–18 is like Example 11–17 except that when the procedure returns an unhandled exception error to the invoker, the invoker handles it.

Example 11–18 Exception Raised in Exception Handler is Handled by Invoker

```
CREATE PROCEDURE print_reciprocal (n NUMBER) AUTHID DEFINER IS
BEGIN
  DBMS_OUTPUT.PUT_LINE(1/n);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error:');
    DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined');
END;
/

BEGIN  -- invoking block
  print_reciprocal(0);
EXCEPTION
  WHEN ZERO_DIVIDE THEN  -- handles exception raised in exception handler
    DBMS_OUTPUT.PUT_LINE('1/0 is undefined.');
END;
/
```

Result:

```
Error:
1/0 is undefined.
```

Example 11–19 is like Example 11–17 except that an enclosing block handles the exception that the exception handler in the inner block raises.

Example 11–19 Exception Raised in Exception Handler is Handled by Enclosing Block

```
CREATE PROCEDURE print_reciprocal (n NUMBER) AUTHID DEFINER IS
BEGIN

  BEGIN
    DBMS_OUTPUT.PUT_LINE(1/n);
  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Error in inner block:');
      DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined.');
  END;

EXCEPTION
  WHEN ZERO_DIVIDE THEN  -- handles exception raised in exception handler
    DBMS_OUTPUT.PUT('Error in outer block: ');
    DBMS_OUTPUT.PUT_LINE('1/0 is undefined.');
END;
/

BEGIN
  print_reciprocal(0);
END;
/
```

Result:

```
Error in inner block:
Error in outer block: 1/0 is undefined.
```

In Example 11–20, the exception-handling part of the procedure has exception handlers for user-defined exception `i_is_one` and predefined exception `ZERO_DIVIDE`. When the `i_is_one` exception handler raises `ZERO_DIVIDE`, the exception propagates immediately to the invoker (therefore, the `ZERO_DIVIDE` exception handler does not handle it). The invoker does not handle the exception, so PL/SQL returns an unhandled exception error to the host environment.

Example 11–20 Exception Raised in Exception Handler is Not Handled

```
CREATE PROCEDURE descending_reciprocals (n INTEGER) AUTHID DEFINER IS
  i INTEGER;
  i_is_one EXCEPTION;
BEGIN
  i := n;

  LOOP
    IF i = 1 THEN
      RAISE i_is_one;
    ELSE
      DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || i || ' is ' || 1/i);
    END IF;

    i := i - 1;
  END LOOP;
EXCEPTION
  WHEN i_is_one THEN
    DBMS_OUTPUT.PUT_LINE('1 is its own reciprocal.');
```

```
    DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || TO_CHAR(i-1) ||
                         ' is ' || TO_CHAR(1/(i-1)));

  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Error:');
    DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined');
END;
/

BEGIN
  descending_reciprocals(3);
END;
/
```

Result:

```
Reciprocal of 3 is .3333333333333333333333333333333333333333
Reciprocal of 2 is .5
1 is its own reciprocal.
BEGIN
*
ERROR at line 1:
ORA-01476: divisor is equal to zero
ORA-06512: at "HR.DESCENDING_RECIPROCALS", line 19
ORA-06510: PL/SQL: unhandled user-defined exception
ORA-06512: at line 2
```

Example 11–21 is like Example 11–20 except that an enclosing block handles the `ZERO_DIVIDE` exception that the `i_is_one` exception handler raises.

Example 11–21 Exception Raised in Exception Handler is Handled by Enclosing Block

```
CREATE PROCEDURE descending_reciprocals (n INTEGER) AUTHID DEFINER IS
  i INTEGER;
  i_is_one EXCEPTION;
BEGIN

  BEGIN
    i := n;

    LOOP
      IF i = 1 THEN
        RAISE i_is_one;
      ELSE
        DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || i || ' is ' || 1/i);
      END IF;

      i := i - 1;
    END LOOP;
  EXCEPTION
    WHEN i_is_one THEN
      DBMS_OUTPUT.PUT_LINE('1 is its own reciprocal.');
      DBMS_OUTPUT.PUT_LINE('Reciprocal of ' || TO_CHAR(i-1) ||
                           ' is ' || TO_CHAR(1/(i-1)));

    WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Error:');
      DBMS_OUTPUT.PUT_LINE(1/n || ' is undefined');
  END;

EXCEPTION
```

```
  WHEN ZERO_DIVIDE THEN  -- handles exception raised in exception handler
    DBMS_OUTPUT.PUT_LINE('Error:');
    DBMS_OUTPUT.PUT_LINE('1/0 is undefined');
END;
/

BEGIN
  descending_reciprocals(3);
END;
/
```

Result:

```
Reciprocal of 3 is .3333333333333333333333333333333333333333
Reciprocal of 2 is .5
1 is its own reciprocal.
Error:
1/0 is undefined
```

Unhandled Exceptions

If there is no handler for a raised exception, PL/SQL returns an unhandled exception error to the invoker or host environment, which determines the outcome.

If a subprogram exits with an unhandled exception, then actual parameters for `OUT` and `IN OUT` formal parameters passed by value (the default) retain the values that they had before the subprogram invocation (see Example 8–13).

If a stored subprogram exits with an unhandled exception, PL/SQL does not roll back database changes made by the subprogram.

The `FORALL` statement runs one DML statement multiple times, with different values in the `VALUES` and `WHERE` clauses. If one set of values raises an unhandled exception, then PL/SQL rolls back all database changes made earlier in the `FORALL` statement. For more information, see "Handling FORALL Exceptions Immediately" on page 12-18 and "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-19.

> **Tip:** Avoid unhandled exceptions by including an `OTHERS` exception handler at the top level of every PL/SQL program.

Error Code and Error Message Retrieval

In an exception handler, for the exception being handled:

- You can retrieve the error code with the PL/SQL function `SQLCODE`, described in "SQLCODE Function" on page 13-131.
- You can retrieve the error message with either:
  - The PL/SQL function `SQLERRM`, described in "SQLERRM Function" on page 13-132

    This function 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).
  - The package function `DBMS_UTILITY.FORMAT_ERROR_STACK`, described in *Oracle Database PL/SQL Packages and Types Reference*

    This function returns the full error stack, up to 2000 bytes.

Oracle recommends using DBMS_UTILITY.FORMAT_ERROR_STACK, except when using the FORALL statement with its SAVE EXCEPTIONS clause, as in Example 12–13.

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 11–22.

Example 11–22 Displaying SQLCODE and SQLERRM Values

```
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 Handling Exceptions

After an exception handler runs, control transfers to the next statement of the enclosing block (or to the invoker or host environment if there is no enclosing block). The exception handler cannot transfer control back to its own block.

For example, in Example 11–23, after the SELECT INTO statement raises ZERO_DIVIDE and the exception handler handles it, execution cannot continue from the INSERT statement that follows the SELECT INTO statement.

Example 11–23 Exception Handler Runs and Execution Ends

```
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);
  DBMS_OUTPUT.PUT_LINE('Row inserted.');
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Division by zero.');
END;
/
```

Result:

```
Division by zero.
```

If you want execution to resume with the `INSERT` statement that follows the `SELECT INTO` statement, then put the `SELECT INTO` statement in an inner block with its own `ZERO_DIVIDE` exception handler, as in Example 11–24.

Example 11–24 Exception Handler Runs and Execution Continues

```
DECLARE
  sal_calc NUMBER(8,2);
BEGIN
  INSERT INTO employees_temp (employee_id, salary, commission_pct)
  VALUES (301, 2500, 0);

  BEGIN
    SELECT (salary / commission_pct) INTO sal_calc
    FROM employees_temp
    WHERE employee_id = 301;
  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      DBMS_OUTPUT.PUT_LINE('Substituting 2500 for undefined number.');
      sal_calc := 2500;
  END;

  INSERT INTO employees_temp VALUES (302, sal_calc/100, .1);
  DBMS_OUTPUT.PUT_LINE('Enclosing block: Row inserted.');
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    DBMS_OUTPUT.PUT_LINE('Enclosing block: Division by zero.');
END;
/
```

Result:

```
Substituting 2500 for undefined number.
Enclosing block: Row inserted.
```

See Also: Example 12–13, where a bulk SQL operation continues despite exceptions

Retrying Transactions After Handling Exceptions

To retry a transaction after handling an exception that it raised, use this technique:

1. Enclose the transaction in a sub-block that has an exception-handling part.
2. In the sub-block, before the transaction starts, mark a savepoint.
3. In the exception-handling part of the sub-block, put an exception handler that rolls back to the savepoint and then tries to correct the problem.
4. Put the sub-block inside a `LOOP` statement.
5. In the sub-block, after the `COMMIT` statement that ends the transaction, put an `EXIT` statement.

   If the transaction succeeds, the `COMMIT` and `EXIT` statements execute.

   If the transaction fails, control transfers to the exception-handling part of the sub-block, and after the exception handler runs, the loop repeats.

Example 11–25 uses the preceding technique to retry a transaction whose `INSERT` statement raises the predefined exception `DUP_VAL_ON_INDEX` if the value of `res_name` is not unique.

Example 11–25 Retrying Transaction After Handling 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 transaction at most 5 times.

    DBMS_OUTPUT.PUT('Try #' || i);

    BEGIN  -- sub-block begins

       SAVEPOINT start_transaction;

       -- transaction begins

       DELETE FROM results WHERE res_answer = 'NO';

       INSERT INTO results (res_name, res_answer) VALUES (name, answer);

       -- Nonunique name raises DUP_VAL_ON_INDEX.

       -- If transaction succeeded:

       COMMIT;
       DBMS_OUTPUT.PUT_LINE(' succeeded.');
       EXIT;
```

```
    EXCEPTION
      WHEN DUP_VAL_ON_INDEX THEN
        DBMS_OUTPUT.PUT_LINE(' failed; trying again.');
        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;
/
```

Result:

```
Try #1 failed; trying again.
Try #2 succeeded.
```

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 improve existing PL/SQL code.

Topics

- PL/SQL Optimizer
- Candidates for Tuning
- Minimizing CPU Overhead
- Bulk SQL and Bulk Binding
- Chaining Pipelined Table Functions for Multiple Transformations
- Updating Large Tables in Parallel
- Collecting Data About User-Defined Identifiers
- Profiling and Tracing PL/SQL Programs
- Compiling PL/SQL Units for Native Execution

PL/SQL Optimizer

Prior to Oracle Database 10*g* Release 1 (10.1), the PL/SQL compiler translated your source text 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, PL/SQL might raise an exception earlier than expected or not at all. Setting `PLSQL_OPTIMIZE_LEVEL=1` prevents the code from being rearranged.

See Also:

- *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 invocation with a copy of the invoked subprogram (if

the invoked and invoking subprograms are in the same program unit). 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 the `INLINE` pragma:

```
PRAGMA INLINE (subprogram, 'YES')
```

If *`subprogram`* is overloaded, then the preceding pragma applies to every subprogram with that name.

With `PLSQL_OPTIMIZE_LEVEL=3`, the PL/SQL compiler seeks opportunities to inline subprograms. You need not specify subprograms to be inlined. However, you can use the `INLINE` pragma (with the preceding syntax) to give a subprogram a high priority for inlining, and then the compiler inlines it unless other considerations or limits make the inlining undesirable.

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, then use the PL/SQL hierarchical profiler (explained in *Oracle Database Advanced Application Developer's Guide*) to identify subprograms for which you want to turn off inlining. To turn off inlining for a subprogram, use the `INLINE` pragma:

```
PRAGMA INLINE (subprogram, 'NO')
```

The `INLINE` pragma affects only the immediately following declaration or statement, and only some kinds of statements.

When the `INLINE` pragma immediately precedes a declaration, it affects:

- Every invocation of the specified subprogram in that declaration
- Every initialization value in that declaration except the default initialization values of records

When the `INLINE` pragma immediately precedes one of these statements, the pragma affects every invocation of the specified subprogram in that statement:

- 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.

In Example 12–1, if `PLSQL_OPTIMIZE_LEVEL=2`, the `INLINE` pragma affects the procedure invocations `p1(1)` and `p1(2)`, but not the procedure invocations `p1(3)` and `p1(4)`.

Example 12–1 Specifying that Subprogram Is To Be Inlined

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'YES');
x:= p1(1) + p1(2) + 17;    -- These 2 invocations to p1 are inlined
...
x:= p1(3) + p1(4) + 17;    -- These 2 invocations to p1 are not inlined
...
```

In Example 12–2, if `PLSQL_OPTIMIZE_LEVEL=2`, the `INLINE` pragma affects both functions named `p2`.

Example 12–2 Specifying that 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, the `INLINE` pragma affects the procedure invocations `p1(1)` and `p1(2)`, but not the procedure invocations `p1(3)` and `p1(4)`.

Example 12–3 Specifying that Subprogram Is Not To Be Inlined

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'NO');
x:= p1(1) + p1(2) + 17;    -- These 2 invocations to p1 are not inlined
...
x:= p1(3) + p1(4) + 17;    -- These 2 invocations to p1 might be inlined
...
```

Multiple pragmas can affect the same declaration or statement. Each pragma applies its own effect to the statement. If `PRAGMA INLINE(`*`subprogram`*`,'YES')` and `PRAGMA INLINE(`*`identifier`*`,'NO')` have the same *`subprogram`*, then `'NO'` overrides `'YES'`. One `PRAGMA INLINE(`*`subprogram`*`,'NO')` overrides any number of occurrences of `PRAGMA INLINE(`*`subprogram`*`,'YES')`, and the order of these pragmas is not important.

In Example 12–4, the second `INLINE` pragma overrides both the first and third `INLINE` pragmas.

Example 12–4 PRAGMA INLINE ... 'NO' Overrides PRAGMA INLINE ... 'YES'

```
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 invocations to p1 are not inlined
...
```

See Also:

- *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`

Candidates for Tuning

The following kinds of PL/SQL code are very likely to benefit from tuning:

- Older code that does not take advantage of new PL/SQL language features.

  For information about new PL/SQL language features, see "What's New in PL/SQL?" on page xxxi.

  Tip: Before tuning older code, benchmark the current system and profile the older subprograms that your program invokes (see "Profiling and Tracing PL/SQL Programs" on page 12-49). With the many automatic optimizations of the PL/SQL optimizer (described in "PL/SQL Optimizer" on page 12-1), you might see performance improvements before doing any tuning.

- Older dynamic SQL statements written with the `DBMS_SQL` package.

  If you know at compile time the number and data types of the input and output variables of a dynamic SQL statement, then you can rewrite the statement in native dynamic SQL, which runs noticeably faster than equivalent code that uses the `DBMS_SQL` package (especially when it can be optimized by the compiler). For more information, see Chapter 7, "PL/SQL Dynamic SQL."

- Code that spends much time processing SQL statements.

  See "Tune SQL Statements" on page 12-5.

- Functions invoked in queries, which might run millions of times.

  See "Tune Function Invocations in Queries" on page 12-5.

- Code that spends much time looping through query results.

  See "Tune Loops" on page 12-8.

- Code that does many numeric computations.

  See "Tune Computation-Intensive PL/SQL Code" on page 12-8.

- Code that spends much time processing PL/SQL statements (as opposed to issuing database definition language (DDL) statements that PL/SQL passes directly to SQL).

  See "Compiling PL/SQL Units for Native Execution" on page 12-51.

Minimizing CPU Overhead

Topics

- Tune SQL Statements
- Tune Function Invocations in Queries

- Tune Subprogram Invocations
- Tune Loops
- Tune Computation-Intensive PL/SQL Code
- Use SQL Character Functions
- Put Least Expensive Conditional Tests First

Tune SQL Statements

The most common cause of slowness in PL/SQL programs is slow SQL statements. To make SQL statements in a PL/SQL program as efficient as possible:

- Use appropriate indexes.

  For details, see *Oracle Database Performance Tuning Guide*.

- Use query hints to avoid unnecessary full-table scans.

  For details, see *Oracle Database Performance Tuning Guide*.

- Collect current statistics on all tables, using the subprograms in the `DBMS_STATS` package.

  For details, see *Oracle Database Performance Tuning Guide*.

- Analyze the execution plans and performance of the SQL statements, using:
  - `EXPLAIN PLAN` statement

    For details, see *Oracle Database Performance Tuning Guide*.

  - SQL Trace facility with `TKPROF` utility

    For details, see *Oracle Database Performance Tuning Guide*.

- Use bulk SQL, a set of PL/SQL features that minimizes the performance overhead of the communication between PL/SQL and SQL.

  For details, see "Bulk SQL and Bulk Binding" on page 12-11.

Tune Function Invocations in Queries

Functions invoked in queries might run millions of times. Do not invoke a function in a query unnecessarily, and make the invocation as efficient as possible.

Create a function-based index on the table in the query. The `CREATE INDEX` statement (described in *Oracle Database SQL Language Reference*) might take a while, but the query can run much faster because the function value for each row is cached.

> **See Also:** "PL/SQL Function Result Cache" on page 8-31 for information about caching the results of PL/SQL functions

If the query passes a column to a function, then the query cannot use user-created indexes on that column, so the query might invoke the function for every row of the table (which might be very large). To minimize the number of function invocations, use a nested query. Have the inner query filter the result set to a small number of rows, and have the outer query invoke the function for only those rows.

In Example 12–5, the two queries produce the same result set, but the second query is more efficient than the first. (In the example, the times and time difference are very small, because the `EMPLOYEES` table is very small. For a very large table, they would be significant.)

Example 12–5 Nested Query Improves Performance

```
DECLARE
  starting_time  TIMESTAMP WITH TIME ZONE;
  ending_time    TIMESTAMP WITH TIME ZONE;
BEGIN
  -- Invokes SQRT for every row of employees table:

  SELECT SYSTIMESTAMP INTO starting_time FROM DUAL;

  FOR item IN (
    SELECT DISTINCT(SQRT(department_id)) col_alias
    FROM employees
    ORDER BY col_alias
  )
  LOOP
    DBMS_OUTPUT.PUT_LINE('Square root of dept. ID = ' || item.col_alias);
  END LOOP;

  SELECT SYSTIMESTAMP INTO ending_time FROM DUAL;

  DBMS_OUTPUT.PUT_LINE('Time = ' || TO_CHAR(ending_time - starting_time));

  -- Invokes SQRT for every distinct department_id of employees table:

  SELECT SYSTIMESTAMP INTO starting_time FROM DUAL;

  FOR item IN (
    SELECT SQRT(department_id) col_alias
    FROM (SELECT DISTINCT department_id FROM employees)
    ORDER BY col_alias
  )
  LOOP
    IF item.col_alias IS NOT NULL THEN
      DBMS_OUTPUT.PUT_LINE('Square root of dept. ID = ' || item.col_alias);
    END IF;
  END LOOP;

  SELECT SYSTIMESTAMP INTO ending_time FROM DUAL;

  DBMS_OUTPUT.PUT_LINE('Time = ' || TO_CHAR(ending_time - starting_time));
END;
/
```

Result:

```
Square root of dept. ID = 3.16227766016837933199889354443271853372
Square root of dept. ID = 4.47213595499957939281834733746255247088
Square root of dept. ID = 5.47722557505166113456969782800802133953
Square root of dept. ID = 6.32455532033675866399778708886543706744
Square root of dept. ID = 7.07106781186547524400844362104849039285
Square root of dept. ID = 7.74596669241483377035853079956479922167
Square root of dept. ID = 8.36660026534075547978172025785187489393
Square root of dept. ID = 8.94427190999915878563669467492510494176
Square root of dept. ID = 9.48683298050513799599668063329815560116
Square root of dept. ID = 10
Square root of dept. ID = 10.48808848170151546991453513679937598475
Time = +000000000 00:00:00.046000000
Square root of dept. ID = 3.16227766016837933199889354443271853372
Square root of dept. ID = 4.47213595499957939281834733746255247088
Square root of dept. ID = 5.47722557505166113456969782800802133953
```

```
Square root of dept. ID = 6.3245553203367586639977870888654370674 4
Square root of dept. ID = 7.0710678118654752440084436210484903928 5
Square root of dept. ID = 7.7459666924148337703585307995647992216 7
Square root of dept. ID = 8.3666002653407554797817202578518748939 3
Square root of dept. ID = 8.9442719099991587856366946749251049417 6
Square root of dept. ID = 9.4868329805051379959966806332981556011 6
Square root of dept. ID = 10
Square root of dept. ID = 10.488088481701515469914535136799375984 75
Time = +000000000 00:00:00.000000000
```

Tune Subprogram Invocations

If a subprogram has OUT or IN OUT parameters, you can sometimes decrease its invocation overhead by declaring those parameters with the NOCOPY hint (described in "NOCOPY" on page 13-81).

By default, PL/SQL passes OUT and IN OUT subprogram parameters by value. Before running the subprogram, PL/SQL copies each OUT and IN OUT parameter to a temporary variable, which holds the value of the parameter during subprogram execution. If the subprogram is exited normally, then PL/SQL copies the value of the temporary variable to the corresponding actual parameter. If the subprogram is exited with an unhandled exception, then PL/SQL does not change the value of the actual parameter.

When OUT or IN OUT parameters represent large data structures such as collections, records, and instances of ADTs, copying them slows execution and increases memory use—especially for an instance of an ADT.

For each invocation of an ADT method, PL/SQL copies every attribute of the ADT. If the method is exited normally, then PL/SQL applies any changes that the method made to the attributes. If the method is exited with an unhandled exception, then PL/SQL does not change the attributes.

If your program does not require that an OUT or IN OUT parameter retain its pre-invocation value if the subprogram ends with an unhandled exception, then include the NOCOPY hint in the parameter declaration. The NOCOPY hint requests (but does not ensure) that the compiler pass the corresponding actual parameter by reference instead of value. For more information about NOCOPY, see "NOCOPY" on page 13-81. For information about using NOCOPY with member methods of ADTs, see *Oracle Database Object-Relational Developer's Guide*.

> **Caution:** Do not rely on NOCOPY (which the compiler might or might not obey for a particular invocation) to ensure that an actual parameter or ADT attribute retains its pre-invocation value if the subprogram is exited with an unhandled exception. Instead, ensure that the subprogram handle all exceptions.

In Example 12–6, if the compiler obeys the NOCOPY hint for the invocation of do_nothing2, then the invocation of do_nothing2 is faster than the invocation of do_nothing1.

Example 12–6 NOCOPY Subprogram 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;
/
```

Tune Loops

Because PL/SQL applications are often built around loops, it is important to optimize both the loops themselves and the code inside them.

If you must loop through a result set more than once, or issue other queries as you loop through a result set, you might be able to change the original query to give you exactly the results you want. Explore the SQL set operators that let you combine multiple queries, described in *Oracle Database SQL Language Reference*.

You can also use subqueries to do the filtering and sorting in multiple stages—see "Query Result Set Processing with Subqueries" on page 6-26.

> **See Also:** "Bulk SQL and Bulk Binding" on page 12-11

Tune Computation-Intensive PL/SQL Code

These recommendations apply especially (but not only) to computation-intensive PL/SQL code.

Topics

- Use Data Types that Use Hardware Arithmetic
- Avoid Constrained Subtypes in Performance-Critical Code

- Minimize Implicit Data Type Conversion

Use Data Types that Use Hardware Arithmetic

Avoid using data types in the NUMBER data type family (described in "NUMBER Data Type Family" on page E-2). These data types are represented internally in a format designed for portability and arbitrary scale and precision, not for performance. Operations on data of these types use library arithmetic, while operations on data of the types PLS_INTEGER, BINARY_FLOAT and BINARY_DOUBLE use hardware arithmetic.

For local integer variables, use PLS_INTEGER, described in "PLS_INTEGER and BINARY_INTEGER Data Types" on page 3-8. For variables that can never have the value NULL, do not need overflow checking, and are not used in performance-critical code, use SIMPLE_INTEGER, described in "SIMPLE_INTEGER Subtype of PLS_INTEGER" on page 3-10.

For floating-point variables, use BINARY_FLOAT or BINARY_DOUBLE, described in *Oracle Database SQL Language Reference*. For variables that can never have the value NULL and are not used in performance-critical code, use SIMPLE_FLOAT or SIMPLE_DOUBLE, explained in "Additional PL/SQL Subtypes of BINARY_FLOAT and BINARY_DOUBLE" on page 3-3.

> **Note:** BINARY_FLOAT and BINARY_DOUBLE and their subtypes 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.

Many SQL numeric functions (described in *Oracle Database SQL Language Reference*) are overloaded with versions that accept BINARY_FLOAT and BINARY_DOUBLE parameters. You can speed up computation-intensive code by passing variables of these data types to such functions, and by invoking the conversion functions TO_BINARY_FLOAT (described in *Oracle Database SQL Language Reference*) and TO_BINARY_DOUBLE (described in *Oracle Database SQL Language Reference*) when passing expressions to such functions.

Avoid Constrained Subtypes in Performance-Critical Code

In performance-critical code, avoid constrained subtypes (described in "Constrained Subtypes" on page 3-12). Each assignment to a variable or parameter of a constrained subtype requires extra checking at run time to ensure that the value to be assigned does not violate the constraint.

> **See Also:** Appendix E, "PL/SQL Predefined Data Types" includes predefined constrained subtypes

Minimize Implicit Data Type Conversion

At run time, PL/SQL converts between different data types implicitly (automatically) if necessary. For example, if you assign a PLS_INTEGER variable to a NUMBER variable, then PL/SQL converts the PLS_INTEGER value to a NUMBER value (because the internal representations of the values differ).

Whenever possible, minimize implicit conversions. For example:

- If a variable is to be either inserted into a table column or assigned a value from a table column, then give the variable the same data type as the table column.

Tip: Declare the variable with the `%TYPE` attribute, described in "%TYPE Attribute" on page 13-134.

- Make each literal the same data type as the variable to which it is assigned or the expression in which it appears.
- Convert values from SQL data types to PL/SQL data types and then use the converted values in expressions.

  For example, convert `NUMBER` values to `PLS_INTEGER` values and then use the `PLS_INTEGER` values in expressions. `PLS_INTEGER` operations use hardware arithmetic, so they are faster than `NUMBER` operations, which use library arithmetic. For more information about the `PLS_INTEGER` data type, see "PLS_INTEGER and BINARY_INTEGER Data Types" on page 3-8.
- Before assigning a value of one SQL data type to a variable of another SQL data type, explicitly convert the source value to the target data type, using a SQL conversion function (for information about SQL conversion functions, see *Oracle Database SQL Language Reference*).
- Overload your subprograms with versions that accept parameters of different data types and optimize each version for its parameter types. For information about overloaded subprograms, see "Overloaded Subprograms" on page 8-25.

See Also:

- *Oracle Database SQL Language Reference* for information about implicit conversion of SQL data types (which are also PL/SQL data types)
- "Subtypes with Base Types in Same Data Type Family" on page 3-14

Use SQL Character Functions

SQL has many highly optimized character functions, which use low-level code that is more efficient than PL/SQL code. Use these functions instead of writing PL/SQL code to do the same things.

See:

- *Oracle Database SQL Language Reference* for information about SQL character functions that return character values
- *Oracle Database SQL Language Reference* for information about SQL character functions that return NLS character values
- *Oracle Database SQL Language Reference* for information about SQL character functions that return number values
- Example 6–6 for an example of PL/SQL code that uses SQL character function `REGEXP_LIKE`

Put Least Expensive Conditional Tests First

PL/SQL stops evaluating a logical expression as soon as it can determine the result. Take advantage of this short-circuit evaluation by putting the conditions that are least expensive to evaluate first in logical expressions whenever possible. For example, test the values of PL/SQL variables before testing function return values, so that if the variable tests fail, PL/SQL need not invoke the functions:

```
IF boolean_variable OR (number > 10) OR boolean_function(parameter) THEN ...
```

See Also: "Short-Circuit Evaluation" on page 2-31

Bulk SQL and Bulk Binding

Bulk SQL minimizes the performance overhead of the communication between PL/SQL and SQL.

PL/SQL and SQL communicate as follows: To run a `SELECT INTO` or DML statement, the PL/SQL engine sends the query or DML statement to the SQL engine. The SQL engine runs the query or DML statement and returns the result to the PL/SQL engine.

The PL/SQL features that comprise bulk SQL are the `FORALL` statement and the `BULK COLLECT` clause. The `FORALL` statement sends DML statements from PL/SQL to SQL in batches rather than one at a time. The `BULK COLLECT` clause returns results from SQL to PL/SQL in batches rather than one at a time. If a query or DML statement affects four or more database rows, then bulk SQL can significantly improve performance.

Assigning values to PL/SQL variables that appear 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 or host variable in the database
Out-bind	When the `RETURNING INTO` clause of an `INSERT`, `UPDATE`, or `DELETE` statement assigns a database value to a PL/SQL or host variable
`DEFINE`	When a `SELECT` or `FETCH` statement assigns a database value to a PL/SQL or host variable

For in-binds and out-binds, bulk SQL uses **bulk binding**; that is, it binds an entire collection of values at once. For a collection of *n* elements, bulk SQL uses a single operation to perform the equivalent of *n* `SELECT INTO` or DML statements. A query that uses bulk SQL can return any number of rows, without using a `FETCH` statement for each one.

Note: Parallel DML is disabled with bulk SQL.

Topics

- FORALL Statement
- BULK COLLECT Clause
- Using FORALL Statement and BULK COLLECT Clause Together
- Client Bulk-Binding of Host Arrays

FORALL Statement

The `FORALL` statement, a feature of bulk SQL, sends DML statements from PL/SQL to SQL in batches rather than one at a time. To understand the `FORALL` statement, first consider the `FOR LOOP` statement in Example 12–7. It sends these DML statements from PL/SQL to SQL one at a time:

```
DELETE FROM employees_temp WHERE department_id = depts(10);
```

```
DELETE FROM employees_temp WHERE department_id = depts(30);
DELETE FROM employees_temp WHERE department_id = depts(70);
```

Example 12–7 DELETE Statement in FOR LOOP Statement

```
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
  FOR i IN depts.FIRST..depts.LAST LOOP
    DELETE FROM employees_temp
    WHERE department_id = depts(i);
  END LOOP;
END;
/
```

Now consider the FORALL statement in Example 12–8. It sends the same three DML statements from PL/SQL to SQL as a batch.

Example 12–8 DELETE Statement in FORALL Statement

```
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);
END;
/
```

A FORALL statement is usually much faster than an equivalent FOR LOOP statement. However, a FOR LOOP statement can contain multiple DML statements, while a FORALL statement can contain only one. The batch of DML statements that a FORALL statement sends to SQL differ only in their VALUES and WHERE clauses. The values in those clauses must come from existing, populated collections.

> **Note:** The DML statement in a FORALL statement can reference multiple collections, but performance benefits apply only to collection references that use the FORALL index variable as a subscript.

Example 12–9 inserts the same collection elements into two database tables, using a FOR LOOP statement for the first table and a FORALL statement for the second table and showing how long each statement takes. (Times vary from run to run.)

Example 12–9 Time Difference for INSERT Statement in FOR LOOP and FORALL Statements

```
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 := 50000;
  t1  INTEGER;
  t2  INTEGER;
  t3  INTEGER;
BEGIN
  FOR j IN 1..iterations LOOP  -- populate collections
    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 is similar to:

```
Execution Time (secs)
---------------------
FOR LOOP: 2.16
FORALL:   .11

PL/SQL procedure successfully completed.
```

In Example 12–10, the `FORALL` statement applies to a subset of a collection.

Example 12–10 FORALL Statement for Subset 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
    DELETE FROM employees_temp WHERE department_id = depts(j);
END;
/
```

Topics

- FORALL Statements for Sparse Collections
- Unhandled Exceptions in FORALL Statements
- Handling FORALL Exceptions Immediately
- Handling FORALL Exceptions After FORALL Statement Completes
- Getting Number of Rows Affected by FORALL Statement

See Also:

- "FORALL Statement" on page 13-77 for its complete syntax and semantics, including restrictions
- "Implicit Cursors" on page 6-5 for information about implicit cursor attributes in general and other implicit cursor attributes that you can use with the FORALL statement

FORALL Statements for Sparse Collections

If the FORALL statement bounds clause references a sparse collection, then specify only existing index values, using either the INDICES OF or VALUES OF clause. You can use INDICES OF for any collection except an associative array indexed by string. You can use VALUES OF only for a collection of PLS_INTEGER elements indexed by PLS_INTEGER.

A collection of PLS_INTEGER elements indexed by PLS_INTEGER can be an **index collection**; that is, a collection of pointers to elements of another collection (the **indexed collection**).

Index collections are useful for processing different subsets of the same collection with different FORALL statements. Instead of copying elements of the original collection into new collections that represent the subsets (which can use significant time and memory), represent each subset with an index collection and then use each index collection in the VALUES OF clause of a different FORALL statement.

Example 12–11 uses a FORALL statement with the INDICES OF clause to populate a table with the elements of a sparse collection. Then it uses two FORALL statements with VALUES OF clauses to populate two tables with subsets of a collection.

Example 12–11 FORALL Statements for Sparse Collection and Its Subsets

```
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
  SUBTYPE cust_name IS valid_orders.cust_name%TYPE;
  TYPE cust_typ IS TABLE OF cust_name;
  cust_tab  cust_typ;  -- Collection of customer names

  SUBTYPE order_amount IS valid_orders.amount%TYPE;
  TYPE amount_typ IS TABLE OF NUMBER;
  amount_tab  amount_typ;  -- Collection of order amounts

  TYPE index_pointer_t IS TABLE OF PLS_INTEGER;

  /* Collections for pointers to elements of cust_tab collection
     (to represent two subsets of cust_tab): */

  big_order_tab       index_pointer_t := index_pointer_t();
  rejected_order_tab  index_pointer_t := index_pointer_t();

  PROCEDURE populate_data_collections IS
  BEGIN
    cust_tab := cust_typ(
      'Company1','Company2','Company3','Company4','Company5'
    );

    amount_tab := amount_typ(5000.01, 0, 150.25, 4000.00, NULL);
  END;

BEGIN
  populate_data_collections;

  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:

  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;

  -- cust_tab is now a sparse collection.

  DBMS_OUTPUT.PUT_LINE ('--- Order 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;

  -- Using sparse collection, populate valid_orders table:

  FORALL i IN INDICES OF cust_tab
    INSERT INTO valid_orders (cust_name, amount)
    VALUES (cust_tab(i), amount_tab(i));

  populate_data_collections;  -- Restore original order data

  -- cust_tab is a dense collection again.

  /* Populate collections of pointers to elements of cust_tab collection
     (which represent two subsets of cust_tab): */

  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;

  /* Using each subset in a different FORALL statement,
     populate rejected_orders and big_orders tables: */

  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));
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
```

```
ORDER BY cust_name;
```

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
ORDER BY cust_name;
```

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
ORDER BY cust_name;
```

Result:

```
Customer                        Rejected order amount
------------------------------- ---------------------
Company2                                          0
Company5

2 rows selected.
```

See Also: "Sparse Collections and SQL%BULK_EXCEPTIONS" on page 12-21

Unhandled Exceptions in FORALL Statements

In a `FORALL` statement without the `SAVE EXCEPTIONS` clause, if one DML statement raises an unhandled exception, then PL/SQL stops the `FORALL` statement and rolls back all changes made by previous DML statements.

For example, the `FORALL` statement in Example 12–8 executes these DML statements in this order, unless one of them raises an unhandled exception:

```
DELETE FROM employees_temp WHERE department_id = depts(10);
DELETE FROM employees_temp WHERE department_id = depts(30);
DELETE FROM employees_temp WHERE department_id = depts(70);
```

If the third statement raises an unhandled exception, then PL/SQL rolls back the changes that the first and second statements made. If the second statement raises an

unhandled exception, then PL/SQL rolls back the changes that the first statement made and never runs the third statement.

You can handle exceptions raised in a FORALL statement in either of these ways:

- As each exception is raised (see "Handling FORALL Exceptions Immediately" on page 12-18)
- After the FORALL statement completes execution, by including the SAVE EXCEPTIONS clause (see "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-19)

Handling FORALL Exceptions Immediately

To handle exceptions raised in a FORALL statement immediately, omit the SAVE EXCEPTIONS clause and write the appropriate exception handlers. (For information about exception handlers, see Chapter 11, "PL/SQL Error Handling.") If one DML statement raises a handled exception, then PL/SQL rolls back the changes made by that statement, but does not roll back changes made by previous DML statements.

In Example 12–12, the FORALL statement is designed to run three UPDATE statements. However, the second one raises an exception. An exception handler handles the exception, displaying the error message and committing the change made by the first UPDATE statement. The third UPDATE statement never runs.

Example 12–12 Handling FORALL Exceptions Immediately

```
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);
  error_message  VARCHAR2(100);

BEGIN
  -- Populate table:

  INSERT INTO emp_temp (deptno, job) VALUES (10, 'Clerk');
  INSERT INTO emp_temp (deptno, job) VALUES (20, 'Bookkeeper');
  INSERT INTO emp_temp (deptno, job) VALUES (30, 'Analyst');
  COMMIT;

  -- Append 9-character string to each job:

  FORALL j IN depts.FIRST..depts.LAST
    UPDATE emp_temp SET job = job || ' (Senior)'
    WHERE deptno = depts(j);

EXCEPTION
  WHEN OTHERS THEN
    error_message := SQLERRM;
    DBMS_OUTPUT.PUT_LINE (error_message);

    COMMIT;  -- Commit results of successful updates
END;
/
```

Result:

```
ORA-12899: value too large for column "HR"."EMP_TEMP"."JOB" (actual: 19,
maximum: 18)

PL/SQL procedure successfully completed.
```

Query:

```
SELECT * FROM emp_temp;
```

Result:

```
    DEPTNO JOB
---------- ------------------
        10 Clerk (Senior)
        20 Bookkeeper
        30 Analyst

3 rows selected.
```

Handling FORALL Exceptions After FORALL Statement Completes

To allow a `FORALL` statement to continue even if some of its DML statements fail, include the `SAVE EXCEPTIONS` clause. When a DML statement fails, PL/SQL does not raise an exception; instead, it saves information about the failure. After the `FORALL` statement completes, PL/SQL raises a single exception for the `FORALL` statement (ORA-24381). In the exception handler for ORA-24381, you can get information about each individual DML statement failure from the implicit cursor attribute `SQL%BULK_EXCEPTIONS`.

`SQL%BULK_EXCEPTIONS` is like an associative array of information about the DML statements that failed during the most recently run `FORALL` statement.

`SQL%BULK_EXCEPTIONS.COUNT` is the number of DML statements that failed. If `SQL%BULK_EXCEPTIONS.COUNT` is not zero, then for each index value *i* from 1 through `SQL%BULK_EXCEPTIONS.COUNT`:

- `SQL%BULK_EXCEPTIONS(i).ERROR_INDEX` is the number of the DML statement that failed.
- `SQL%BULK_EXCEPTIONS(i).ERROR_CODE` is the Oracle Database error code for the failure.

For example, if a `FORALL SAVE EXCEPTIONS` statement runs 100 DML statements, and the tenth and sixty-fourth ones fail with error codes ORA-12899 and ORA-19278, respectively, then:

- `SQL%BULK_EXCEPTIONS.COUNT` = 2
- `SQL%BULK_EXCEPTIONS(1).ERROR_INDEX` = 10
- `SQL%BULK_EXCEPTIONS(1).ERROR_CODE` = 12899
- `SQL%BULK_EXCEPTIONS(2).ERROR_INDEX` = 64
- `SQL%BULK_EXCEPTIONS(2).ERROR_CODE` = 19278

Note: After a `FORALL` statement *without* the `SAVE EXCEPTIONS` clause raises an exception, `SQL%BULK_EXCEPTIONS.COUNT` = 1.

With the error code, you can get the associated error message with the SQLERRM function (described in "SQLERRM Function" on page 13-132):

```
SQLERRM(-(SQL%BULK_EXCEPTIONS(i).ERROR_CODE))
```

However, the error message that SQLERRM returns excludes any substitution arguments (compare the error messages in Example 12–12 and Example 12–13).

Example 12–13 is like Example 12–12 except:

- The FORALL statement includes the SAVE EXCEPTIONS clause.
- The exception-handling part has an exception handler for ORA-24381, the internally defined exception that PL/SQL raises implicitly when a bulk operation raises and saves exceptions. The example gives ORA-24381 the user-defined name dml_errors.
- The exception handler for dml_errors uses SQL%BULK_EXCEPTIONS and SQLERRM (and some local variables) to show the error message and which statement, collection item, and string caused the error.

Example 12–13 Handling FORALL Exceptions After FORALL Statement Completes

```
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts        NumList := NumList(10, 20, 30);

  error_message  VARCHAR2(100);
  bad_stmt_no    PLS_INTEGER;
  bad_deptno     emp_temp.deptno%TYPE;
  bad_job        emp_temp.job%TYPE;

  dml_errors  EXCEPTION;
  PRAGMA EXCEPTION_INIT(dml_errors, -24381);
BEGIN
  -- Populate table:

  INSERT INTO emp_temp (deptno, job) VALUES (10, 'Clerk');
  INSERT INTO emp_temp (deptno, job) VALUES (20, 'Bookkeeper');
  INSERT INTO emp_temp (deptno, job) VALUES (30, 'Analyst');
  COMMIT;

  -- Append 9-character string to each job:

  FORALL j IN depts.FIRST..depts.LAST SAVE EXCEPTIONS
    UPDATE emp_temp SET job = job || ' (Senior)'
    WHERE deptno = depts(j);

EXCEPTION
  WHEN dml_errors THEN
    FOR i IN 1..SQL%BULK_EXCEPTIONS.COUNT LOOP
      error_message := SQLERRM(-(SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
      DBMS_OUTPUT.PUT_LINE (error_message);

      bad_stmt_no := SQL%BULK_EXCEPTIONS(i).ERROR_INDEX;
      DBMS_OUTPUT.PUT_LINE('Bad statement #: ' || bad_stmt_no);

      bad_deptno := depts(bad_stmt_no);
      DBMS_OUTPUT.PUT_LINE('Bad department #: ' || bad_deptno);

      SELECT job INTO bad_job FROM emp_temp WHERE deptno = bad_deptno;
```

```
      DBMS_OUTPUT.PUT_LINE('Bad job: ' || bad_job);
    END LOOP;

    COMMIT;  -- Commit results of successful updates

    WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE('Unrecognized error.');
END;
/
```

Result:

```
ORA-12899: value too large for column  (actual: , maximum: )
Bad statement #: 2
Bad department #: 20
Bad job: Bookkeeper

PL/SQL procedure successfully completed.
```

Query:

```
SELECT * FROM emp_temp;
```

Result:

```
    DEPTNO JOB
---------- ------------------
        10 Clerk (Senior)
        20 Bookkeeper
        30 Analyst (Senior)

3 rows selected.
```

Sparse Collections and SQL%BULK_EXCEPTIONS If the `FORALL` statement bounds clause references a sparse collection, then to find the collection element that caused a DML statement to fail, you must step through the elements one by one until you find the element whose index is `SQL%BULK_EXCEPTIONS(i).ERROR_INDEX`. Then, if the `FORALL` statement uses the `VALUES OF` clause to reference a collection of pointers into another collection, you must find the element of the other collection whose subscript is `SQL%BULK_EXCEPTIONS(i).ERROR_INDEX`.

Getting Number of Rows Affected by FORALL Statement

After a `FORALL` statement completes, you can get the number of rows that each DML statement affected from the implicit cursor attribute `SQL%BULK_ROWCOUNT`. (To get the total number of rows affected by the `FORALL` statement, use the implicit cursor attribute `SQL%ROWCOUNT`, described in "SQL%ROWCOUNT Attribute: How Many Rows Were Affected?" on page 6-7.)

`SQL%BULK_ROWCOUNT` is like an associative array whose *i*th element is the number of rows affected by the *i*th DML statement in the most recently completed `FORALL` statement.

Example 12–14 uses `SQL%BULK_ROWCOUNT` to show how many rows each `DELETE` statement in the `FORALL` statement deleted and `SQL%ROWCOUNT` to show the total number of rows deleted.

Example 12–14 Showing Number of Rows Affected by Each DELETE in FORALL

```
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);

  FOR i IN depts.FIRST..depts.LAST LOOP
    DBMS_OUTPUT.PUT_LINE (
      'Statement #' || i || ' deleted ' ||
      SQL%BULK_ROWCOUNT(i) || ' rows.'
    );
  END LOOP;

  DBMS_OUTPUT.PUT_LINE('Total rows deleted: ' || SQL%ROWCOUNT);
END;
/
```

Result:

```
Statement #1 deleted 6 rows.
Statement #2 deleted 45 rows.
Statement #3 deleted 5 rows.
Total rows deleted: 56
```

Example 12–15 uses SQL%BULK_ROWCOUNT to show how many rows each INSERT SELECT construct in the FORALL statement inserted and SQL%ROWCOUNT to show the total number of rows inserted.

Example 12–15 Showing Number of Rows Affected by Each INSERT SELECT in FORALL

```
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)
      ORDER BY department_id, employee_id;

  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
```

BULK COLLECT Clause

The `BULK COLLECT` clause, a feature of bulk SQL, returns results from SQL to PL/SQL in batches rather than one at a time. The `BULK COLLECT` clause can appear in:

- `SELECT INTO` statement
- `FETCH` statement
- `RETURNING INTO` clause of:
  - `DELETE` statement
  - `INSERT` statement
  - `UPDATE` statement
  - `EXECUTE IMMEDIATE` statement

With the `BULK COLLECT` clause, each of the preceding statements retrieves an entire result set and stores it in one or more collection variables in a single operation (which is more efficient than using a loop statement to retrieve one result row at a time).

> **Note:** PL/SQL processes the BULK COLLECT clause similar to the way it processes a FETCH statement inside a LOOP statement. PL/SQL does not raise an exception when a statement with a BULK COLLECT clause returns no rows. You must check the target collections for emptiness (if they are associative arrays) or nullness (if they are varrays or nested tables), as in Example 12–22.

Topics

- SELECT INTO Statement with BULK COLLECT Clause
- FETCH Statement with BULK COLLECT Clause
- RETURNING INTO Clause with BULK COLLECT Clause

SELECT INTO Statement with BULK COLLECT Clause

The SELECT INTO statement with the BULK COLLECT clause (also called the SELECT BULK COLLECT INTO statement) selects an entire result set into one or more collection variables. For more information, see "SELECT INTO Statement" on page 13-125.

> **Caution:** The SELECT BULK COLLECT INTO statement is vulnerable to aliasing, which can cause unexpected results. For details, see "SELECT BULK COLLECT INTO Statements and Aliasing" on page 12-26.

Example 12–16 uses a SELECT BULK COLLECT INTO statement to select two database columns into two collections (nested tables).

Example 12–16 Bulk-Selecting Two Database Columns into Two Nested Tables

```
DECLARE
  TYPE NumTab IS TABLE OF employees.employee_id%TYPE;
  TYPE NameTab IS TABLE OF employees.last_name%TYPE;

  enums NumTab;
  names NameTab;

  PROCEDURE print_first_n (n POSITIVE) IS
  BEGIN
    IF enums.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE ('Collections are empty.');
    ELSE
      DBMS_OUTPUT.PUT_LINE ('First ' || n || ' employees:');

      FOR i IN 1 .. n LOOP
        DBMS_OUTPUT.PUT_LINE (
          '  Employee #' || enums(i) || ': ' || names(i));
      END LOOP;
    END IF;
  END;

BEGIN
  SELECT employee_id, last_name
  BULK COLLECT INTO enums, names
  FROM employees
  ORDER BY employee_id;
```

```
  print_first_n(3);
  print_first_n(6);
END;
/
```

Result:

```
First 3 employees:
Employee #100: King
Employee #101: Kochhar
Employee #102: De Haan
First 6 employees:
Employee #100: King
Employee #101: Kochhar
Employee #102: De Haan
Employee #103: Hunold
Employee #104: Ernst
Employee #105: Austin
```

Example 12–17 uses a `SELECT BULK COLLECT INTO` statement to select a result set into a nested table of records.

Example 12–17 Bulk-Selecting into Nested Table of Records

```
DECLARE
  CURSOR c1 IS
    SELECT first_name, last_name, hire_date
    FROM employees;

  TYPE NameSet IS TABLE OF c1%ROWTYPE;

  stock_managers  NameSet;  -- nested table of records

BEGIN
  -- Assign values to nested table of records:

  SELECT first_name, last_name, hire_date
    BULK COLLECT INTO stock_managers
    FROM employees
    WHERE job_id = 'ST_MAN'
    ORDER BY hire_date;

  -- Print nested table of records:

    FOR i IN stock_managers.FIRST .. stock_managers.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
        stock_managers(i).hire_date || ' ' ||
        stock_managers(i).last_name  || ', ' ||
        stock_managers(i).first_name
      );
    END LOOP;END;
/
```

Result:

```
01-MAY-03 Kaufling, Payam
18-JUL-04 Weiss, Matthew
10-APR-05 Fripp, Adam
10-OCT-05 Vollman, Shanta
16-NOV-07 Mourgos, Kevin
```

Topics

- SELECT BULK COLLECT INTO Statements and Aliasing
- Row Limits for SELECT BULK COLLECT INTO Statements
- Guidelines for Looping Through Collections

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 NOCOPY 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 with Parameters Passed by Reference" on page 8-16

In Example 12–18, 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 12–19 and Example 12–20.

Example 12–18 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 12–19 uses a cursor to achieve the result intended by Example 12–18.

Example 12–19 Cursor Workaround for Example 12–18

```
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 12–20 selects specific values from a collection, `numbers1`, and then stores them in a different collection, `numbers2`. Example 12–20 runs faster than Example 12–19.

Example 12–20 Second Collection Workaround for Example 12–18

```
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
```

Row Limits for SELECT BULK COLLECT INTO Statements A `SELECT BULK COLLECT INTO` statement that returns a large number of rows produces a large collection. To limit the number of rows and the collection size, use either the `ROWNUM` pseudocolumn (described in *Oracle Database SQL Language Reference*) or `SAMPLE` clause (described in *Oracle Database SQL Language Reference*).

In Example 12–21, the first `SELECT BULK COLLECT INTO` statement uses `ROWNUM` to limit the number of rows to 50, and the second `SELECT BULK COLLECT INTO` statement uses `SAMPLE` to limit the number of rows to approximately 10% of the total.

Example 12–21 Limiting Bulk Selection with ROWNUM and SAMPLE

```
DECLARE
  TYPE SalList IS TABLE OF employees.salary%TYPE;
  sals SalList;
BEGIN
```

```
  SELECT salary BULK COLLECT INTO sals
  FROM employees
  WHERE ROWNUM <= 50;

  SELECT salary BULK COLLECT INTO sals FROM employees SAMPLE (10);
END;
/
```

Guidelines for Looping Through Collections When a result set is stored in a collection, it is easy to loop through the rows and refer to different columns. This technique can be very fast, but also very memory-intensive. If you use it often:

- To loop once through the result set, use a cursor `FOR LOOP` (see "Query Result Set Processing With Cursor FOR LOOP Statements" on page 6-24).

  This technique avoids the memory overhead of storing a copy of the result set.

- Instead of looping through the result set to search for certain values or filter the results into a smaller set, do the searching or filtering in the query of the `SELECT INTO` statement.

  For example, in simple queries, use `WHERE` clauses; in queries that compare multiple result sets, use set operators such as `INTERSECT` and `MINUS`. For information about set operators, see *Oracle Database SQL Language Reference*.

- Instead of looping through the result set and running another query for each result row, use a subquery in the query of the `SELECT INTO` statement (see "Query Result Set Processing with Subqueries" on page 6-26).

- Instead of looping through the result set and running another DML statement for each result row, use the `FORALL` statement (see "FORALL Statement" on page 12-11).

FETCH Statement with BULK COLLECT Clause

The `FETCH` statement with the `BULK COLLECT` clause (also called the `FETCH BULK COLLECT` statement) fetches an entire result set into one or more collection variables. For more information, see "FETCH Statement" on page 13-71.

Example 12–22 uses a `FETCH BULK COLLECT` statement to fetch an entire result set into two collections (nested tables).

Example 12–22 Bulk-Fetching into Two Nested Tables

```
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
    ORDER BY last_name;

  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 Abel: $11000
Employee Cambrault: $11000
Employee De Haan: $17000
Employee Errazuriz: $12000
Employee Fripp: $18540.29
Employee Greenberg: $12008
Employee Hartstein: $13000
Employee Higgins: $12008
Employee Kaufling: $17862
Employee King: $24000
Employee Kochhar: $17000
Employee Mourgos: $13113.87
Employee Ozer: $11500
Employee Partners: $13500
Employee Raphaely: $11000
```

```
Employee Russell: $14000
Employee Vishney: $10500
Employee Vollman: $14696.58
Employee Weiss: $22907.66
Employee Zlotkey: $10500
--- Processing 10 rows at a time ---
Result:
Employee Abel: $11000
Employee Cambrault: $11000
Employee De Haan: $17000
Employee Errazuriz: $12000
Employee Fripp: $18540.29
Employee Greenberg: $12008
Employee Hartstein: $13000
Employee Higgins: $12008
Employee Kaufling: $17862
Employee King: $24000
Result:
Employee Kochhar: $17000
Employee Mourgos: $13113.87
Employee Ozer: $11500
Employee Partners: $13500
Employee Raphaely: $11000
Employee Russell: $14000
Employee Vishney: $10500
Employee Vollman: $14696.58
Employee Weiss: $22907.66
Employee Zlotkey: $10500
--- Fetching records rather than columns ---
Employee Abel: $11000
Employee Cambrault: $11000
Employee De Haan: $17000
Employee Errazuriz: $12000
Employee Fripp: $18540.29
Employee Greenberg: $12008
Employee Hartstein: $13000
Employee Higgins: $12008
Employee Kaufling: $17862
Employee King: $24000
Employee Kochhar: $17000
Employee Mourgos: $13113.87
Employee Ozer: $11500
Employee Partners: $13500
Employee Raphaely: $11000
Employee Russell: $14000
Employee Vishney: $10500
Employee Vollman: $14696.58
Employee Weiss: $22907.66
Employee Zlotkey: $10500
```

Example 12–23 uses a `FETCH BULK COLLECT` statement to fetch a result set into a collection (nested table) of records.

Example 12–23 Bulk-Fetching into Nested Table of Records

```
DECLARE
  CURSOR c1 IS
    SELECT first_name, last_name, hire_date
    FROM employees;
```

```
  TYPE NameSet IS TABLE OF c1%ROWTYPE;
  stock_managers  NameSet;  -- nested table of records

  TYPE cursor_var_type is REF CURSOR;
  cv cursor_var_type;

BEGIN
  -- Assign values to nested table of records:

  OPEN cv FOR
    SELECT first_name, last_name, hire_date
    FROM employees
    WHERE job_id = 'ST_MAN'
    ORDER BY hire_date;

  FETCH cv BULK COLLECT INTO stock_managers;
  CLOSE cv;

  -- Print nested table of records:

    FOR i IN stock_managers.FIRST .. stock_managers.LAST LOOP
      DBMS_OUTPUT.PUT_LINE (
        stock_managers(i).hire_date || ' ' ||
        stock_managers(i).last_name  || ', ' ||
        stock_managers(i).first_name
      );
    END LOOP;END;
/
```

Result:

```
01-MAY-03 Kaufling, Payam
18-JUL-04 Weiss, Matthew
10-APR-05 Fripp, Adam
10-OCT-05 Vollman, Shanta
16-NOV-07 Mourgos, Kevin
```

Row Limits for FETCH BULK COLLECT Statements A `FETCH BULK COLLECT` statement that returns a large number of rows produces a large collection. To limit the number of rows and the collection size, use the `LIMIT` clause.

In Example 12–24, with each iteration of the `LOOP` statement, the `FETCH` statement fetches ten rows (or fewer) into associative array `empids` (overwriting the previous values). Note the exit condition for the `LOOP` statement.

Example 12–24 Limiting Bulk FETCH 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
    ORDER BY employee_id;

  empids  numtab;
BEGIN
  OPEN c1;
  LOOP  -- Fetch 10 rows or fewer in each iteration
    FETCH c1 BULK COLLECT INTO empids LIMIT 10;
```

```
    EXIT WHEN empids.COUNT = 0;  -- Not: EXIT WHEN c1%NOTFOUND
    DBMS_OUTPUT.PUT_LINE ('------- Results from One 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 One 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 One 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 One 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 One Bulk Fetch --------
Employee Id: 175
Employee Id: 176
Employee Id: 177
Employee Id: 179
```

RETURNING INTO Clause with BULK COLLECT Clause

The `RETURNING INTO` clause with the `BULK COLLECT` clause (also called the `RETURNING BULK COLLECT INTO` clause) can appear in an `INSERT`, `UPDATE`, `DELETE`, or `EXECUTE IMMEDIATE` statement. With the `RETURNING BULK COLLECT INTO` clause, the statement stores its result set in one or more collections. For more information, see "RETURNING INTO Clause" on page 13-118.

Example 12–25 uses a `DELETE` statement with the `RETURNING BULK COLLECT INTO` clause to delete rows from a table and return them in two collections (nested tables).

Example 12–25 Returning Deleted Rows in Two Nested Tables

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS
SELECT * FROM employees
ORDER BY employee_id;

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 Statement and BULK COLLECT Clause Together

In a `FORALL` statement, the DML statement can have a `RETURNING BULK COLLECT INTO` clause. For each iteration of the `FORALL` statement, the DML statement stores the specified values in the specified collections—without overwriting the previous values, as the same DML statement would do in a `FOR LOOP` statement.

In Example 12–26, the `FORALL` statement runs a `DELETE` statement that has a `RETURNING BULK COLLECT INTO` clause. For each iteration of the `FORALL` statement, the `DELETE` statement stores the `employee_id` and `department_id` values of the deleted row in the collections `e_ids` and `d_ids`, respectively.

Example 12–26 DELETE with RETURN BULK COLLECT INTO in FORALL Statement

```
DROP TABLE emp_temp;
CREATE TABLE emp_temp AS
SELECT * FROM employees
ORDER BY employee_id, department_id;

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
```

Example 12–27 is like Example 12–26 except that it uses a `FOR LOOP` statement instead of a `FORALL` statement.

Example 12–27 DELETE with RETURN BULK COLLECT INTO in FOR LOOP Statement

```
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
  FOR j IN depts.FIRST..depts.LAST LOOP
    DELETE FROM emp_temp
    WHERE department_id = depts(j)
    RETURNING employee_id, department_id
    BULK COLLECT INTO e_ids, d_ids;
  END LOOP;

  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 6 rows:
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
```

Client Bulk-Binding of Host Arrays

Client programs (such as OCI and Pro*C programs) can use PL/SQL anonymous blocks to bulk-bind input and output host arrays. This is the most efficient way to pass collections to and from the database server.

In the client program, declare and assign values to the host variables to be referenced in the anonymous block. In the anonymous block, prefix each host variable name with a colon (:) to distinguish it from a PL/SQL collection variable name. When the client program runs, the database server runs the PL/SQL anonymous block.

In Example 12–28, the anonymous block uses a `FORALL` statement to bulk-bind a host input array. In the `FORALL` statement, the `DELETE` statement refers to four host variables: scalars `lower`, `upper`, and `emp_id` and array `depts`.

Example 12–28 Anonymous Block Bulk-Binds Input Host Array

```
BEGIN
  FORALL i IN :lower..:upper
    DELETE FROM employees
    WHERE department_id = :depts(i);
END;
/
```

Chaining Pipelined Table Functions for Multiple Transformations

Chaining pipelined table functions is an efficient way to perform multiple transformations on data.

> **Note:** You cannot run a pipelined table function 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 Table Functions

- Creating Pipelined Table Functions
- Pipelined Table Functions as Transformation Functions
- Chaining Pipelined Table Functions
- Fetching from Results of Pipelined Table Functions
- Passing CURSOR Expressions to Pipelined Table Functions
- DML Statements on Pipelined Table Function Results
- NO_DATA_NEEDED Exception

Overview of Table Functions

A **table function** is a user-defined PL/SQL function that returns a collection of rows (a nested table or varray). You can select from this collection as if it were a database table by invoking the table function inside the `TABLE` clause in a `SELECT` statement. For example:

```
SELECT * FROM TABLE(table_function_name(parameter_list))
```

(For more information about the `TABLE` clause of the `SELECT` statement, see *Oracle Database SQL Language Reference*.)

A table function can take a collection of rows as input (that is, it can have an input parameter that is a nested table, varray, or cursor variable). Therefore, output from table function `tf1` can be input to table function `tf2`, and output from `tf2` can be input to table function `tf3`, and so on. For more information, see "Chaining Pipelined Table Functions" on page 12-43.

To improve the performance of a table function, you can:

- Enable the function for parallel execution, with the `PARALLEL_ENABLE` option.

  Functions enabled for parallel execution can run concurrently.

- Stream the function results directly to the next process, with Oracle Streams.

  Streaming eliminates intermediate staging between processes. For information about Oracle Streams, see *Oracle Streams Concepts and Administration*.

- Pipeline the function results, with the `PIPELINED` option.

  A **pipelined table function** returns a row to its invoker immediately after processing that row and continues to process rows. Response time improves because the entire collection need not be constructed and returned to the server before the query can return a single result row. (Also, the function needs less memory, because the object cache need not materialize the entire collection.)

Caution: A pipelined table function always references the current state of the data. If the data in the collection changes after the cursor opens for the collection, then the cursor reflects the changes. PL/SQL variables are private to a session and are not transactional. Therefore, read consistency, well known for its applicability to table data, does not apply to PL/SQL collection variables.

See Also: *Oracle Database Data Cartridge Developer's Guide* for information about using pipelined and parallel table functions

Creating Pipelined Table Functions

A pipelined table function must be either a standalone stored function or a package function.

PIPELINED Option (Required)

For a standalone stored function, specify the `PIPELINED` option in the `CREATE FUNCTION` statement (for syntax, see "CREATE FUNCTION Statement" on page 14-32). For a package function, specify the `PIPELINED` option in both the function declaration and function definition (for syntax, see "Function Declaration and Definition" on page 13-83).

PARALLEL_ENABLE Option (Recommended)

To improve its performance, enable the pipelined table function for parallel execution by specifying the `PARALLEL_ENABLE` option.

AUTONOMOUS_TRANSACTION Pragma

If the pipelined table function runs DML statements, then make it autonomous, with the `AUTONOMOUS_TRANSACTION` pragma (described in "AUTONOMOUS_TRANSACTION Pragma" on page 13-6). Then, during parallel execution, each instance of the function creates an independent transaction.

DETERMINISTIC Option (Recommended)

Multiple invocations of a pipelined table function, in either the same query or separate queries, cause multiple executions of the underlying implementation. If the function is deterministic, specify the `DETERMINISTIC` option, described in "DETERMINISTIC" on page 13-84.

Parameters

Typically, a pipelined table function has one or more cursor variable parameters. For information about cursor variables as function parameters, see "Cursor Variables as Subprogram Parameters" on page 6-35.

See Also:

- "Cursor Variables" on page 6-28 for general information about cursor variables
- "Subprogram Parameters" on page 8-9 for general information about subprogram parameters

RETURN Data Type

The data type of the value that a pipelined table function returns must be a collection type defined either at schema level or inside a package (therefore, it cannot be an associative array type). The elements of the collection type must be SQL data types, not data types supported only by PL/SQL (such as `PLS_INTEGER` and `BOOLEAN`). For information about collection types, see "Collection Types" on page 5-2. For information about SQL data types, see *Oracle Database SQL Language Reference*.

You can use SQL data types `ANYTYPE`, `ANYDATA`, and `ANYDATASET` 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 types to create unnamed types, including anonymous collection types. For information about these types, see *Oracle Database PL/SQL Packages and Types Reference*.

PIPE ROW Statement

Inside a pipelined table function, use the `PIPE ROW` statement to return a collection element to the invoker without returning control to the invoker. See "PIPE ROW Statement" on page 13-106 for its syntax and semantics.

RETURN Statement

As in every function, every execution path in a pipelined table function must lead to a `RETURN` statement, which returns control to the invoker. However, in a pipelined table function, a `RETURN` statement need not return a value to the invoker. See "RETURN Statement" on page 13-116 for its syntax and semantics.

Example

Example 12–29 creates a package that includes a pipelined table function, `f1`, and then selects from the collection of rows that `f1` returns.

Example 12–29 Creating and Invoking Pipelined 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;
/

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

Result:

```
COLUMN_VALUE
------------
           1
           2
           3
           4
           5

5 rows selected.
```

Pipelined Table Functions as Transformation Functions

A pipelined table function with a cursor variable parameter can serve as a transformation function. Using the cursor variable, the function fetches an input row. Using the `PIPE ROW` statement, the function pipes the transformed row or rows to the invoker. If the `FETCH` and `PIPE ROW` statements are inside a `LOOP` statement, the function can transform multiple input rows.

In Example 12–30, the pipelined table function transforms each selected row of the `employees` table to two nested table rows, which it pipes to the `SELECT` statement that invokes it. The actual parameter that corresponds to the formal cursor variable parameter is a `CURSOR` expression; for information about these, see "Passing CURSOR Expressions to Pipelined Table Functions" on page 12-44.

Example 12–30 Pipelined Table Function Transforms Each Row to Two Rows

```
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;  -- input row
      EXIT WHEN p%NOTFOUND;

      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);  -- first transformed output row

      out_rec.var_char1 := in_rec.email;
      out_rec.var_char2 := in_rec.phone_number;
      PIPE ROW(out_rec);  -- second transformed output row
    END LOOP;
    CLOSE p;
    RETURN;
  END f_trans;
END refcur_pkg;
/

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

10 rows selected.
```

Chaining Pipelined Table Functions

To **chain** pipelined table functions `tf1` and `tf2` is to make the output of `tf1` the input of `tf2`. For example:

```
SELECT * FROM TABLE(tf2(CURSOR(SELECT * FROM TABLE(tf1()))));
```

The rows that `tf1` pipes out must be compatible actual parameters for the formal input parameters of `tf2`.

If chained pipelined table functions are enabled for parallel execution, then each function runs in a different process (or set of processes).

> **See Also:** "Passing CURSOR Expressions to Pipelined Table Functions" on page 12-44

Fetching from Results of Pipelined Table Functions

You can associate a named cursor with a query that invokes a pipelined table function. Such a cursor has no special fetch semantics, and such a cursor variable has no special assignment semantics.

However, the SQL optimizer does not optimize across PL/SQL statements. Therefore, in Example 12–31, the first PL/SQL statement is slower than the second—despite the overhead of running two SQL statements in the second PL/SQL statement, and even if function results are piped between the two SQL statements in the first PL/SQL statement.

In Example 12–31, assume that `f` and `g` are pipelined table functions, and that each function accepts a cursor variable parameter. The first PL/SQL statement associates cursor variable `r` with a query that invokes `f`, and then passes `r` to `g`. The second PL/SQL statement passes `CURSOR` expressions to both `f` and `g`.

Example 12–31 Fetching from Results of Pipelined Table Functions

```
DECLARE
  r SYS_REFCURSOR;
  ...
  -- First PL/SQL statement (slower):
BEGIN
  OPEN r FOR SELECT * FROM TABLE(f(CURSOR(SELECT * FROM tab)));
  SELECT * BULK COLLECT INTO rec_tab FROM TABLE(g(r));

  -- NOTE: When g completes, it closes r.
END;

-- Second PL/SQL statement (faster):

SELECT * FROM TABLE(g(CURSOR(SELECT * FROM
  TABLE(f(CURSOR(SELECT * FROM tab))))));
/
```

See Also: "Cursor Variables as Subprogram Parameters" on page 6-35

Passing CURSOR Expressions to Pipelined Table Functions

As Example 12–31 shows, the actual parameter for the cursor variable parameter of a pipelined table function can be either a cursor variable or a CURSOR expression, and the latter is more efficient.

Note: When a SQL SELECT statement passes a CURSOR expression to a function, the referenced cursor opens when the function begins to run and closes when the function completes.

See Also: "CURSOR Expressions" on page 6-38 for general information about CURSOR expressions

Example 12–32 creates a package that includes a pipelined table function with two cursor variable parameters and then invokes the function in a SELECT statement, using CURSOR expressions for actual parameters.

Example 12–32 Pipelined Table 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;
/

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.
```

Example 12–33 uses a pipelined table function as an aggregate function, which takes a set of input rows and returns a single result. The `SELECT` statement selects the function result. (For information about the pseudocolumn `COLUMN_VALUE`, see *Oracle Database SQL Language Reference*.)

Example 12–33 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
    LOOP
      FETCH input_values INTO weight, grade;
      EXIT WHEN input_values%NOTFOUND;
      total_weight := total_weight + weight;  -- Accumulate weighted average
      total := total + grade*weight;
    END LOOP;
    PIPE ROW (total / total_weight);
    RETURN; -- returns single result
  END weighted_average;
END pkg_gpa;
/

SELECT w.column_value "weighted result" FROM TABLE (
  pkg_gpa.weighted_average (
    CURSOR (SELECT weight, grade FROM gradereport)
  )
) w;
```

Result:

```
weighted result
---------------
            3.5

1 row selected.
```

DML Statements on Pipelined Table Function Results

The "table" that a pipelined table function returns cannot be the target table of a `DELETE`, `INSERT`, or `UPDATE` statement. However, you can create a view of such a table and create `INSTEAD OF` triggers on the view. For information about `INSTEAD OF` triggers, see "INSTEAD OF Triggers" on page 9-10.

> **See Also:** *Oracle Database SQL Language Reference* for information about the `CREATE VIEW` statement

NO_DATA_NEEDED Exception

You must understand the predefined exception `NO_DATA_NEEDED` in two cases:

- You include an OTHERS exception handler in a block that includes a PIPE ROW statement
- Your code that feeds a PIPE ROW statement must be followed by a clean-up procedure

  Typically, the clean-up procedure releases resources that the code no longer needs.

When the invoker of a pipelined table function needs no more rows from the function, the PIPE ROW statement raises NO_DATA_NEEDED. If the pipelined table function does not handle NO_DATA_NEEDED, as in Example 12–34, then the function invocation terminates but the invoking statement does not terminate. If the pipelined table function handles NO_DATA_NEEDED, its exception handler can release the resources that it no longer needs, as in Example 12–35.

In Example 12–34, the pipelined table function pipe_rows does not handle the NO_DATA_NEEDED exception. The SELECT statement that invokes pipe_rows needs only four rows. Therefore, during the fifth invocation of pipe_rows, the PIPE ROW statement raises the exception NO_DATA_NEEDED. The fifth invocation of pipe_rows terminates, but the SELECT statement does not terminate.

Example 12–34 Pipelined Table Function Does Not Handle NO_DATA_NEEDED

```
CREATE TYPE t IS TABLE OF NUMBER
/
CREATE OR REPLACE FUNCTION pipe_rows RETURN t PIPELINED IS
  n NUMBER := 0;
BEGIN
  LOOP
    n := n + 1;
    PIPE ROW (n);
  END LOOP;
END pipe_rows;
/
SELECT COLUMN_VALUE
  FROM TABLE(pipe_rows())
  WHERE ROWNUM < 5
/
```

Result:

```
COLUMN_VALUE
------------
           1
           2
           3
           4

4 rows selected.
```

If the exception-handling part of a block that includes a PIPE ROW statement includes an OTHERS exception handler to handle unexpected exceptions, then it must also include an exception handler for the expected NO_DATA_NEEDED exception. Otherwise, the OTHERS exception handler handles the NO_DATA_NEEDED exception, treating it as an unexpected error. The following exception handler reraises the NO_DATA_NEEDED exception, instead of treating it as a irrecoverable error:

```
EXCEPTION
  WHEN NO_DATA_NEEDED THEN
    RAISE;
  WHEN OTHERS THEN
```

```
    -- (Put error-logging code here)
    RAISE_APPLICATION_ERROR(-20000, 'Fatal error.');
END;
```

In Example 12–35, assume that the package `External_Source` contains these public items:

- Procedure `Init`, which allocates and initializes the resources that `Next_Row` needs
- Function `Next_Row`, which returns some data from a specific external source and raises the user-defined exception `Done` (which is also a public item in the package) when the external source has no more data
- Procedure `Clean_Up`, which releases the resources that `Init` allocated

The pipelined table function `get_external_source_data` pipes rows from the external source by invoking `External_Source.Next_Row` until either:

- The external source has no more rows.

  In this case, the `External_Source.Next_Row` function raises the user-defined exception `External_Source.Done`.
- `get_external_source_data` needs no more rows.

  In this case, the `PIPE ROW` statement in `get_external_source_data` raises the `NO_DATA_NEEDED` exception.

In either case, an exception handler in block `b` in `get_external_source_data` invokes `External_Source.Clean_Up`, which releases the resources that `Next_Row` was using.

Example 12–35 Pipelined Table Function Handles NO_DATA_NEEDED

```
CREATE OR REPLACE FUNCTION get_external_source_data
  RETURN t AUTHID DEFINER PIPELINED IS
BEGIN
  External_Source.Init();           -- Initialize.
  <<b>> BEGIN
    LOOP                            -- Pipe rows from external source.
      PIPE ROW (External_Source.Next_Row());
    END LOOP;
  EXCEPTION
    WHEN External_Source.Done THEN  -- When no more rows are available,
      External_Source.Clean_Up();   --  clean up.
    WHEN NO_DATA_NEEDED THEN        -- When no more rows are needed,
      External_Source.Clean_Up();   --  clean up.
      RAISE NO_DATA_NEEDED;           -- Optional, equivalent to RETURN.
  END b;
END get_external_source_data;
/
```

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

Collecting Data About User-Defined Identifiers

PL/Scope extracts, organizes, and stores data about user-defined identifiers from PL/SQL source text. You can retrieve source text 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. You must have `CREATE` privileges on the units to be profiled. Saves run-time statistics in database tables, which you can query.
Trace API	`DBMS_TRACE`	Traces the order in which subprograms run. You can specify the subprograms to trace and the tracing level. 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 invocations. Accounts for SQL and PL/SQL execution times separately. Requires no special source or compile-time preparation. 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 12-50).

For detailed information about the `DBMS_TRACE` subprograms, see *Oracle Database PL/SQL Packages and Types Reference*.

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 the PL/SQL packages that Oracle Database supplies.

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 uses only 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 invoke 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 invoke each other.

PL/SQL native compilation works transparently in an 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 1-10.

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 1-10.

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 the PL/SQL packages that Oracle Database supplies, 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 explained in this section. Before making the conversion, review "Determining Whether to Use PL/SQL Native Compilation" on page 12-51.

> **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 *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 *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;
    ```

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
- Basic LOOP Statement
- Block
- CASE Statement
- CLOSE Statement
- Collection Variable Declaration
- Collection Method Invocation
- Comment
- Constant Declaration
- CONTINUE Statement
- Cursor FOR LOOP Statement
- Cursor Variable Declaration
- DELETE Statement Extension
- EXCEPTION_INIT Pragma
- Exception Declaration
- Exception Handler
- EXECUTE IMMEDIATE Statement
- EXIT Statement
- Explicit Cursor Declaration and Definition
- Expression
- FETCH Statement
- FOR LOOP Statement

- FORALL Statement
- Formal Parameter Declaration
- Function Declaration and Definition
- GOTO Statement
- IF Statement
- Implicit Cursor Attribute
- INLINE Pragma
- INSERT Statement Extension
- Named Cursor Attribute
- NULL Statement
- OPEN Statement
- OPEN FOR Statement
- PIPE ROW Statement
- Procedure Declaration and Definition
- RAISE Statement
- Record Variable Declaration
- RESTRICT_REFERENCES Pragma (deprecated)
- RETURN Statement
- RETURNING INTO Clause
- %ROWTYPE Attribute
- Scalar Variable Declaration
- SELECT INTO Statement
- SERIALLY_REUSABLE Pragma
- SQLCODE Function
- SQLERRM Function
- %TYPE Attribute
- UPDATE Statement Extensions
- WHILE LOOP Statement

See Also: Chapter 2, "PL/SQL Language Fundamentals"

Assignment Statement

The assignment statement sets the value of a data item to a valid value.

Topics

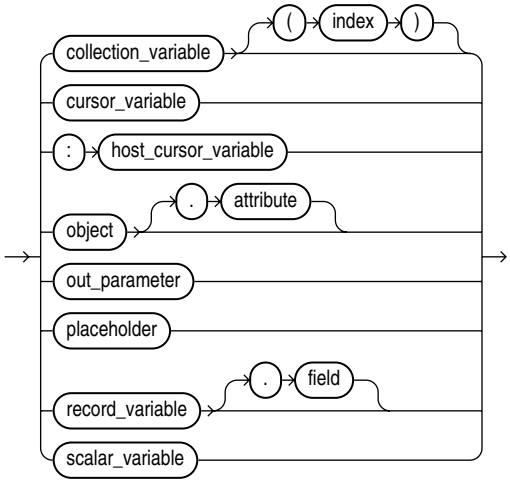
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

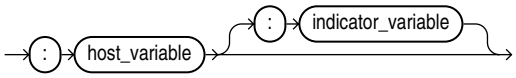
assignment_statement ::=

See "expression ::=" on page 13-61.

assignment_statement_target ::=



placeholder ::=



Semantics

assignment_statement_target

Data item to which the value of `expression` is to be assigned.

collection_variable

Name of a collection variable.

index

Index of an element of *collection_variable*. Without *index*, the entire collection variable is the assignment statement target.

index must be a numeric expression whose data type either is PLS_INTEGER or can be implicitly converted to PLS_INTEGER (for information about the latter, see "Predefined PLS_INTEGER Subtypes" on page 3-9).

cursor_variable

Name of a cursor variable.

:*host_cursor_variable*

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*.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

object

Name of an instance of an abstract data type (ADT).

attribute

Name of an attribute of *object*. Without *attribute*, the entire ADT is the assignment statement target.

out_parameter

Name of a formal OUT or IN OUT parameter of the subprogram in which the assignment statement appears.

record_variable

Name of a record variable.

field

Name of a field of *record_variable*. Without *field*, the entire record variable is the assignment statement target.

scalar_variable

Name of a PL/SQL scalar variable.

placeholder

:*host_variable*

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*.

:*indicator_variable*

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* and the colon (:) or between the colon and *indicator_variable*. This is correct:

`:host_variable:indicator_variable`

expression

Expression whose value is to be assigned to *assignment_statement_target*.

expression and *assignment_statement_target* must have compatible data types.

> **Note:** Collections with elements of the same type might not have the same data type. For the syntax of collection type definitions, see "Collection Variable Declaration" on page 13-25.

Examples

- Example 2–24, "Assigning Values to Variables with Assignment Statement" on page 2-23
- Example 2–27, "Assigning Value to BOOLEAN Variable" on page 2-23
- Example 5–7, "Data Type Compatibility for Collection Assignment" on page 5-14

Related Topics

In this chapter:

- "Expression" on page 13-61
- "FETCH Statement" on page 13-71
- "SELECT INTO Statement" on page 13-125

In other chapters:

- "Assigning Values to Variables" on page 2-20
- "Assigning Values to Collection Variables" on page 5-14
- "Assigning Values to Record Variables" on page 5-44

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:

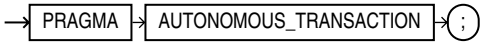
- Schema-level (not nested) anonymous PL/SQL block
- Standalone, package, or nested subprogram
- Method of an ADT
- Noncompound trigger

Topics

- Syntax
- Examples
- Related Topics

Syntax

***autonomous_transaction_pragma* ::=**



Examples

- Example 6–44, "Declaring Autonomous Function in Package" on page 6-51
- Example 6–45, "Declaring Autonomous Standalone Procedure" on page 6-51
- Example 6–46, "Declaring Autonomous PL/SQL Block" on page 6-52
- Example 6–47, "Autonomous Trigger Logs INSERT Statements" on page 6-54
- Example 6–48, "Autonomous Trigger Uses Native Dynamic SQL for DDL" on page 6-55
- Example 6–49, "Invoking Autonomous Function" on page 6-56

Related Topics

In this chapter:

- "EXCEPTION_INIT Pragma" on page 13-46
- "INLINE Pragma" on page 13-94
- "RESTRICT_REFERENCES Pragma" on page 13-114
- "SERIALLY_REUSABLE Pragma" on page 13-130

In other chapters:

- "Autonomous Transactions" on page 6-49

Basic LOOP Statement

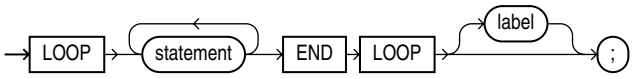
With each iteration of the basic `LOOP` statement, its statements run and control returns to the top of the loop. The `LOOP` statement ends when a statement inside the loop transfers control outside the loop or raises an exception.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***basic_loop_statement* ::=**



See "statement ::=" on page 13-13.

Semantics

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 13-38 (when it transfers control to the next iteration of an enclosing labeled loop)
- "EXIT Statement" on page 13-55
- "GOTO Statement" on page 13-87
- "RAISE Statement" on page 13-110

label

A label that identifies *`basic_loop_statement`* (see "statement ::=" on page 13-13 and "label" on page 13-16). `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 ine the `END LOOP` statement matches a label at the beginning of the same `LOOP` statement (the compiler does not check).

Examples

- Example 1–2, "Processing Query Result Rows One at a Time" on page 1-9
- 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 13-40
- "FOR LOOP Statement" on page 13-74
- "WHILE LOOP Statement" on page 13-138

In other chapters:

- "Basic LOOP Statement" on page 4-9

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.

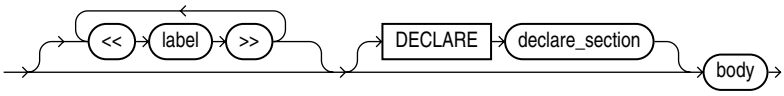
An anonymous block is an executable statement.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

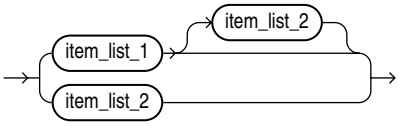
Syntax

plsql_block ::=



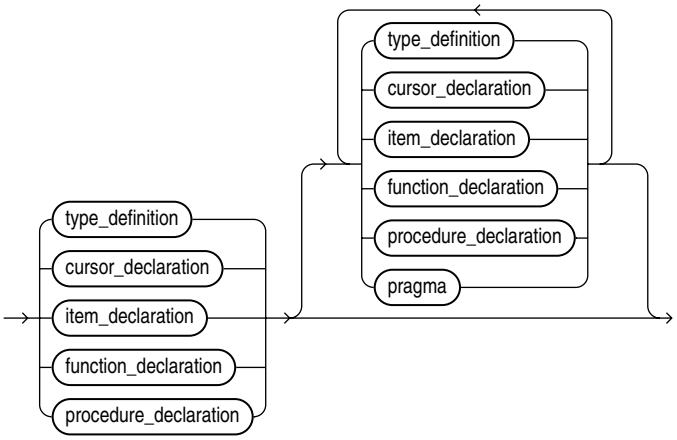
See "body ::=" on page 13-12.

declare_section ::=



See "item_list_2 ::=" on page 13-10.

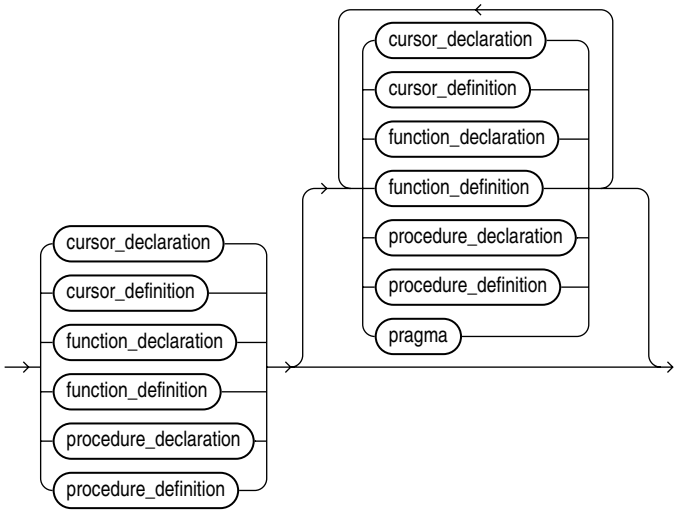
item_list_1 ::=



See:

- "cursor_declaration ::=" on page 13-57
- "function_declaration ::=" on page 13-83
- "item_declaration ::=" on page 13-11
- "procedure_declaration ::=" on page 13-108
- "pragma ::=" on page 13-11
- "type_definition ::=" on page 13-10

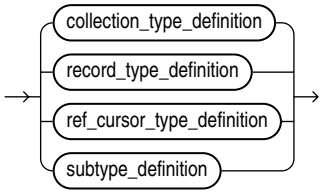
***item_list_2* ::=**



See:

- "cursor_declaration ::=" on page 13-57
- "cursor_definition ::=" on page 13-57
- "function_declaration ::=" on page 13-83
- "function_definition ::=" on page 13-84
- "pragma ::=" on page 13-11
- "procedure_declaration ::=" on page 13-108
- "procedure_definition ::=" on page 13-108
- "type_definition ::=" on page 13-10

***type_definition* ::=**

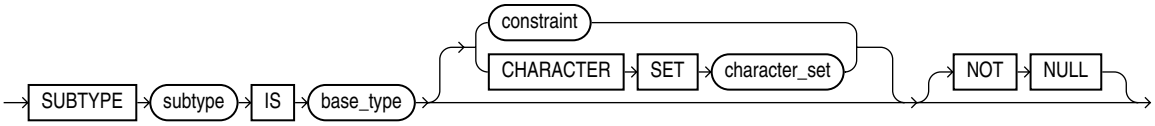


See:

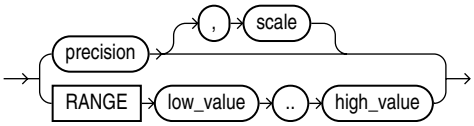
- "collection_type_definition ::=" on page 13-25
- "record_type_definition ::=" on page 13-111

- "ref_cursor_type_definition ::=" on page 13-42
- "subtype_definition ::=" on page 13-11

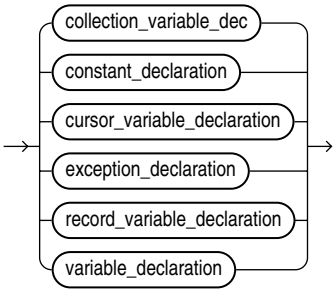
subtype_definition ::=



constraint ::=



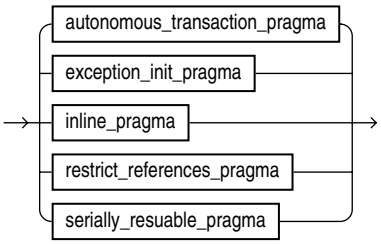
item_declaration ::=



See:

- "collection_variable_dec ::=" on page 13-27
- "constant_declaration ::=" on page 13-36
- "cursor_declaration ::=" on page 13-57
- "cursor_variable_declaration ::=" on page 13-42
- "exception_declaration ::=" on page 13-48
- "record_variable_declaration ::=" on page 13-111
- "variable_declaration ::=" on page 13-123

pragma ::=

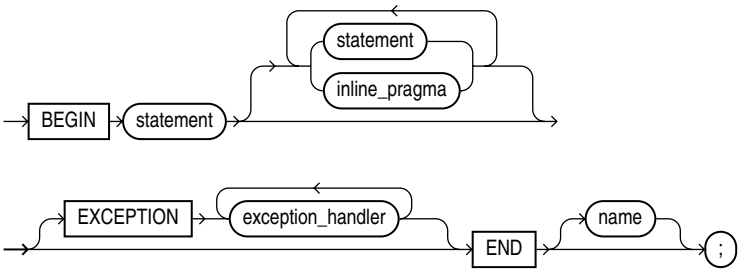


See:

- "autonomous_transaction_pragma ::=" on page 13-6

- "exception_init_pragma ::=" on page 13-46
- "inline_pragma ::=" on page 13-94
- "restrict_references_pragma ::=" on page 13-114
- "serially_resuable_pragma ::=" on page 13-130

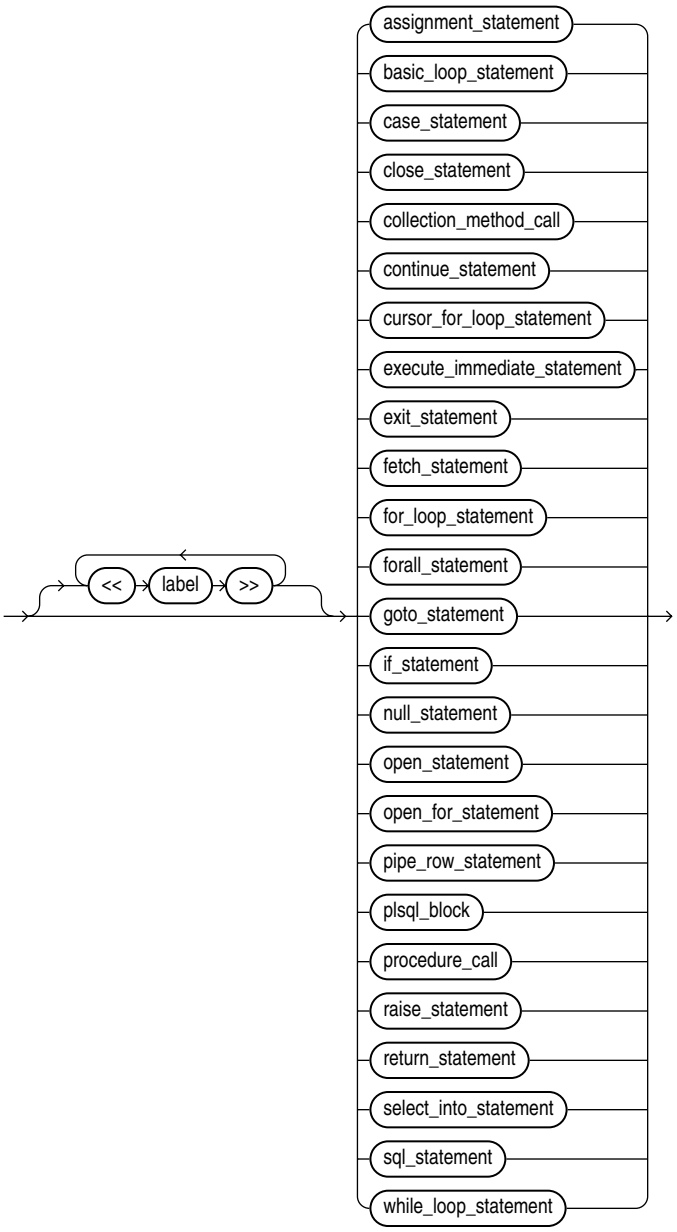
***body* ::=**



See:

- "exception_handler ::=" on page 13-50
- "inline_pragma ::=" on page 13-94

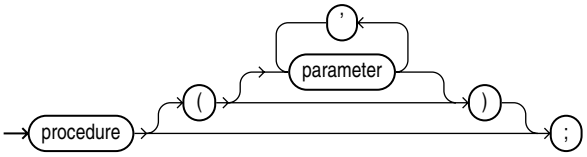
***statement* ::=**



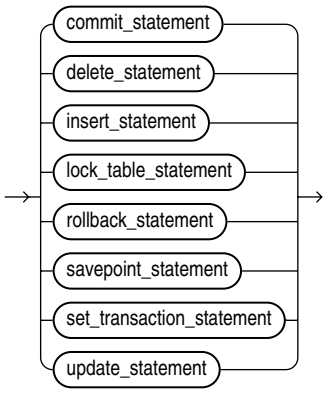
See:

- "plsql_block ::=" on page 13-9
- "procedure_call ::=" on page 13-13
- "sql_statement ::=" on page 13-14

***procedure_call* ::=**



***sql_statement* ::=**



Semantics

plsql_block

label

Undeclared identifier, unique for the block.

DECLARE

Starts the declarative part of the block.

declare_section

Contains local declarations, which exist only in the block and its sub-blocks and are not visible to enclosing blocks.

Restrictions on *declare_section*

- A *declare_section* in *create_package*, *create_package_body*, or *compound_trigger_block* cannot include PRAGMA AUTONOMOUS_TRANSACTION.
- A *declare_section* in *trigger_body* or *tps_body* cannot declare variables of the data type LONG or LONG RAW.

See Also:

- "CREATE PACKAGE Statement" on page 14-43 for more information about *create_package*
- "CREATE PACKAGE BODY Statement" on page 14-46 for more information about *create_package_body*
- "CREATE TRIGGER Statement" on page 14-54 for more information about *compound_trigger_block*, *trigger_body*, and *tps_body*

subtype_definition

subtype

Name of the user-defined subtype that you are defining.

base_type

Base type of the subtype that you are defining. *base_type* must be scalar.

CHARACTER SET *character_set*

Specifies the character set for a subtype of a character data type.

Restriction on CHARACTER SET *character_set* Do not specify this clause if *base_type* is not a character data type.

NOT NULL

Imposes the NOT NULL constraint on data items declared with this subtype. For information about this constraint, see "NOT NULL Constraint" on page 2-13.

constraint

Specifies a constraint for a subtype of a numeric data type.

Restriction on *constraint* Do not specify *constraint* if *base_type* is not a numeric data type.

***precision*]**

Specifies the precision for a constrained subtype of a numeric data type.

Restriction on *precision* Do not specify *precision* if *base_type* cannot specify precision.

scale

Specifies the scale for a constrained subtype of a numeric data type.

Restriction on *scale* Do not specify *scale* if *base_type* cannot specify scale.

RANGE *low_value .. high_value*

Specifies the range for a constrained subtype of a numeric data type. The *low_value* and *high_value* must be numeric literals.

Restriction on RANGE *high_value .. low_value* Specify this clause only if *base_type* can specify a range; that is, if *base_type* is a subtype of PLS_INTEGER. (Table 3–3 summarizes the predefined subtypes of PLS_INTEGER.)

body

BEGIN

Starts the executable part of the block, which contains executable statements.

EXCEPTION

Starts the exception-handling part of the block. When PL/SQL raises an exception, 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 13-50.

END

Ends the block.

name

The name of the block to which `END` applies—a label, function name, procedure name, or package name.

statement

label

Undeclared identifier, unique for the statement.

assignment_statement

See "Assignment Statement" on page 13-3.

basic_loop_statement

See "Basic LOOP Statement" on page 13-7.

case_statement

See "CASE Statement" on page 13-20.

close_statement

See "CLOSE Statement" on page 13-23.

collection_method_call

Invocation of one of these collection methods, which are procedures:

- `DELETE`
- `EXTEND`
- `TRIM`

For syntax, see "Collection Method Invocation" on page 13-31.

continue_statement

See "CONTINUE Statement" on page 13-38.

cursor_for_loop_statement

See "Cursor FOR LOOP Statement" on page 13-40.

execute_immediate_statement

See "EXECUTE IMMEDIATE Statement" on page 13-52.

exit_statement

See "EXIT Statement" on page 13-55.

fetch_statement

See "FETCH Statement" on page 13-71.

for_loop_statement

See "FOR LOOP Statement" on page 13-74.

forall_statement
See "FORALL Statement" on page 13-77.

goto_statement
See "GOTO Statement" on page 13-87.

if_statement
See "IF Statement" on page 13-89.

null_statement
See "NULL Statement" on page 13-100.

open_statement
See "OPEN Statement" on page 13-101.

open_for_statement
See "OPEN FOR Statement" on page 13-103.

pipe_row_statement
See "PIPE ROW Statement" on page 13-106.

Restriction on *pipe_row_statement* This statement can appear only in the body of a pipelined table function; otherwise, PL/SQL raises an exception.

raise_statement
See "RAISE Statement" on page 13-110.

return_statement
See "RETURN Statement" on page 13-116.

select_into_statement
See "SELECT INTO Statement" on page 13-125.

while_loop_statement
See "WHILE LOOP Statement" on page 13-138.

procedure_call

procedure
Name of the procedure that you are invoking.

***parameter* [, *parameter*]...**
List of actual parameters for the procedure that you are invoking. The data type of each actual parameter must be compatible with the data type of the corresponding formal parameter. The mode of the formal parameter determines what the actual parameter can be:

Formal Parameter Mode	Actual Parameter
`IN`	Constant, initialized variable, literal, or expression

Formal Parameter Mode	Actual Parameter
OUT	Variable whose data type is not defined as NOT NULL
IN OUT	Variable (typically, it is a string buffer or numeric accumulator)

If the procedure specifies a default value for a parameter, you can omit that parameter from the parameter list. If the procedure has no parameters, or specifies a default 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 8-21

sql_statement

commit_statement

SQL COMMIT statement. For syntax, see *Oracle Database SQL Language Reference*.

delete_statement

SQL DELETE statement. For syntax, see *Oracle Database SQL Language Reference*. See also "DELETE Statement Extension" on page 13-45.

insert_statement

SQL INSERT statement. For syntax, see *Oracle Database SQL Language Reference*. See also "INSERT Statement Extension" on page 13-96.

lock_table_statement

SQL LOCK TABLE statement. For syntax, see *Oracle Database SQL Language Reference*.

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*.

set_transaction_statement

SQL SET TRANSACTION statement. For syntax, see *Oracle Database SQL Language Reference*.

update_statement

SQL UPDATE statement. For syntax, see *Oracle Database SQL Language Reference*. See also "UPDATE Statement Extensions" on page 13-136.

Examples

- Example 1–1, "PL/SQL Block Structure" on page 1-5
- Example 2–23, "Block with Multiple and Duplicate Labels" on page 2-20
- Example 4–29, "Incorrect Label Placement" on page 4-21

Related Topics

In this chapter:

- "Comment" on page 13-34

In other chapters:

- "Blocks" on page 1-5
- "Identifiers" on page 2-4
- "Pragmas" on page 2-41
- Chapter 3, "PL/SQL Data Types"
- "User-Defined PL/SQL Subtypes" on page 3-11

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.

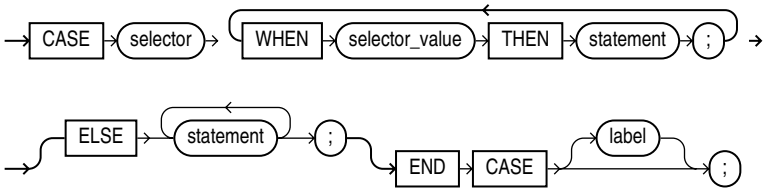
The searched CASE statement evaluates multiple Boolean expressions and chooses the first one whose value is TRUE.

Topics

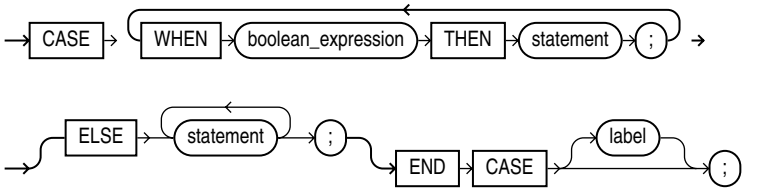
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

simple_case_statement ::=



searched_case_statement ::=



See:

- "boolean_expression ::=" on page 13-62
- "statement ::=" on page 13-13

Semantics

simple_case_statement

selector

Expression whose value is evaluated once and used to select one of several alternatives. *selector* can have any PL/SQL data type except BLOB, BFILE, or a user-defined type.

WHEN *selector_value* THEN *statement*

selector_value can be an expression of any PL/SQL type except BLOB, BFILE, or a user-defined type.

The *selector_value*s are evaluated sequentially. If the value of a *selector_value* equals the value of *selector*, then the *statement* associated with that *selector_value* runs, and the CASE statement ends. Subsequent *selector_value*s are not evaluated.

> **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*]...

The *statement*s run if and only if no *selector_value* has the same value as *selector*.

Without the ELSE clause, if no *selector_value* has the same value as *selector*, the system raises the predefined exception CASE_NOT_FOUND.

label

A label that identifies the statement (see "statement ::=" on page 13-13 and "label" on page 13-16).

searched_case_statement

WHEN *boolean_expression* THEN *statement*

The *boolean_expression*s are evaluated sequentially. If the value of a *boolean_expression* is TRUE, the *statement* associated with that *boolean_expression* runs, and the CASE statement ends. Subsequent *boolean_expression*s are not evaluated.

> **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*]...

The *statement*s run if and only if no *boolean_expression* has the value TRUE.

Without the ELSE clause, if no *boolean_expression* has the value TRUE, the system raises the predefined exception CASE_NOT_FOUND.

label

A label that identifies the statement (see "statement ::=" on page 13-13 and "label" on page 13-16).

Examples

- Example 3–2, "Printing BOOLEAN Values" on page 3-7
- 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 13-89

In other chapters:

- "CASE Expressions" on page 2-37
- "Conditional Selection Statements" on page 4-1
- "Simple CASE Statement" on page 4-6
- "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, freeing its resources for reuse.

After closing an explicit cursor, you can reopen it with the `OPEN` statement. You must close an explicit cursor before reopening it.

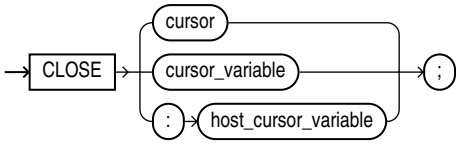
After closing a cursor variable, you can reopen it with the `OPEN FOR` statement. You need not close a cursor variable before reopening it.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***close_statement* ::=**



Semantics

cursor

Name of an open explicit cursor.

cursor_variable

Name of an open cursor variable.

:host_cursor_variable

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`*.

Examples

- Example 6–6, "FETCH Statements Inside LOOP Statements" on page 6-10

Related Topics

In this chapter:

- "FETCH Statement" on page 13-71
- "OPEN Statement" on page 13-101
- "OPEN FOR Statement" on page 13-103

In other chapters:

- "Opening and Closing Explicit Cursors" on page 6-9
- "Opening and Closing Cursor Variables" on page 6-30

Collection Variable Declaration

A **collection variable** is a composite variable whose internal components, called elements, have the same data type. The value of a collection variable and the values of its elements can change.

You reference an entire collection by its name. You reference a collection element with the syntax `collection(subscript)`.

PL/SQL has three kinds of collection types:

- Associative array (formerly called *PL/SQL table* or *index-by table*)
- Variable-size array (varray)
- Nested table

Associative arrays can be indexed by either integers or strings. Varrays and nested tables are indexed by integers.

You can create a collection variable in either of these ways:

- Define a collection type and then declare a variable of that type.
- Use `%TYPE` to declare a collection variable of the same type as a previously declared collection variable.

Note: This topic applies to collection types that you define inside a PL/SQL block or package, which differ from standalone stored collection types that you create with the "CREATE TYPE Statement" on page 14-68.

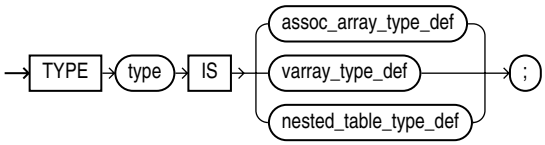
In a PL/SQL block or package, you can define all three collection types. With the `CREATE TYPE` statement, you can create nested table types and `VARRAY` types, but not associative array types.

Topics

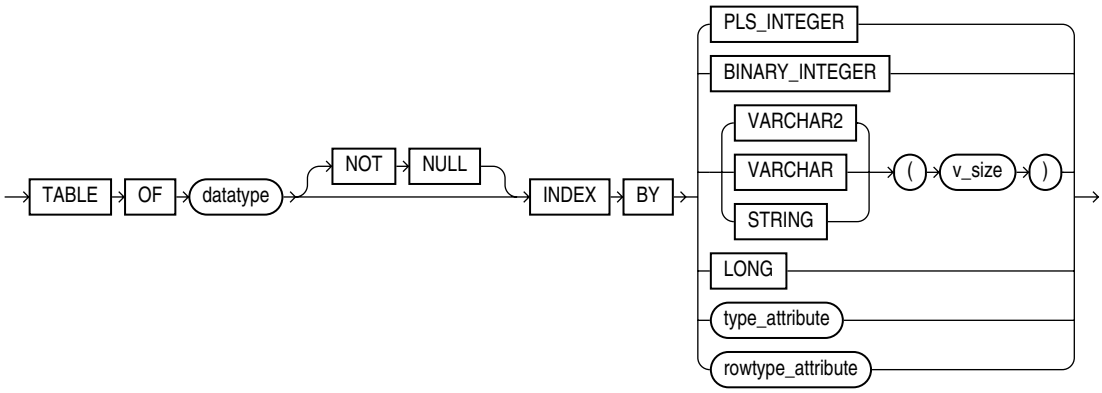
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***collection_type_definition* ::=**



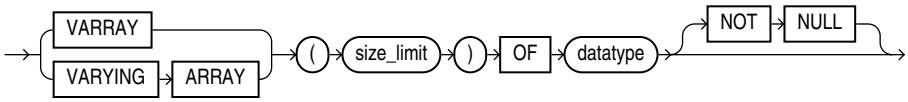
assoc_array_type_def ::=



See:

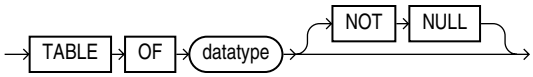
- "datatype ::=" on page 13-26
- "rowtype_attribute ::=" on page 13-121
- "type_attribute ::=" on page 13-134

varray_type_def ::=

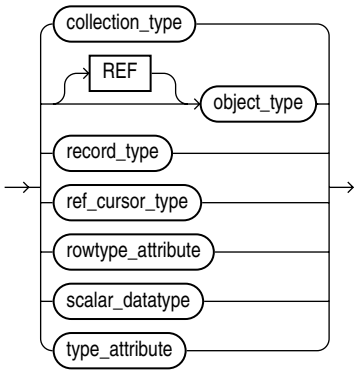


See "datatype ::=" on page 13-26.

nested_table_type_def ::=



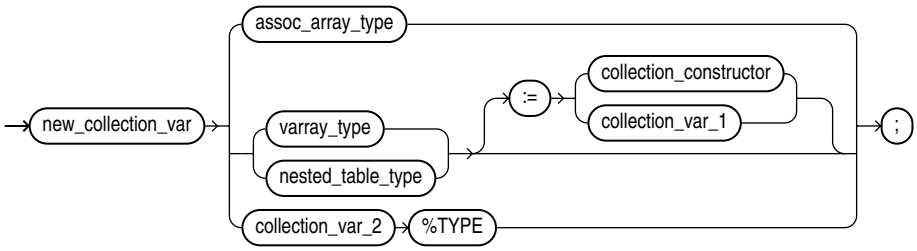
datatype ::=



See:

- "rowtype_attribute ::=" on page 13-121
- "type_attribute ::=" on page 13-134

***collection_variable_dec* ::=**



See "collection_constructor ::=" on page 13-63.

Semantics

collection_type_definition

type

Name of the collection type that you are defining.

assoc_array_type_def

Type definition for an associative array.

Restriction on *assoc_array_type_def* Can appear only in the declarative part of a block, subprogram, package specification, or package body.

nested_table_type_def

Type definition for a nested table.

varray_type_def

Type definition for a variable-size array.

assoc_array_type_def

datatype

Data type of the elements of the associative array. *datatype* can be any PL/SQL data type except `REF CURSOR`.

NOT NULL

Imposes the `NOT NULL` constraint on every element of the associative array. For information about this constraint, see "NOT NULL Constraint" on page 2-13.

{ PLS_INTEGER | BINARY_INTEGER }

Specifies that the data type of the indexes of the associative array is `PLS_INTEGER`.

{ VARCHAR2 | VARCHAR | STRING } (*v_size*)

Specifies that the data type of the indexes of the associative array is `VARCHAR2` (or its subtype `VARCHAR` or `STRING`) with length *v_size*.

You can populate an element of the associative array with a value of any type that can be converted to `VARCHAR2` with the `TO_CHAR` function (described in *Oracle Database SQL Language Reference*).

> **Caution:** Associative arrays indexed by strings can be affected by National Language Support (NLS) parameters. For more information, see "NLS Parameter Values Affect Associative Arrays Indexed by String" on page 5-6.

LONG

Specifies that the data type of the indexes of the associative array is `LONG`, which is equivalent to `VARCHAR2(32760)`.

> **Note:** Oracle supports `LONG` only for backward compatibility with existing applications. For new applications, use `VARCHAR2(32760)`.

type_attribute, rowtype_attribute

Specifies that the data type of the indexes of the associative array is a data type specified with either `%ROWTYPE` or `%TYPE`. This data type must represent either `PLS_INTEGER`, `BINARY_INTEGER`, or `VARCHAR2(v_size)`.

varray_type_def

size_limit

Maximum number of elements that the varray can have. *`size_limit`* must be an integer literal in the range from 1 through 2147483647.

datatype

Data type of the varray element. *`datatype`* can be any PL/SQL data type except `REF CURSOR`.

NOT NULL

Imposes the `NOT NULL` constraint on every element of the varray. For information about this constraint, see "NOT NULL Constraint" on page 2-13.

nested_table_type_def

datatype

Data type of the elements of the nested table. *`datatype`* can be any PL/SQL data type except `REF CURSOR` or `NCLOB`.

If *`datatype`* is a scalar type, then the nested table has a single column of that type, called `COLUMN_VALUE`.

If *`datatype`* is an ADT, then the columns of the nested table match the name and attributes of the ADT.

NOT NULL

Imposes the `NOT NULL` constraint on every element of the nested table. For information about this constraint, see "NOT NULL Constraint" on page 2-13.

datatype

collection_type

Name of a user-defined varray or nested table type (not the name of an associative array type).

object_type

Instance of a user-defined type.

record_type

Name of a user-defined type that was defined with the data type specifier `RECORD`.

ref_cursor_type

Name of a user-defined type that was defined with the data type specifier `REF CURSOR`.

scalar_datatype

Name of a scalar data type, including any qualifiers for size, precision, and character or byte semantics.

collection_variable_dec

new_collection_var

Name of the collection variable that you are declaring.

assoc_array_type

Name of a previously defined associative array type; the data type of *`new_collection_var`*.

varray_type

Name of a previously defined `VARRAY` type; the data type of *`new_collection_var`*.

nested_table_type

Name of a previously defined nested table type; the data type of *`new_collection_var`*.

collection_constructor

Collection constructor for the data type of *`new_collection_var`*, which provides the initial value of *`new_collection_var`*.

collection_var_1

Name of a previously declared collection variable of the same data type as *`new_collection_var`*, which provides the initial value of *`new_collection_var`*.

> **Note:** *`collection_var_1`* and *`new_collection_var`* must have the same data type, not only elements of the same type.

collection_var_2

Name of a previously declared collection variable.

%TYPE

See "%TYPE Attribute" on page 13-134.

Examples

- Example 5–1, "Associative Array Indexed by String" on page 5-4
- Example 5–2, "Function Returns Associative Array Indexed by Integer" on page 5-5
- Example 5–3, "Varray (Variable-Size Array)" on page 5-8
- Example 5–4, "Nested Table of Local Type" on page 5-10
- Example 5–10, "Two-Dimensional Varray (Varray of Varrays)" on page 5-17
- Example 5–11, "Nested Tables of Nested Tables and Varrays of Integers" on page 5-17

Related Topics

In this chapter:

- "Collection Method Invocation" on page 13-31
- "FORALL Statement" on page 13-77
- "Record Variable Declaration" on page 13-111
- "%ROWTYPE Attribute" on page 13-121
- "%TYPE Attribute" on page 13-134

In other chapters:

- "Collection Topics" on page 5-1
- "BULK COLLECT Clause" on page 12-23
- "CREATE TYPE Statement" on page 14-68

Collection Method Invocation

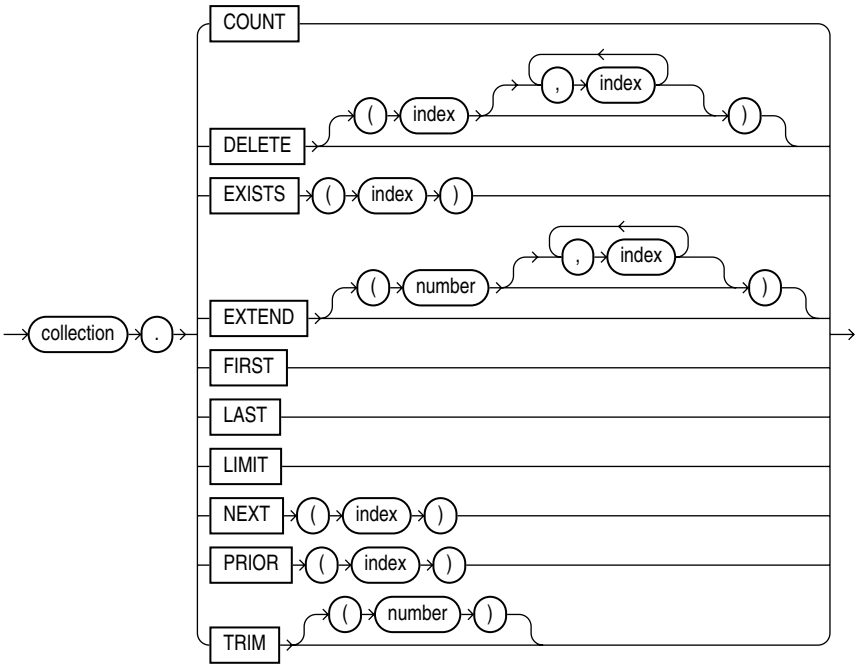
A collection method is a PL/SQL subprogram that either returns information about a collection or operates on a collection.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***collection_method_call* ::=**



Semantics

collection

Name of the collection whose method you are invoking.

COUNT

Function that returns the number of elements in the collection, explained in "COUNT Collection Method" on page 5-32.

DELETE

Procedure that deletes elements from the collection, explained in "DELETE Collection Method" on page 5-22.

Restriction on DELETE If *collection* is a varray, you cannot specify indexes with DELETE.

index

Numeric expression whose data type either is PLS_INTEGER or can be implicitly converted to PLS_INTEGER (for information about the latter, see "Predefined PLS_INTEGER Subtypes" on page 3-9).

EXISTS

Function that returns TRUE if the *index*th element of the collection exists and FALSE otherwise, explained in "EXISTS Collection Method" on page 5-28.

Restriction on EXISTS You cannot use EXISTS if *collection* is an associative array.

EXTEND

Procedure that adds elements to the end of the collection, explained in "EXTEND Collection Method" on page 5-26.

Restriction on EXTEND You cannot use EXTEND if *collection* is an associative array.

FIRST

Function that returns the first subscript in the collection, explained in "FIRST and LAST Collection Methods" on page 5-28.

LAST

Function that returns the last subscript in the collection, explained in "FIRST and LAST Collection Methods" on page 5-28.

LIMIT

Function that returns the maximum number of elements that the collection can have. If the collection has no maximum size, then LIMIT returns NULL. For an example, see "LIMIT Collection Method" on page 5-34.

NEXT

Function that returns the subscript of the succeeding existing element of the collection, if one exists. Otherwise, NEXT returns NULL. For more information, see "PRIOR and NEXT Collection Methods" on page 5-35.

PRIOR

Function that returns the subscript of the preceding existing element of the collection, if one exists. Otherwise, NEXT returns NULL. For more information, see "PRIOR and NEXT Collection Methods" on page 5-35.

TRIM

Procedure that deletes elements from the end of a collection, explained in "TRIM Collection Method" on page 5-25.

Restriction on TRIM You cannot use TRIM if *collection* is an associative array.

number

Number of elements to delete from the end of a collection. **Default:** one.

Examples

- Example 5–16, "DELETE Method with Nested Table" on page 5-23
- Example 5–17, "DELETE Method with Associative Array Indexed by String" on page 5-24
- Example 5–18, "TRIM Method with Nested Table" on page 5-26
- Example 5–19, "EXTEND Method with Nested Table" on page 5-27
- Example 5–20, "EXISTS Method with Nested Table" on page 5-28
- Example 5–21, "FIRST and LAST Values for Associative Array Indexed by Integer" on page 5-29
- Example 5–22, "FIRST and LAST Values for Associative Array Indexed by String" on page 5-29
- Example 5–23, "Printing Varray with FIRST and LAST in FOR LOOP" on page 5-30
- Example 5–24, "Printing Nested Table with FIRST and LAST in FOR LOOP" on page 5-31
- Example 5–25, "COUNT and LAST Values for Varray" on page 5-33
- Example 5–26, "COUNT and LAST Values for Nested Table" on page 5-33
- Example 5–27, "LIMIT and COUNT Values for Different Collection Types" on page 5-34
- Example 5–28, "PRIOR and NEXT Methods" on page 5-35
- Example 5–29, "Printing Elements of Sparse Nested Table" on page 5-36

Related Topics

In this chapter:

- "Collection Variable Declaration" on page 13-25

In other chapters:

- "Collection Methods" on page 5-21

Comment

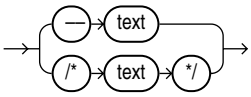
A comment is source program text that the PL/SQL compiler ignores. Its primary purpose is to document code, but you can also use it to disable obsolete or unfinished pieces of code (that is, you can turn the code into comments). PL/SQL has both single-line and multiline comments.

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 2–6, "Single-Line Comments" on page 2-10

- Example 2–7, "Multiline Comments" on page 2-10

Related Topics

- "Comments" on page 2-9

Constant Declaration

A constant holds a value that does not change.

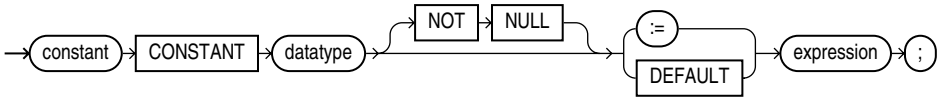
A constant declaration specifies the name, data type, and value of the constant and allocates storage for it. The declaration can also impose the `NOT NULL` constraint.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

constant_declaration ::=



See:

- "datatype ::=" on page 13-26
- "expression ::=" on page 13-61

Semantics

constant

Name of the constant that you are declaring.

datatype

Data type for which a variable can be declared with an initial value.

NOT NULL

Imposes the `NOT NULL` constraint on the constant. For information about this constraint, see "NOT NULL Constraint" on page 2-13.

expression

Initial value for the constant. *`expression`* must have a data type that is compatible with *`datatype`*. When *`constant_declaration`* is elaborated, the value of *`expression`* is assigned to *`constant`*.

Examples

- Example 2–10, "Constant Declarations" on page 2-12
- Example 2–11, "Variable and Constant Declarations with Initial Values" on page 2-13

Related Topics

In this chapter:

- "Collection Variable Declaration" on page 13-25
- "Record Variable Declaration" on page 13-111
- "%ROWTYPE Attribute" on page 13-121
- "Scalar Variable Declaration" on page 13-123
- "%TYPE Attribute" on page 13-134

In other chapters:

- "Comments" on page 2-9
- "Constant Declarations" on page 2-12

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.

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 Oracle Database 11*g* Release 1 (11.1), CONTINUE is a PL/SQL keyword. If your program invokes a subprogram named CONTINUE, you get a warning.

Restrictions on CONTINUE Statement

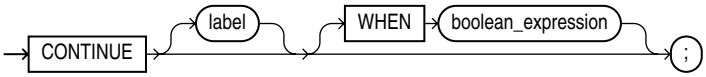
- A CONTINUE statement must be inside a LOOP statement.
- A CONTINUE statement cannot cross a subprogram or method boundary.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***continue_statement* ::=**



See "boolean_expression ::=" on page 13-62.

Semantics

label

Name that identifies either the current loop or an enclosing loop (see "Basic LOOP Statement" on page 13-7).

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.

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
- Example 4–26, "CONTINUE WHEN Statement in Inner FOR LOOP Statement" on page 4-19

Related Topics

In this chapter:

- "Basic LOOP Statement" on page 13-7
- "Cursor FOR LOOP Statement" on page 13-40
- "EXIT Statement" on page 13-55
- "Expression" on page 13-61
- "FOR LOOP Statement" on page 13-74
- "WHILE LOOP Statement" on page 13-138

In other chapters:

- "LOOP Statements" on page 4-9
- "CONTINUE Statement" on page 4-11
- "CONTINUE WHEN Statement" on page 4-12

Cursor FOR LOOP Statement

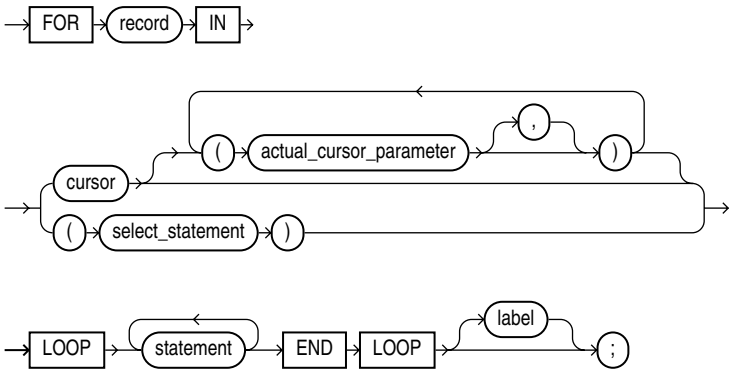
The cursor `FOR LOOP` statement implicitly declares its loop index as a record variable of the row type that a specified cursor returns, and then opens a cursor. With each iteration, the cursor `FOR LOOP` statement fetches a row from the result set into the record. When there are no more rows to fetch, the cursor `FOR LOOP` statement closes the cursor. The cursor also closes if a statement inside the loop transfers control outside the loop or raises an exception.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***cursor_for_loop_statement* ::=**



See "statement ::=" on page 13-13.

Semantics

record

Name for the loop index that the cursor `FOR LOOP` statement implicitly declares as a `%ROWTYPE` record variable of the type that *`cursor`* or *`select_statement`* returns.

`record` is local to the cursor `FOR LOOP` statement. Statements inside the loop can reference *`record`* and its fields. They can reference calculated columns only by aliases. Statements outside the loop cannot reference *`record`*. After the cursor `FOR LOOP` statement runs, *`record`* is undefined.

cursor

Name of an explicit cursor that is not open when the cursor `FOR LOOP` is entered.

actual_cursor_parameter

Actual parameter that corresponds to a formal parameter of *`cursor`*.

select_statement

SQL `SELECT` statement (not PL/SQL `SELECT INTO` statement). For *`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

label

Label that identifies *`cursor_for_loop_statement`* (see "statement ::=" on page 13-13 and "label" on page 13-16). `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 in the `END LOOP` statement matches a label at the beginning of the same `LOOP` statement (the compiler does not check).

Examples

- Example 6–18, "Implicit Cursor FOR LOOP Statement" on page 6-24
- Example 6–19, "Explicit Cursor FOR LOOP Statement" on page 6-25
- Example 6–20, "Passing Parameters to Explicit Cursor FOR LOOP Statement" on page 6-25
- Example 6–21, "Cursor FOR Loop References Calculated Columns" on page 6-26

Related Topics

In this chapter:

- "Basic LOOP Statement" on page 13-7
- "CONTINUE Statement" on page 13-38
- "EXIT Statement" on page 13-55
- "Explicit Cursor Declaration and Definition" on page 13-57
- "FETCH Statement" on page 13-71
- "FOR LOOP Statement" on page 13-74
- "FORALL Statement" on page 13-77
- "OPEN Statement" on page 13-101
- "WHILE LOOP Statement" on page 13-138

In other chapters:

- "Query Result Set Processing With Cursor FOR LOOP Statements" on page 6-24

Cursor Variable Declaration

A cursor variable is like an explicit cursor that is not limited to one query.

To create a cursor variable, either declare a variable of the predefined type `SYS_REFCURSOR` or define a `REF CURSOR` type and then declare a variable of that type.

Restrictions on Cursor Variables

- You cannot declare a cursor variable in a package specification.

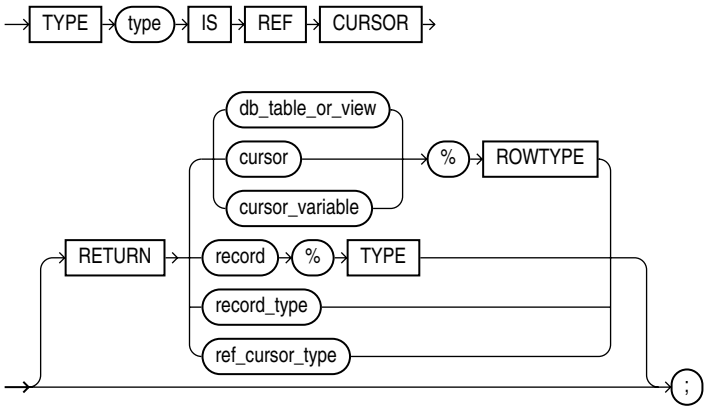
  That is, a package cannot have a public cursor variable (a cursor variable that can be referenced from outside the package).
- You cannot store the value of a cursor variable in a collection or database column.
- You cannot use comparison operators to test cursor variables for equality, inequality, or nullity.
- You can use a cursor variable in a server-to-server remote procedure call (RPC) only if the remote database is not an Oracle Database accessed through a Procedural Gateway.
- You cannot use LOB parameters in a server-to-server RPC.

Topics

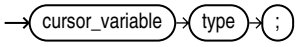
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***ref_cursor_type_definition* ::=**



***cursor_variable_declaration* ::=**



Semantics

ref_cursor_type_definition

type

Name of the `REF CURSOR` type that you are defining.

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 "Creating Cursor Variables" on page 6-28.

db_table_or_view

Name of a database table or view, which must be accessible when the declaration is elaborated.

cursor

Name of a previously declared explicit cursor.

cursor_variable

Name of a previously declared cursor variable.

record

Name of a user-defined record.

record_type

Name of a user-defined type that was defined with the data type specifier `RECORD`.

ref_cursor_type

Name of a user-defined type that was defined with the data type specifier `REF CURSOR`.

cursor_variable_declaration

cursor_variable

Name of the cursor variable that you are declaring.

type

Type of the cursor variable that you are declaring—either `SYS_REFCURSOR` or the name of the `REF CURSOR` type that you defined previously.

`SYS_REFCURSOR` is a weak type. For information about strong and weak `REF CURSOR` types, see "Creating Cursor Variables" on page 6-28.

Examples

- Example 6–24, "Cursor Variable Declarations" on page 6-29
- Example 6–25, "Cursor Variable with User-Defined Return Type" on page 6-29
- Example 6–28, "Variable in Cursor Variable Query—No Result Set Change" on page 6-33

- Example 6–29, "Variable in Cursor Variable Query—Result Set Change" on page 6-33
- Example 6–30, "Procedure to Open Cursor Variable for One Query" on page 6-35
- Example 6–31, "Opening Cursor Variable for Chosen Query (Same Return Type)" on page 6-36
- Example 6–32, "Opening Cursor Variable for Chosen Query (Different Return Types)" on page 6-36
- Example 6–33, "Cursor Variable as Host Variable in Pro*C Client Program" on page 6-37

Related Topics

In this chapter:

- "CLOSE Statement" on page 13-23
- "Named Cursor Attribute" on page 13-98
- "Explicit Cursor Declaration and Definition" on page 13-57
- "FETCH Statement" on page 13-71
- "OPEN FOR Statement" on page 13-103
- "%ROWTYPE Attribute" on page 13-121
- "%TYPE Attribute" on page 13-134

In other chapters:

- "Cursor Variables" on page 6-28
- "Passing CURSOR Expressions to Pipelined Table Functions" on page 12-44

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 13-136.

> **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 error code.

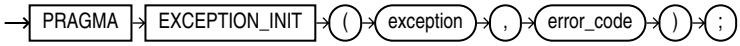
The `EXCEPTION_INIT` pragma can appear only in the same declarative part as its associated exception, anywhere after the exception declaration.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***exception_init_pragma* ::=**



Semantics

exception

Name of a previously declared user-defined exception.

error_code

Error code to be associated with *`exception`*. *`error_code`* can be either 100 (the numeric code for "no data found" that "SQLCODE Function" on page 13-131 returns) or any negative integer greater than -10000000 except -1403 (another numeric code for "no data found").

> **Note:** `NO_DATA_FOUND` is a predefined exception.

Examples

- Example 11–5, "Naming Internally Defined Exception" on page 11-10
- Example 11–12, "Raising User-Defined Exception with RAISE_APPLICATION_ERROR" on page 11-16
- Example 12–13, "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-20

Related Topics

In this chapter:

- "Exception Declaration" on page 13-48
- "Exception Handler" on page 13-50
- "SQLCODE Function" on page 13-131

- "SQLERRM Function" on page 13-132

In other chapters:

- "Internally Defined Exceptions" on page 11-9
- "RAISE_APPLICATION_ERROR Procedure" on page 11-16

Exception Declaration

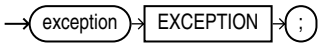
An exception declaration declares the name of a user-defined exception. You can use the `EXCEPTION_INIT` pragma to assign this name to an internally defined exception.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***exception_declaration* ::=**



Semantics

exception

Name of the exception that you are declaring.

Restriction on *exception*

You can use *`exception`* only in an `EXCEPTION_INIT` pragma, `RAISE` statement, `RAISE_APPLICATION_ERROR` invocation, or exception handler.

Caution: Oracle recommends against using a predefined exception name for *`exception`*. For details, see "Redeclared Predefined Exceptions" on page 11-12. For a list of predefined exception names, see Table 11–3.

Examples

- Example 11–5, "Naming Internally Defined Exception" on page 11-10
- Example 11–8, "Redeclared Predefined Identifier" on page 11-12
- Example 11–9, "Declaring, Raising, and Handling User-Defined Exception" on page 11-13

Related Topics

In this chapter:

- "EXCEPTION_INIT Pragma" on page 13-46
- "Exception Handler" on page 13-50
- "RAISE Statement" on page 13-110

In other chapters:

- "Internally Defined Exceptions" on page 11-9
- "User-Defined Exceptions" on page 11-11

Exception Handler

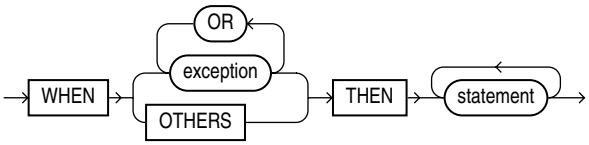
An exception handler processes a raised exception. Exception handlers appear in the exception-handling parts of anonymous blocks, subprograms, triggers, and packages.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

exception_handler ::=



See "statement ::=" on page 13-13.

Semantics

exception

Name of either a predefined exception (see Table 11–3) or a user-defined exception (see "Exception Declaration" on page 13-48).

If PL/SQL raises a specified exception, then the associated statements run.

OTHERS

Specifies all exceptions not explicitly specified in the exception-handling part of the block. If PL/SQL raises such an exception, then the associated statements run.

In the exception-handling part of a block, the `WHEN OTHERS` exception handler is optional. It can appear only once, as the last exception handler in the exception-handling part of the block.

Examples

- Example 11–3, "Single Exception Handler for Multiple Exceptions" on page 11-6
- Example 11–4, "Locator Variables for Statements that Share Exception Handler" on page 11-7
- Example 11–6, "Anonymous Block Handles ZERO_DIVIDE" on page 11-11
- Example 11–9, "Declaring, Raising, and Handling User-Defined Exception" on page 11-13
- Example 11–13, "Exception that Propagates Beyond Scope is Handled" on page 11-19
- Example 11–23, "Exception Handler Runs and Execution Ends" on page 11-25

- Example 11–24, "Exception Handler Runs and Execution Continues" on page 11-26
- Example 12–12, "Handling FORALL Exceptions Immediately" on page 12-18
- Example 12–13, "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-20

Related Topics

In this chapter:

- "Block" on page 13-9
- "EXCEPTION_INIT Pragma" on page 13-46
- "Exception Declaration" on page 13-48
- "Function Declaration and Definition" on page 13-83
- "Procedure Declaration and Definition" on page 13-108
- "RAISE Statement" on page 13-110
- "SQLCODE Function" on page 13-131
- "SQLERRM Function" on page 13-132

In other chapters:

- "Overview of Exception Handling" on page 11-4
- "Continuing Execution After Handling Exceptions" on page 11-25
- "Retrying Transactions After Handling Exceptions" on page 11-27
- "CREATE PACKAGE BODY Statement" on page 14-46
- "CREATE TRIGGER Statement" on page 14-54

EXECUTE IMMEDIATE Statement

The `EXECUTE IMMEDIATE` statement builds and runs a dynamic SQL statement in a single operation. Native dynamic SQL uses the `EXECUTE IMMEDIATE` statement to process most dynamic SQL statements.

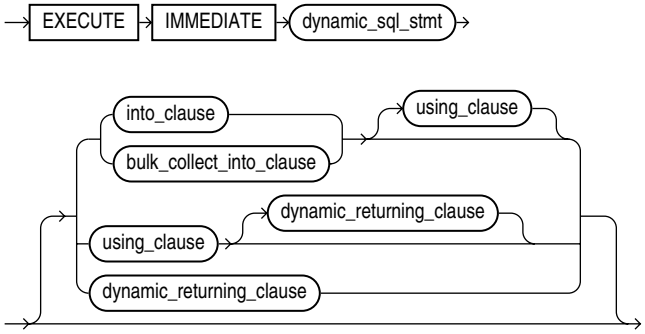
> **Caution:** When using dynamic SQL, beware of SQL injection, a security risk. For more information about SQL injection, see "SQL Injection" on page 7-9.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

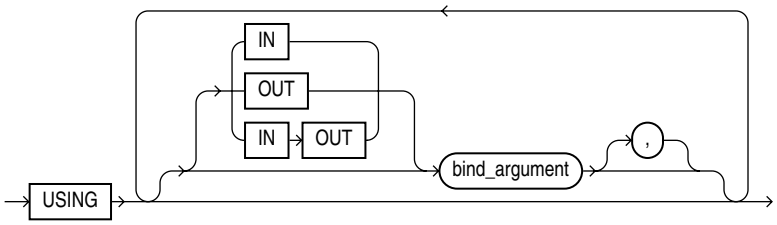
execute_immediate_statement ::=



See:

- "bulk_collect_into_clause ::=" on page 13-118
- "dynamic_returning_clause ::=" on page 13-118
- "into_clause ::=" on page 13-118

using_clause ::=



Semantics

execute_immediate_statement

dynamic_sql_stmt

String literal, string variable, or string expression that represents a SQL statement. Its type must be either `CHAR`, `VARCHAR2`, or `CLOB`.

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 13-118.

Restriction on *into_clause* Use if and only if *`dynamic_sql_stmt`* returns a single row.

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 13-118.

Restriction on *bulk_collect_into_clause* Use if and only if *`dynamic_sql_stmt`* can return multiple rows.

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.

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 13-118.

Restriction on *dynamic_returning_clause* Use if and only if *`dynamic_sql_stmt`* has a `RETURNING INTO` clause.

using_clause

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. **Default:** `IN`.

For DML a statement with a `RETURNING` clause, you can place `OUT` bind arguments in the `RETURNING INTO` clause without specifying the parameter mode, which is always `OUT`.

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 Represents NULL in USING Clause" on page 7-4.

Examples

- Example 7–1, "Invoking Subprogram from 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 Represents 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 13-118

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.

Restriction on EXIT Statement

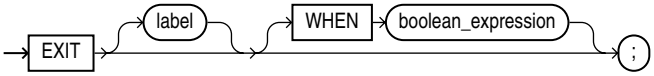
An `EXIT` statement must be inside a `LOOP` statement.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***exit_statement* ::=**



See "boolean_expression ::=" on page 13-62.

Semantics

label

Name that identifies either the current loop or an enclosing loop (see "Basic LOOP Statement" on page 13-7).

Without *`label`*, the `EXIT` statement transfers control to the next iteration of the current loop. With *`label`*, the `EXIT` statement transfers control to the next iteration of the loop that *`label`* identifies.

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 4–24, "EXIT WHEN Statement in FOR LOOP Statement" on page 4-18
- Example 4–25, "EXIT WHEN Statement in Inner FOR LOOP Statement" on page 4-19

Related Topics

In this chapter:

- "Basic LOOP Statement" on page 13-7
- "CONTINUE Statement" on page 13-38
- "EXIT Statement" on page 4-10
- "EXIT WHEN Statement" on page 4-10

Explicit Cursor Declaration and Definition

An **explicit cursor** is a named pointer to a private SQL area that stores information for processing a specific query or DML statement—typically, one that returns or affects multiple rows. You can use an explicit cursor to retrieve the rows of a result set one at a time.

Before using an explicit cursor, you must declare and define it. You can either declare it first (with ***cursor_declaration***) and then define it later in the same block, subprogram, or package (with ***cursor_definition***) or declare and define it at the same time (with ***cursor_definition***).

An explicit cursor declaration and definition are also called a **cursor specification** and **cursor body**, respectively.

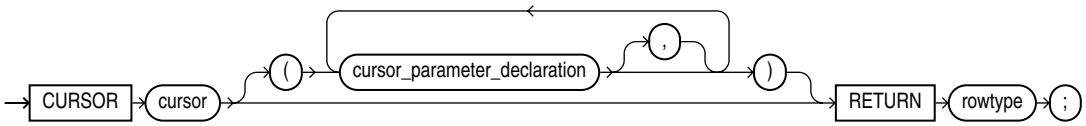
Note: An explicit cursor declared in a package specification is affected by the `AUTHID` clause of the package. For more information, see "CREATE PACKAGE Statement" on page 14-43.

Topics

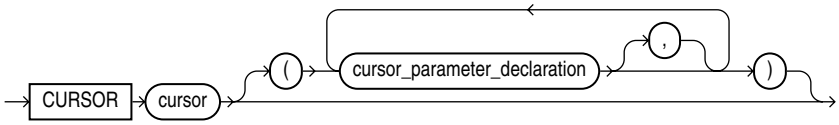
- Syntax
- Semantics
- Examples
- Related Topics

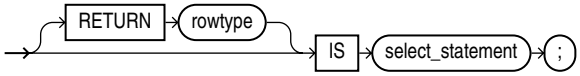
Syntax

***cursor_declaration* ::=**

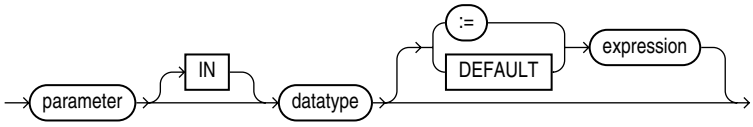


***cursor_definition* ::=**

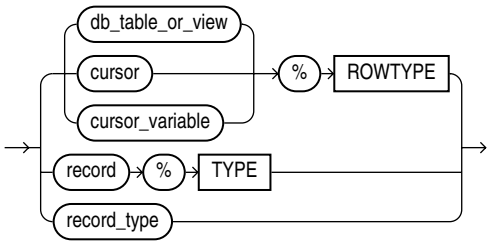




***cursor_parameter_declaration* ::=**



***rowtype* ::=**



Semantics

cursor_declaration

cursor

Name of the explicit cursor that you are declaring now and will define later in the same block, subprogram, or package. *cursor* can be any identifier except the reserved word SQL. Oracle recommends against giving a cursor the same name as a database table.

Explicit cursor names follow the same scoping rules as variables (see "Scope and Visibility of Identifiers" on page 2-16).

cursor_definition

Either defines an explicit cursor that was declared earlier or both declares and defines an explicit cursor.

cursor

Either the name of the explicit cursor that you previously declared and are now defining or the name of the explicit cursor that you are both declaring and defining. *cursor* can be any identifier except the reserved word SQL. Oracle recommends against giving a cursor the same name as a database table.

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

cursor_parameter_declaration

parameter

The name of the formal cursor parameter that you are declaring. This name can appear anywhere in *select_statement* that a constant can appear.

IN

Whether or not you specify IN, a formal cursor parameter has the characteristics of an IN subprogram parameter, which are summarized in Table 8–1. When the cursor opens, the value of the formal parameter is that of either its actual parameter or default value.

datatype

The data type of the parameter.

Restriction on *datatype* This *datatype* cannot have constraints (for example, NOT NULL, or precision and scale for a number, or length for a string).

expression

Specifies the default value for the formal cursor parameter. The data types of *expression* and the formal cursor parameter must be compatible.

If an OPEN statement does not specify an actual parameter for the formal cursor parameter, then the statement evaluates *expression* and assigns its value to the formal cursor parameter.

If an OPEN statement does specify an actual parameter for the formal cursor parameter, then the statement assigns the value of the actual parameter to the formal cursor parameter and does not evaluate *expression*.

rowtype

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.

db_table_or_view

Name of a database table or view, which must be accessible when the declaration is elaborated.

cursor

Name of a previously declared explicit cursor.

cursor_variable

Name of a previously declared cursor variable.

record

Name of a previously declared record variable.

record_type

Name of a user-defined type that was defined with the data type specifier RECORD.

Examples

- Example 6–5, "Explicit Cursor Declaration and Definition" on page 6-9
- Example 6–8, "Variable in Explicit Cursor Query—No Result Set Change" on page 6-12
- Example 6–9, "Variable in Explicit Cursor Query—Result Set Change" on page 6-13
- Example 6–10, "Explicit Cursor with Calculated Column that Needs Alias" on page 6-14
- Example 6–11, "Explicit Cursor that Accepts Parameters" on page 6-15
- Example 6–12, "Cursor Parameters with Default Values" on page 6-16
- Example 6–13, "Adding Formal Parameter to Existing Cursor" on page 6-18
- Example 6–22, "Subquery in FROM Clause of Parent Query" on page 6-27

- Example 6–23, "Correlated Subquery" on page 6-27
- Example 6–34, "CURSOR Expression" on page 6-39
- Example 6–40, "FOR UPDATE Cursor in CURRENT OF Clause of UPDATE Statement" on page 6-47
- Example 6–41, "SELECT FOR UPDATE Statement for Multiple Tables" on page 6-47
- Example 6–42, "FETCH with FOR UPDATE Cursor After COMMIT Statement" on page 6-48

Related Topics

In this chapter:

- "CLOSE Statement" on page 13-23
- "Cursor FOR LOOP Statement" on page 13-40
- "Cursor Variable Declaration" on page 13-42
- "FETCH Statement" on page 13-71
- "Named Cursor Attribute" on page 13-98
- "OPEN Statement" on page 13-101
- "%ROWTYPE Attribute" on page 13-121
- "%TYPE Attribute" on page 13-134

In other chapters:

- "Explicit Cursors" on page 6-8
- "Query Result Set Processing" on page 6-22
- "SELECT FOR UPDATE and FOR UPDATE Cursors" on page 6-46

Expression

An expression is an arbitrarily complex combination of operands (variables, constants, literals, operators, function invocations, and placeholders) and operators. The simplest expression is a single variable.

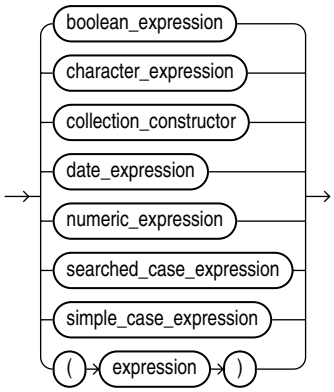
The PL/SQL compiler determines the data type of an expression from the types of the operands and operators that comprise the expression. Every time the expression is evaluated, a single value of that type results.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

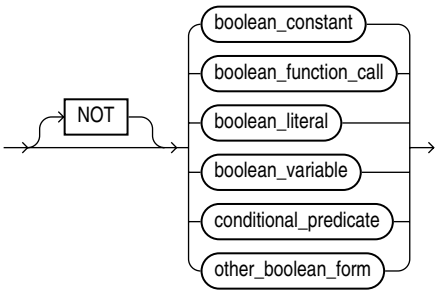
expression ::=

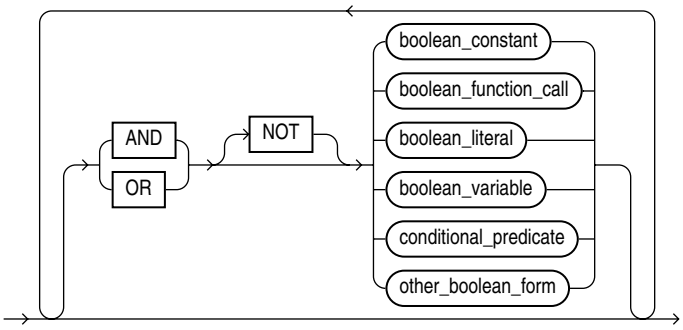


See:

- "boolean_expression ::=" on page 13-62
- "character_expression ::=" on page 13-63
- "collection_constructor ::=" on page 13-63
- "date_expression ::=" on page 13-64
- "numeric_expression ::=" on page 13-64
- "searched_case_expression ::=" on page 13-65
- "simple_case_expression ::=" on page 13-65

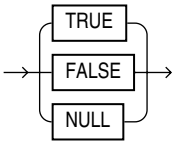
boolean_expression ::=



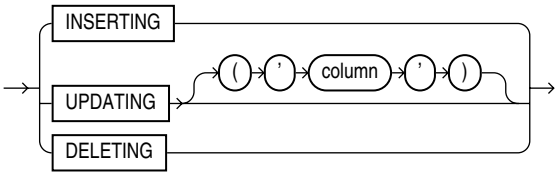


See "function_call ::=" on page 13-65.

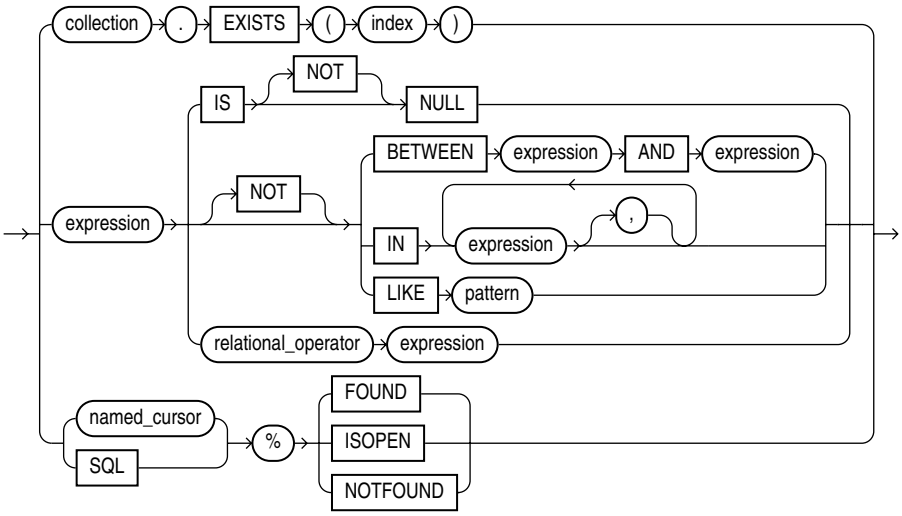
boolean_literal ::=



conditional_predicate ::=



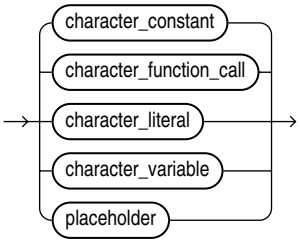
other_boolean_form ::=

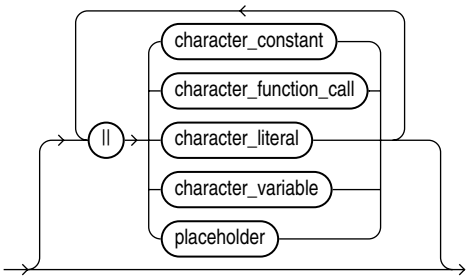


See:

- "expression ::=" on page 13-61
- "named_cursor ::=" on page 13-98

character_expression ::=

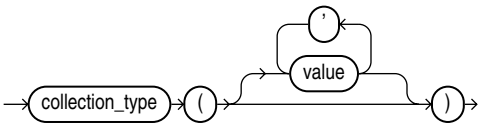




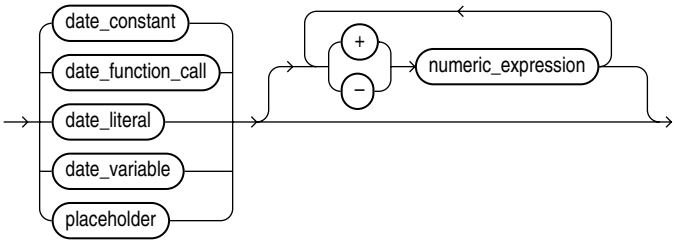
See:

- "function_call ::=" on page 13-65
- "placeholder ::=" on page 13-3

collection_constructor ::=



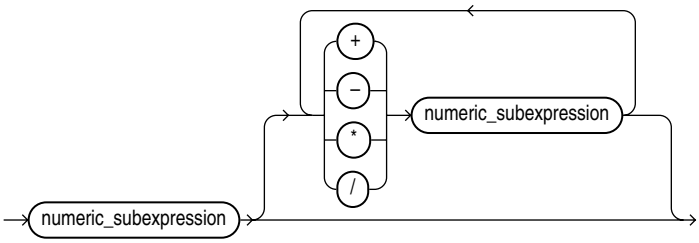
date_expression ::=



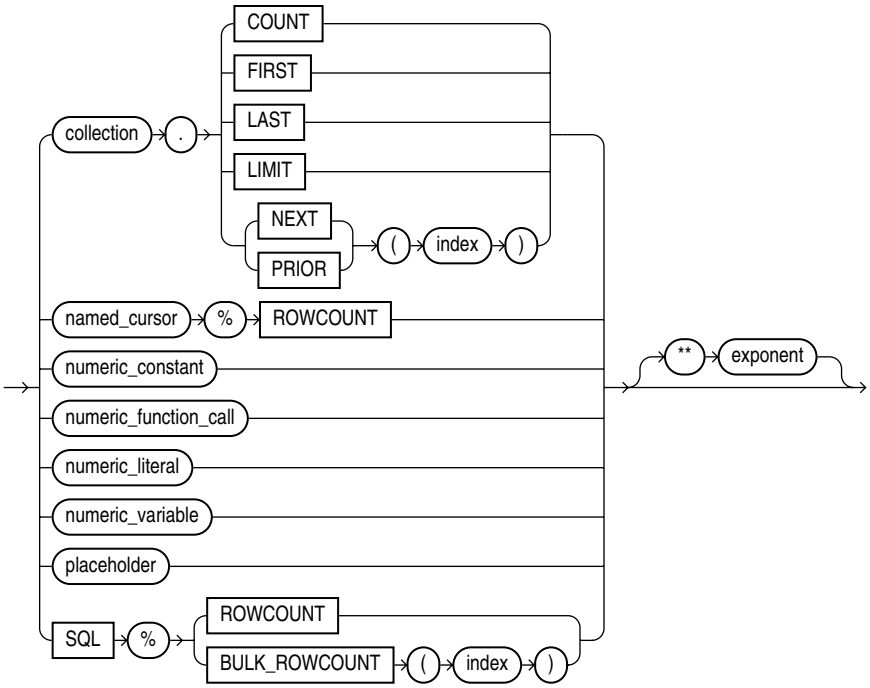
See:

- "function_call ::=" on page 13-65
- "placeholder ::=" on page 13-3

numeric_expression ::=



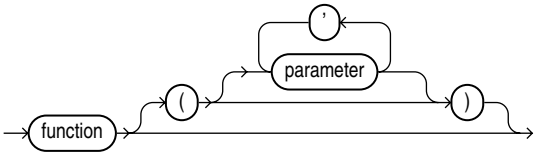
numeric_subexpression ::=



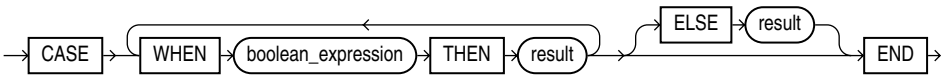
See:

- "function_call ::=" on page 13-65
- "named_cursor ::=" on page 13-98
- "placeholder ::=" on page 13-3

***function_call* ::=**

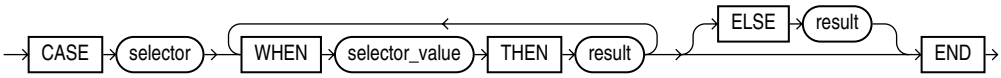


***searched_case_expression* ::=**



See "boolean_expression ::=" on page 13-62.

***simple_case_expression* ::=**



Semantics

boolean_expression

Expression whose value is `TRUE`, `FALSE`, or `NULL`. For more information, see "BOOLEAN Expressions" on page 2-37.

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, SQL function, or PL/SQL function invoked from a SQL statement

NOT, AND, OR

See "Logical Operators" on page 2-27.

boolean_constant

Name of a constant of type `BOOLEAN`.

boolean_function_call

Invocation of a previously defined function that returns a `BOOLEAN` value. For more semantic information, see "function_call" on page 13-68.

boolean_variable

Name of a variable of type `BOOLEAN`.

conditional_predicate

See "Conditional Predicates for Detecting Triggering DML Statement" on page 9-4.

other_boolean_form

collection

Name of a collection variable.

EXISTS

Collection method (function) that returns `TRUE` if the *`index`*th element of *`collection`* exists and `FALSE` otherwise. For more information, see "EXISTS Collection Method" on page 5-28.

Restriction on EXISTS You cannot use `EXISTS` if *`collection`* is an associative array.

index

Numeric expression whose data type either is `PLS_INTEGER` or can be implicitly converted to `PLS_INTEGER` (for information about the latter, see "Predefined PLS_INTEGER Subtypes" on page 3-9).

IS [NOT] NULL

See "IS [NOT] NULL Operator" on page 2-34.

BETWEEN *expression* AND *expression*

See "BETWEEN Operator" on page 2-35.

IN *expression* [, *expression*]...

See "IN Operator" on page 2-36.

LIKE *pattern*

See "LIKE Operator" on page 2-34.

relational_operator

See "Relational Operators" on page 2-32.

SQL

Implicit cursor associated with the most recently run `SELECT` or DML statement. For more information, see "Implicit Cursors" on page 6-5.

%FOUND, %ISOPEN, %NOTFOUND

Cursor attributes explained in "Implicit Cursor Attribute" on page 13-91 and "Named Cursor Attribute" on page 13-98.

character_expression

Expression whose value has a character data type (that is, a data type in the `CHAR` family, described in "CHAR Data Type Family" on page E-1).

character_constant

Name of a constant that has a character data type.

character_function_call

Invocation of a previously defined function that returns a value that either has a character data type or can be implicitly converted to a character data type. For more semantic information, see "function_call" on page 13-68.

character_literal

Literal of a character data type.

character_variable

Name of a variable that has a character data type.

||

Concatenation operator, which appends one string operand to another. For more information, see "Concatenation Operator" on page 2-24.

collection_constructor

Constructs a collection of the specified type with elements that have the specified values. For more information, see "Collection Constructors" on page 5-13.

collection_type

Name of a previously declared nested table type or `VARRAY` type (not an associative array type).

value

Valid value for an element of a collection of *`collection_type`*.

If *`collection_type`* is a varray type, then it has a maximum size, which the number of values cannot exceed. If *`collection_type`* is a nested table type, then it has no maximum size.

If you specify no values, then the constructed collection is empty but not null (for the difference between *empty* and *null*, see "Collection Types" on page 5-2).

date_expression

Expression whose value has a date data type (that is, a data type in the `DATE` family, described in "DATE Data Type Family" on page E-2).

date_constant

Name of a constant that has a date data type.

date_function_call

Invocation of a previously defined function that returns a value that either has a date data type or can be implicitly converted to a date data type. For more semantic information, see "function_call" on page 13-68.

date_literal

Literal whose value either has a date data type or can be implicitly converted to a date data type.

date_variable

Name of a variable that has a date data type.

+, -
Addition and subtraction operators.

numeric_expression

Expression whose value has a date numeric type (that is, a data type in the `DATE` family, described in "NUMBER Data Type Family" on page E-2).

+, -, /, *, **
Addition, subtraction, division, multiplication, and exponentiation operators.

numeric_subexpression

collection
Name of a collection variable.

COUNT, FIRST, LAST, LIMIT, NEXT, PRIOR
Collection methods explained in "Collection Method Invocation" on page 13-31.

***named_cursor*%ROWCOUNT**
See "Named Cursor Attribute" on page 13-98.

numeric_constant
Name of a constant that has a numeric data type.

numeric_function_call
Invocation of a previously defined function that returns a value that either has a numeric data type or can be implicitly converted to a numeric data type. For more semantic information, see "function_call" on page 13-68.

numeric_literal
Literal of a numeric data type.

numeric_variable
Name of variable that has a numeric data type.

SQL%ROWCOUNT
Cursor attribute explained in "Implicit Cursor Attribute" on page 13-91.

SQL%BULK_ROWCOUNT]
Cursor attribute explained in "SQL%BULK_ROWCOUNT" on page 13-92.

exponent
Numeric expression.

function_call

function
Name of a previously defined function.

***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. The mode of the formal parameter determines what the actual parameter can be:

Formal Parameter Mode	Actual Parameter
`IN`	Constant, initialized variable, literal, or expression
`OUT`	Variable whose data type is not defined as `NOT NULL`
`IN OUT`	Variable (typically, it is a string buffer or numeric accumulator)

If the function specifies a default value for a parameter, you can omit that parameter from the parameter list. If the function has no parameters, or specifies a default 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 8-21

searched_case_expression

WHEN *boolean_expression* THEN *result*

The *`boolean_expression`*s are evaluated sequentially. If a *`boolean_expression`* has the value `TRUE`, then the *`result`* associated with that *`boolean_expression`* is returned. Subsequent *`boolean_expression`*s are not evaluated.

ELSE *result*

The *`result`* is returned if and only if no *`boolean_expression`* has the value `TRUE`.

If you omit the `ELSE` clause, the searched case expression returns `NULL`.

> **See Also:** "Searched CASE Statement" on page 4-7

simple_case_expression

selector

An expression of any PL/SQL type except `BLOB`, `BFILE`, or a user-defined type. The *`selector`* is evaluated once.

WHEN *selector_value* THEN *result*

The *`selector_value`*s are evaluated sequentially. If a *`selector_value`* is the value of *`selector`*, then the *`result`* associated with that *`selector_value`* is returned. Subsequent *`selector_value`*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.

ELSE *result*

The *`result`* is returned if and only if no *`selector_value`* has the same value as *`selector`*.

If you omit the `ELSE` clause, the simple case expression returns `NULL`.

See Also: "Simple CASE Statement" on page 4-6

Examples

- Example 2–28, "Concatenation Operator" on page 2-24
- Example 2–29, "Concatenation Operator with NULL Operands" on page 2-24
- Example 2–30, "Controlling Evaluation Order with Parentheses" on page 2-25
- Example 2–31, "Expression with Nested Parentheses" on page 2-25
- Example 2–32, "Improving Readability with Parentheses" on page 2-26
- Example 2–33, "Operator Precedence" on page 2-26
- Example 2–43, "Relational Operators in Expressions" on page 2-33
- Example 2–44, "LIKE Operator in Expression" on page 2-34
- Example 2–46, "BETWEEN Operator in Expressions" on page 2-35
- Example 2–47, "IN Operator in Expressions" on page 2-36
- Example 2–50, "Simple CASE Expression" on page 2-38
- Example 2–52, "Searched CASE Expression" on page 2-39
- Example 9–1, "Trigger Uses Conditional Predicates to Detect Triggering Statement" on page 9-5

Related Topics

In this chapter:

- "Collection Method Invocation" on page 13-31
- "Constant Declaration" on page 13-36
- "Scalar Variable Declaration" on page 13-123

In other chapters:

- "Literals" on page 2-8
- "Expressions" on page 2-23
- "Operator Precedence" on page 2-24
- Chapter 3, "PL/SQL Data Types"
- "Subprogram Parameters" on page 8-9

FETCH Statement

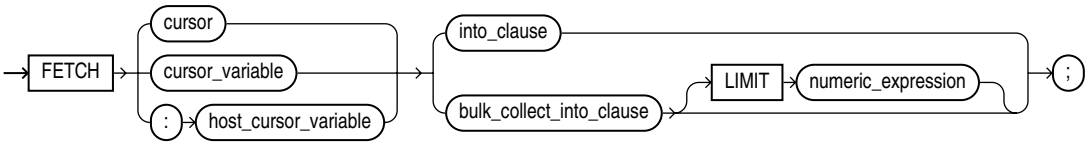
The `FETCH` statement retrieves rows of data from the result set of a multiple-row query—one row at a time, several rows at a time, or all rows at once—and stores the data in variables, records, or collections.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

fetch_statement ::=



See:

- "bulk_collect_into_clause ::=" on page 13-118
- "into_clause ::=" on page 13-118
- "numeric_expression ::=" on page 13-64

Semantics

cursor

Name of an open explicit cursor. To open an explicit cursor, use the "OPEN Statement" on page 13-101.

If you try to fetch from an explicit cursor before opening it or after closing it, PL/SQL raises the predefined exception `INVALID_CURSOR`.

cursor_variable

Name of an open cursor variable. To open a cursor variable, use the "OPEN FOR Statement" on page 13-103. The cursor variable can be a formal subprogram parameter (see "Cursor Variables as Subprogram Parameters" on page 6-35).

If you try to fetch from a cursor variable before opening it or after closing it, PL/SQL raises the predefined exception `INVALID_CURSOR`.

:*host_cursor_variable*

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 host cursor variable, use the "OPEN FOR Statement" on page 13-103. Do not put space between the colon (:) and *`host_cursor_variable`*.

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 *`into_clause`*, see "into_clause" on page 13-119.

bulk_collect_into_clause [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 "bulk_collect_into_clause" on page 13-119.

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`*.

Restrictions on *bulk_collect_into_clause*

- You cannot use *`bulk_collect_into_clause`* in client programs.
- When the `FETCH` statement requires implicit data type conversions, *`bulk_collect_into_clause`* can have only one *`collection`* or *`host_array`*.

Examples

- Example 5–44, "FETCH Assigns Values to Record that Function Returns" on page 5-48
- Example 6–6, "FETCH Statements Inside LOOP Statements" on page 6-10
- Example 6–7, "Fetching Same Explicit Cursor into Different Variables" on page 6-11
- Example 6–26, "Fetching Data with Cursor Variables" on page 6-31
- Example 6–27, "Fetching from Cursor Variable into Collections" on page 6-32
- Example 6–42, "FETCH with FOR UPDATE Cursor After COMMIT Statement" on page 6-48
- Example 7–4, "Native Dynamic SQL with OPEN FOR, FETCH, and CLOSE Statements" on page 7-5
- Example 12–22, "Bulk-Fetching into Two Nested Tables" on page 12-31
- Example 12–23, "Bulk-Fetching into Nested Table of Records" on page 12-33
- Example 12–24, "Limiting Bulk FETCH with LIMIT" on page 12-34

Related Topics

In this chapter:

- "Assignment Statement" on page 13-3
- "CLOSE Statement" on page 13-23
- "Cursor Variable Declaration" on page 13-42
- "Explicit Cursor Declaration and Definition" on page 13-57
- "OPEN Statement" on page 13-101

- "OPEN FOR Statement" on page 13-103
- "RETURNING INTO Clause" on page 13-118
- "%ROWTYPE Attribute" on page 13-121
- "SELECT INTO Statement" on page 13-125
- "%TYPE Attribute" on page 13-134

In other chapters:

- "FETCH Statement for Assigning Row to Record Variable" on page 5-48
- "Fetching Data with Explicit Cursors" on page 6-10
- "Query Result Set Processing With Cursor FOR LOOP Statements" on page 6-24
- "Fetching Data with Cursor Variables" on page 6-30
- "OPEN FOR, FETCH, and CLOSE Statements" on page 7-4
- "FETCH Statement with BULK COLLECT Clause" on page 12-31
- "Fetching from Results of Pipelined Table Functions" on page 12-43

FOR LOOP Statement

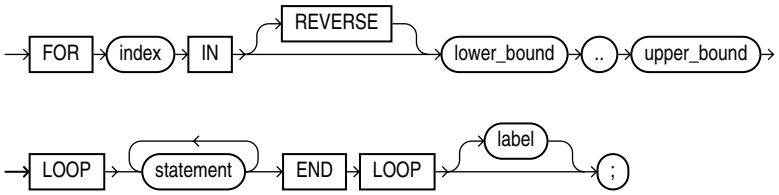
With each iteration of the `FOR LOOP` statement, its statements run, its index is either incremented or decremented, and control returns to the top of the loop. The `FOR LOOP` statement ends when its index reaches a specified value, or when a statement inside the loop transfers control outside the loop or raises an exception.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

for_loop_statement ::=



See "statement ::=" on page 13-13.

Semantics

index

Name for the implicitly declared integer variable that is local to the `FOR LOOP` statement. Statements outside the loop cannot reference *`index`*. Statements inside the loop can reference *`index`*, but cannot change its value. After the `FOR LOOP` statement runs, *`index`* is undefined.

See Also: "FOR LOOP Index" on page 4-15

[REVERSE] *lower_bound* .. *upper_bound*

`lower_bound` and *`upper_bound`* must evaluate to numbers (see "Lower Bound and Upper Bound" on page 4-17). 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.

label

A label that identifies *for_loop_statement* (see "statement ::=" on page 13-13 and "label" on page 13-16). 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 in the END LOOP statement matches a label at the beginning of the same LOOP statement (the compiler does not check).

Examples

- Example 4–14, "FOR LOOP Statements" on page 4-13
- Example 4–15, "Reverse FOR LOOP Statements" on page 4-14
- Example 4–16, "Simulating STEP Clause in FOR LOOP Statement" on page 4-15
- Example 4–18, "Outside Statement References FOR LOOP Statement Index" on page 4-16
- Example 4–19, "FOR LOOP Statement Index with Same Name as Variable" on page 4-16
- Example 4–20, "FOR LOOP Statement References Variable with Same Name as Index" on page 4-16
- Example 4–21, "Nested FOR LOOP Statements with Same Index Name" on page 4-17
- Example 4–22, "FOR LOOP Statement Bounds" on page 4-17
- Example 4–23, "Specifying FOR LOOP Statement Bounds at Run Time" on page 4-18

Related Topics

In this chapter:

- "Basic LOOP Statement" on page 13-7
- "CONTINUE Statement" on page 13-38
- "Cursor FOR LOOP Statement" on page 13-40
- "EXIT Statement" on page 13-55

- "FETCH Statement" on page 13-71
- "FORALL Statement" on page 13-77
- "OPEN Statement" on page 13-101
- "WHILE LOOP Statement" on page 13-138

In other chapters:

- "FOR LOOP Statement" on page 4-13

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 or host arrays. The FORALL statement is usually much faster than an equivalent FOR LOOP statement.

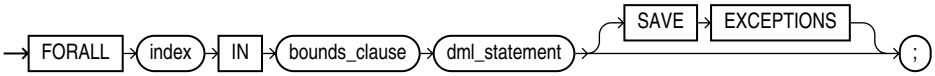
> **Note:** You can use the FORALL statement only in server programs, not in client programs.

Topics

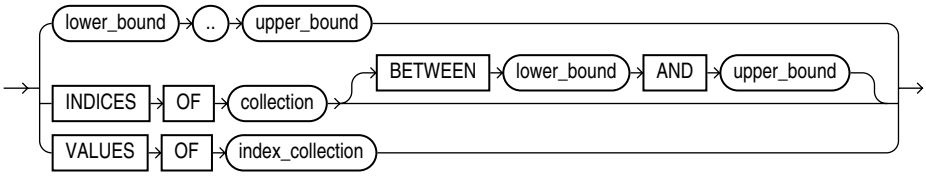
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***forall_statement* ::=**



***bounds_clause* ::=**



Semantics

forall_statement

index

Name for the implicitly declared integer variable that is local to the FORALL statement. Statements outside the FORALL statement cannot reference *index*. Statements inside the FORALL statement can reference *index* as an index variable, but cannot use it in expressions or change its value. After the FORALL statement runs, *index* is undefined.

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* as a subscript.

Every collection that *dml_statement* references must have subscripts that match the values of *index*. 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, PL/SQL raises an exception.

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.

SAVE EXCEPTIONS

Enables the FORALL statement to continue even if some of its DML statements fail. For more information, see "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-19.

bounds_clause

Specifies the collection element subscripts that provide values for the variable *index*. For each value, the SQL engine runs *dml_statement* once.

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, PL/SQL raises an exception.

INDICES OF *collection* [BETWEEN *lower_bound* AND *upper_bound*]

Specifies that the values of *index* correspond to the subscripts of the elements of *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.

VALUES OF *index_collection*

Specifies that the values of *index* 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, PL/SQL raises an exception and the FORALL statement does not run.

Examples

- Example 12–8, "DELETE Statement in FORALL Statement" on page 12-12
- Example 12–9, "Time Difference for INSERT Statement in FOR LOOP and FORALL Statements" on page 12-12

- Example 12–10, "FORALL Statement for Subset of Collection" on page 12-13
- Example 12–11, "FORALL Statements for Sparse Collection and Its Subsets" on page 12-14
- Example 12–12, "Handling FORALL Exceptions Immediately" on page 12-18
- Example 12–13, "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-20
- Example 12–26, "DELETE with RETURN BULK COLLECT INTO in FORALL Statement" on page 12-36
- Example 12–28, "Anonymous Block Bulk-Binds Input Host Array" on page 12-38

Related Topics

In this chapter:

- "FOR LOOP Statement" on page 13-74
- "Implicit Cursor Attribute" on page 13-91

In other chapters:

- "FORALL Statement" on page 12-11
- "BULK COLLECT Clause" on page 12-23
- "Using FORALL Statement and BULK COLLECT Clause Together" on page 12-36

Formal Parameter Declaration

A formal parameter declaration can appear in the following:

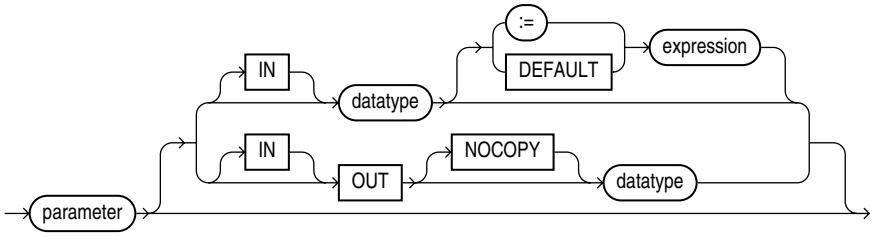
- "Function Declaration and Definition" on page 13-83
- "Procedure Declaration and Definition" on page 13-108
- "CREATE FUNCTION Statement" on page 14-32
- "CREATE PROCEDURE Statement" on page 14-50

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

parameter_declaration ::=



See:

- "datatype ::=" on page 13-26
- "expression ::=" on page 13-61

Semantics

parameter

Name of the formal parameter that you are declaring, which you can reference in the executable part of the subprogram.

IN, OUT, IN OUT

Mode that determines the behavior of the parameter, explained in "Subprogram Parameter Modes" on page 8-12. **Default:** `IN`.

Note: Avoid using `OUT` and `IN OUT` for function parameters. 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

Requests that the compiler pass the corresponding actual parameter by reference instead of value (for the difference, see "Subprogram Parameter Passing Methods" on page 8-11). Each time the subprogram is invoked, the optimizer decides, silently, whether to obey or disregard `NOCOPY`.

> **Caution:** `NOCOPY` increases the likelihood of aliasing. For details, see "Subprogram Parameter Aliasing with Parameters Passed by Reference" on page 8-16.

The compiler always ignores `NOCOPY` in these cases:

- The actual parameter must be implicitly converted to the data type of the formal parameter.
- The actual parameter is the element of an associative array.
- The actual parameter is a scalar variable with the `NOT NULL` constraint.
- The actual parameter is a scalar numeric variable with a range, size, scale, or precision constraint.
- The actual and formal parameters are records, one or both was declared with `%ROWTYPE` or `%TYPE`, and constraints on corresponding fields differ.
- The actual and formal parameters are records, the actual parameter was declared (implicitly) as the index of a cursor `FOR LOOP` statement, and constraints on corresponding fields differ.
- The subprogram is invoked through a database link or as an external subprogram.

datatype

Data type of the formal parameter that you are declaring. If *`datatype`* is a constrained subtype, the corresponding actual parameter inherits the `NOT NULL` constraint of the subtype (if it has one), but not the size (see Example 8–10).

> **Caution:** The data type `REF CURSOR` increases the likelihood of subprogram parameter aliasing, which can have unintended results. For more information, see "Subprogram Parameter Aliasing with Cursor Variable Parameters" on page 8-18.

expression

Default value of the formal parameter that you are declaring. The data type of *`expression`* must be compatible with *`datatype`*.

If a subprogram invocation does not specify an actual parameter for the formal parameter, then that invocation evaluates *`expression`* and assigns its value to the formal parameter.

If a subprogram invocation does specify an actual parameter for the formal parameter, then that invocation assigns the value of the actual parameter to the formal parameter and does not evaluate *`expression`*.

Examples

- Example 2–26, "Assigning Value to Variable as IN OUT Subprogram Parameter" on page 2-22

- Example 8–9, "Formal Parameters and Actual Parameters" on page 8-9
- Example 8–12, "Parameter Values Before, During, and After Procedure Invocation" on page 8-13
- Example 8–13, "OUT and IN OUT Parameter Values After Unhandled Exception" on page 8-15
- Example 8–17, "Procedure with Default Parameter Values" on page 8-19
- Example 8–18, "Function Provides Default Parameter Value" on page 8-19
- Example 8–19, "Adding Subprogram Parameter Without Changing Existing Invocations" on page 8-20

Related Topics

In this chapter:

- "Function Declaration and Definition" on page 13-83
- "Procedure Declaration and Definition" on page 13-108

In other chapters:

- "Subprogram Parameters" on page 8-9
- "Tune Subprogram Invocations" on page 12-7
- "CREATE FUNCTION Statement" on page 14-32
- "CREATE PROCEDURE Statement" on page 14-50

Function Declaration and Definition

A **function** is a subprogram that returns a value. The data type of the value is the data type of the function. A function invocation (or call) is an expression, whose data type is that of the function.

Before invoking a function, you must declare and define it. You can either declare it first (with ***function_declaration***) and then define it later in the same block, subprogram, or package (with ***function_definition***) or declare and define it at the same time (with ***function_definition***).

A function declaration is also called a **function specification** or **function spec**.

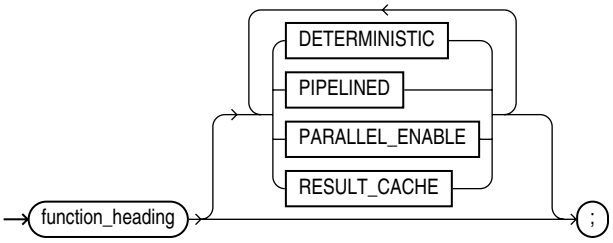
> **Note:** This topic applies to nested functions. For information about standalone stored functions, see "CREATE FUNCTION Statement" on page 14-32. For information about package functions, see "CREATE PACKAGE Statement" on page 14-43.

Topics

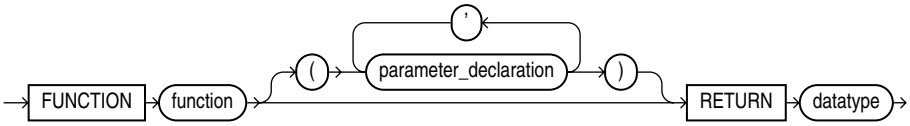
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***function_declaration* ::=**



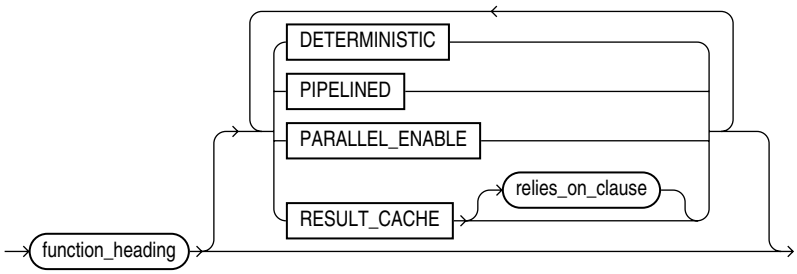
***function_heading* ::=**

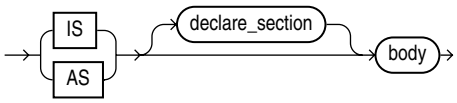


See:

- "datatype ::=" on page 13-26
- "parameter_declaration ::=" on page 13-80

***function_definition* ::=**

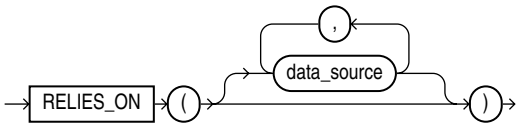




See:

- "body ::=" on page 13-12
- "call_spec ::=" on page 14-34
- "declare_section ::=" on page 13-9

***relies_on_clause* ::=**



Semantics

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.

DETERMINISTIC

Tells the optimizer that the function returns the same value whenever it is invoked with the same parameter values (if this is not true, then specifying `DETERMINISTIC` causes unpredictable results). If the function was invoked previously with the same parameter values, the optimizer can use the previous result instead of invoking the function again.

Do not specify `DETERMINISTIC` for a function whose result depends on the state of session variables or schema objects, because results might vary across invocations. Instead, consider making the function result-cached (see "Making Result-Cached Functions Handle Session-Specific Settings" on page 8-38 and "Making Result-Cached Functions Handle Session-Specific Application Contexts" on page 8-39).

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 8-31
- `CREATE INDEX` statement in *Oracle Database SQL Language Reference*

PIPELINED

Use only with a table function, to specify that it is pipelined. A pipelined table function returns a row to its invoker immediately after processing that row and continues to process rows. To return a row (but not control) to the invoker, the function uses the "PIPE ROW Statement" on page 13-106.

See Also:

- "Overview of Table Functions" on page 12-39
- "Creating Pipelined Table Functions" on page 12-40

Note: You cannot run a pipelined table function 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.

PARALLEL_ENABLE

Enables the function for parallel execution, making it safe for use in slave sessions of parallel DML evaluations.

RESULT_CACHE

Caches the results of the function. For more information, see "PL/SQL Function Result Cache" on page 8-31.

function_heading

function

Name of the function that you are declaring or defining.

RETURN *datatype*

Specifies the data type of the value that the function returns, which can be any PL/SQL data type (see Chapter 3, "PL/SQL Data Types").

Restriction on *datatype* You cannot constrain this data type (with `NOT NULL`, for example).

function_definition

Either defines a function that was declared earlier or both declares and defines a function.

declare_section

Declares items that are local to the function, can be referenced in *body*, and cease to exist when the function completes execution.

body

Required executable part and optional exception-handling part of the function. In the executable part, at least one execution path must lead to a `RETURN` statement; otherwise, a run-time error occurs.

***call_spec*, EXTERNAL**

See "call_spec" on page 14-38 and "EXTERNAL" on page 14-38.

Restriction on *call_spec*, EXTERNAL These clauses can appear only in a package specification or package body.

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.

Examples

- Example 8–2, "Declaring, Defining, and Invoking a Simple PL/SQL Function" on page 8-5

Related Topics

In this chapter:

- "Formal Parameter Declaration" on page 13-80
- "PIPE ROW Statement" on page 13-106
- "Procedure Declaration and Definition" on page 13-108

In other chapters:

- Chapter 8, "PL/SQL Subprograms"
- "Creating Pipelined Table Functions" on page 12-40

GOTO Statement

The `GOTO` statement transfers control to a labeled block or statement.

If a `GOTO` statement exits a cursor `FOR LOOP` statement prematurely, the cursor closes.

Restrictions on GOTO Statement

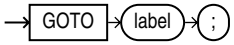
- A `GOTO` statement cannot transfer control into an `IF` statement, `CASE` statement, `LOOP` statement, or sub-block.
- A `GOTO` statement cannot transfer control from one `IF` statement clause to another, or from one `CASE` statement `WHEN` clause to another.
- A `GOTO` statement cannot transfer control out of a subprogram.
- A `GOTO` statement cannot transfer control into an exception handler.
- A `GOTO` statement cannot transfer control from an exception handler back into the current block (but it can transfer control from an exception handler into an enclosing block).

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***goto_statement* ::=**



Semantics

label

Identifies either a block or a statement (see "plsql_block ::=" on page 13-9, "statement ::=" on page 13-13, and "label" on page 13-16).

If *`label`* is not in the current block, then the `GOTO` statement transfers control to the first enclosing block in which *`label`* appears.

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 4-23

Related Topics

In this chapter:

- "Block" on page 13-9

In other chapters:

- "GOTO Statement" on page 4-21

IF Statement

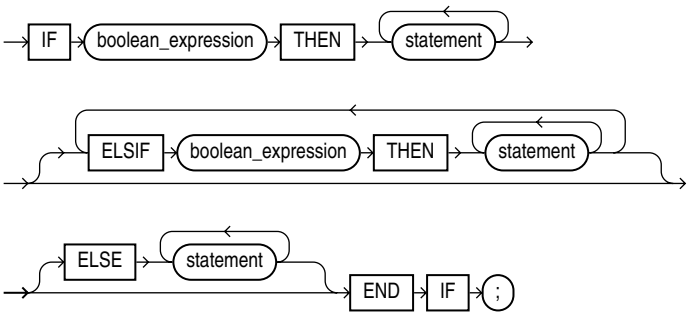
The `IF` statement either runs or skips a sequence of one or more statements, depending on the value of a `BOOLEAN` expression.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***if_statement* ::=**



See:

- "boolean_expression ::=" on page 13-62
- "statement ::=" on page 13-13

Semantics

boolean_expression

Expression whose value is `TRUE`, `FALSE`, or `NULL`.

The first *`boolean_expression`* is always evaluated. Each other *`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 13-20
- "Expression" on page 13-61

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.

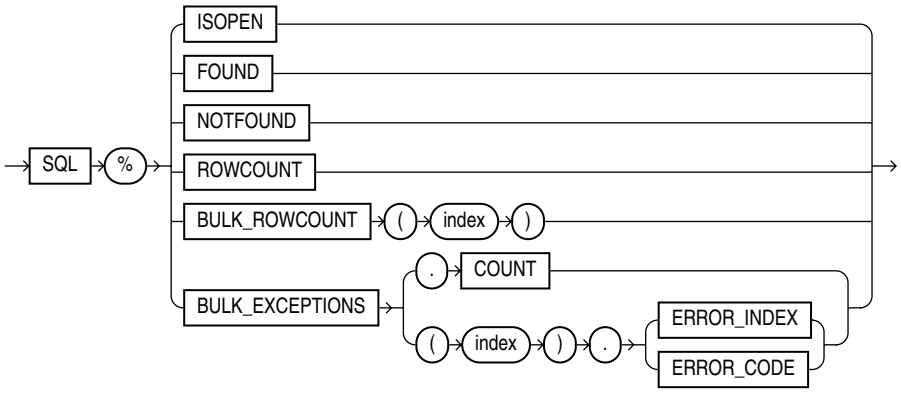
> **Note:** You can use cursor attributes only in procedural statements, not in SQL statements.

Topics

- Syntax
- Semantics
- 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

Composite attribute that is like an associative array whose *i*th element is the number of rows affected by the *i*th DML statement in the most recently completed FORALL statement. For more information, see "Getting Number of Rows Affected by FORALL Statement" on page 12-21.

Restriction on SQL%BULK_ROWCOUNT You cannot assign the value of SQL%BULK_ROWCOUNT(*index*) to another collection.

SQL%BULK_EXCEPTIONS

Composite attribute that is like an associative array of information about the DML statements that failed during the most recently run FORALL statement. SQL%BULK_EXCEPTIONS.COUNT is the number of DML statements that failed. If SQL%BULK_EXCEPTIONS.COUNT is not zero, then for each index value *i* from 1 through SQL%BULK_EXCEPTIONS.COUNT:

- SQL%BULK_EXCEPTIONS(*i*).ERROR_INDEX is the number of the DML statement that failed.
- SQL%BULK_EXCEPTIONS(*i*).ERROR_CODE is the Oracle Database error code for the failure.

Typically, this attribute appears in an exception handler for a FORALL statement that has a SAVE EXCEPTIONS clause. For more information, see "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-19.

Examples

- Example 6–3, "SQL%FOUND Implicit Cursor Attribute" on page 6-6
- Example 6–4, "SQL%ROWCOUNT Implicit Cursor Attribute" on page 6-7
- Example 6–15, "%FOUND Explicit Cursor Attribute" on page 6-20
- Example 6–14, "%ISOPEN Explicit Cursor Attribute" on page 6-20
- Example 6–16, "%NOTFOUND Explicit Cursor Attribute" on page 6-21
- Example 6–17, "%ROWCOUNT Explicit Cursor Attribute" on page 6-22
- Example 12–13, "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-20
- Example 12–14, "Showing Number of Rows Affected by Each DELETE in FORALL" on page 12-21
- Example 12–15, "Showing Number of Rows Affected by Each INSERT SELECT in FORALL" on page 12-22

Related Topics

In this chapter:

- "FORALL Statement" on page 13-77
- "Named Cursor Attribute" on page 13-98

In other chapters:

- "Implicit Cursors" on page 6-5
- "Query Result Set Processing" on page 6-22

INLINE Pragma

The `INLINE` pragma specifies whether a subprogram invocation is to be inlined. Inlining replaces a subprogram invocation with a copy of the invoked subprogram (if the invoked and invoking subprograms are in the same program unit).

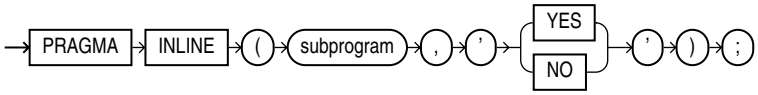
> **Note:** The `INLINE` pragma affects only the immediately following declaration or statement, and only some kinds of statements. For details, see "Subprogram Inlining" on page 12-1.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***inline_pragma* ::=**



Semantics

subprogram

Name of a subprogram. If *`subprogram`* is overloaded, then the `INLINE` pragma applies to every subprogram with that name.

YES

If `PLSQL_OPTIMIZE_LEVEL=2`, `'YES'` specifies that the subprogram invocation is to be inlined.

If `PLSQL_OPTIMIZE_LEVEL=3`, `'YES'` specifies that the subprogram invocation has a high priority for inlining.

NO

Specifies that the subprogram invocation is not to be inlined.

Examples

- Example 12–1, "Specifying that Subprogram Is To Be Inlined" on page 12-3
- Example 12–2, "Specifying that Overloaded Subprogram Is To Be Inlined" on page 12-3
- Example 12–3, "Specifying that Subprogram Is Not To Be Inlined" on page 12-3
- Example 12–4, "PRAGMA INLINE ... 'NO' Overrides PRAGMA INLINE ... 'YES'" on page 12-3

Related Topics

- "Subprogram Inlining" on page 12-1

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.

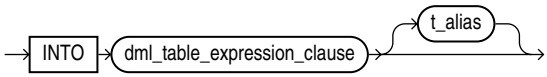
> **See Also:** *Oracle Database SQL Language Reference* for the syntax of the SQL `INSERT` statement

Topics

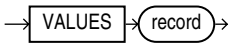
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

insert_into_clause ::=



values_clause ::=



Semantics

insert_into_clause

dml_table_expression_clause

Typically a table name. For complete information, see *Oracle Database SQL Language Reference*.

t_alias

An alias for *`dml_table_expression_clause`*.

values_clause

record

Name of a record variable of type `RECORD` or `%ROWTYPE`. *`record`* must represent a row of the item explained 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.

See Also: *Oracle Database SQL Language Reference* for the complete syntax of the `INSERT` statement

Examples

- Example 5–46, "Initializing Table by Inserting Record of Default Values" on page 5-50

Related Topics

In this chapter:

- "Record Variable Declaration" on page 13-111
- "%ROWTYPE Attribute" on page 13-121

In other chapters:

- "Inserting Records into Tables" on page 5-50
- "Restrictions on Record Inserts and Updates" on page 5-52

Named Cursor Attribute

Every named cursor (explicit cursor or cursor variable) has four attributes, each of which returns information about the execution of a DML statement.

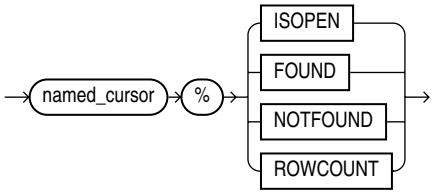
> **Note:** You can use cursor attributes only in procedural statements, not in SQL statements.

Topics

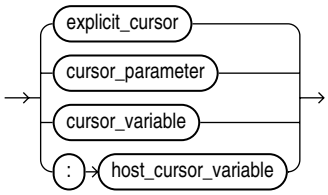
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

named_cursor_attribute ::=



named_cursor ::=



Semantics

named_cursor_attribute

%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.

named_cursor

explicit_cursor

Name of an explicit cursor.

cursor_parameter

Name of a formal cursor parameter.

cursor_variable

Name of a cursor variable.

:*host_cursor_variable*

Name of a cursor variable that was 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*.

Examples

- Example 6–14, "%ISOPEN Explicit Cursor Attribute" on page 6-20
- Example 6–15, "%FOUND Explicit Cursor Attribute" on page 6-20
- Example 6–16, "%NOTFOUND Explicit Cursor Attribute" on page 6-21
- Example 6–17, "%ROWCOUNT Explicit Cursor Attribute" on page 6-22

Related Topics

In this chapter:

- "Cursor Variable Declaration" on page 13-42
- "Explicit Cursor Declaration and Definition" on page 13-57
- "Implicit Cursor Attribute" on page 13-91

In other chapters:

- "Explicit Cursor Attributes" on page 6-19

NULL Statement

The NULL statement is a "no-op" (no operation)—it only passes control to the next statement.

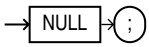
Note: The NULL statement and the BOOLEAN value NULL are not related.

Topics

- Syntax
- Examples
- Related Topics

Syntax

***null_statement* ::=**



Examples

- Example 4–30, "GOTO Statement Goes to Labeled NULL Statement" 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

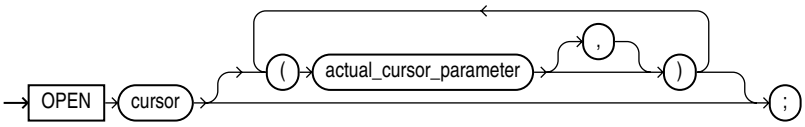
The `OPEN` statement opens an explicit cursor, allocates database resources to process the associated query, identifies the result set, and positions the cursor before the first row of the result set. If the query has a `FOR UPDATE` clause, the `OPEN` statement locks the rows of the result set.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***open_statement* ::=**



Semantics

cursor

Name of an explicit cursor that is not open.

***actual_cursor_parameter* [, *actual_cursor_parameter*]...**

List of actual parameters for the cursor that you are opening. An actual parameter can be a constant, initialized variable, literal, or expression. The data type of each actual parameter must be compatible with the data type of the corresponding formal parameter.

You can specify actual cursor parameters with either positional notation or named notation. For information about these notations, see "Positional, Named, and Mixed Notation for Actual Parameters" on page 8-21.

If the cursor specifies a default value for a parameter, you can omit that parameter from the parameter list. If the cursor has no parameters, or specifies a default value for every parameter, you can either omit the parameter list or specify an empty parameter list.

Examples

- Example 6–11, "Explicit Cursor that Accepts Parameters" on page 6-15
- Example 6–12, "Cursor Parameters with Default Values" on page 6-16

Related Topics

In this chapter:

- "CLOSE Statement" on page 13-23

- "Explicit Cursor Declaration and Definition" on page 13-57
- "FETCH Statement" on page 13-71
- "OPEN FOR Statement" on page 13-103

In other chapters:

- "Opening and Closing Explicit Cursors" on page 6-9
- "Explicit Cursors that Accept Parameters" on page 6-15

OPEN FOR Statement

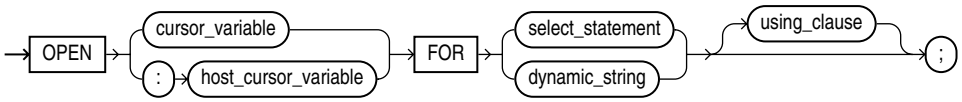
The `OPEN FOR` statement associates a cursor variable with a query, allocates database resources to process the query, identifies the result set, and positions the cursor before the first row of the result set. If the query has a `FOR UPDATE` clause, then the `OPEN FOR` statement locks the rows of the result set.

Topics

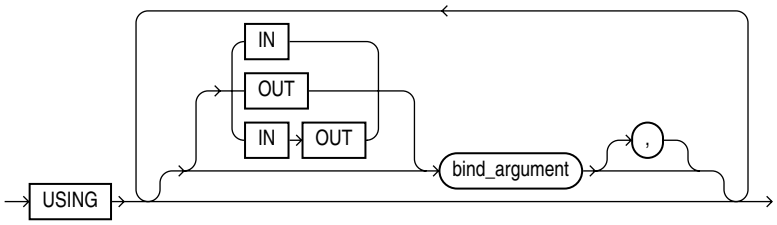
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***open_for_statement* ::=**



***using_clause* ::=**



Semantics

open_for_statement

cursor_variable

Name of a cursor variable. If *`cursor_variable`* is the formal parameter of a subprogram, then it must not have a return type. For information about cursor variables as subprogram parameters, see "Cursor Variables as Subprogram Parameters" on page 6-35.

:host_cursor_variable

Name of a cursor variable that was 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`*.

The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

select_statement

SQL `SELECT` statement (not a PL/SQL `SELECT INTO` statement). Typically, *`select_statement`* returns multiple rows.

> **See:** *Oracle Database SQL Language Reference* for `SELECT` statement syntax

dynamic_string

String literal, string variable, or string expression of the data type `CHAR`, `VARCHAR2`, or `CLOB`, which represents a SQL `SELECT` statement. Typically, *`dynamic_statement`* represents a SQL `SELECT` statement that returns multiple rows.

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.

bind_argument

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.

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. **Default:** `IN`.

Examples

- Example 6–26, "Fetching Data with Cursor Variables" on page 6-31
- Example 6–30, "Procedure to Open Cursor Variable for One Query" on page 6-35
- Example 6–31, "Opening Cursor Variable for Chosen Query (Same Return Type)" on page 6-36
- Example 6–32, "Opening Cursor Variable for Chosen Query (Different Return Types)" on page 6-36
- 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 13-23
- "Cursor Variable Declaration" on page 13-42
- "EXECUTE IMMEDIATE Statement" on page 13-52

- "FETCH Statement" on page 13-71
- "OPEN Statement" on page 13-101

In other chapters:

- "Opening and Closing Cursor Variables" on page 6-30
- "OPEN FOR, FETCH, and CLOSE Statements" on page 7-4

PIPE ROW Statement

The `PIPE ROW` statement, which can appear only in the body of a pipelined table function, returns a table row (but not control) to the invoker of the function.

Note:

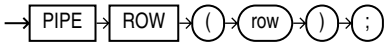
- If a pipelined table function is part of an autonomous transaction, then it must `COMMIT` or `ROLLBACK` before each `PIPE ROW` statement, to avoid an error in the invoking subprogram.
- To improve performance, the PL/SQL run-time system delivers the piped rows to the invoker in batches.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***pipe_row_statement* ::=**



Semantics

row

Row (table element) that the function returns to its invoker, represented by an expression whose type is that of the table element.

If the expression is a record variable, it must be explicitly declared with the data type of the table element. It cannot be declared with a data type that is only structurally identical to the element type. For example, if the element type has a name, then the record variable cannot be declared explicitly with `%TYPE` or `%ROWTYPE` or implicitly with `%ROWTYPE` in a cursor `FOR LOOP` statement.

Examples

- Example 12–29, "Creating and Invoking Pipelined Table Function" on page 12-41
- Example 12–30, "Pipelined Table Function Transforms Each Row to Two Rows" on page 12-42
- Example 12–32, "Pipelined Table Function with Two Cursor Variable Parameters" on page 12-44
- Example 12–33, "Pipelined Table Function as Aggregate Function" on page 12-45
- Example 12–34, "Pipelined Table Function Does Not Handle NO_DATA_NEEDED" on page 12-47

- Example 12–35, "Pipelined Table Function Handles NO_DATA_NEEDED" on page 12-48

Related Topics

In this chapter:

- "Function Declaration and Definition" on page 13-83

In other chapters:

- "Creating Pipelined Table Functions" on page 12-40

Procedure Declaration and Definition

A **procedure** is a subprogram that performs a specific action. A procedure invocation (or call) is a statement.

Before invoking a procedure, you must declare and define it. You can either declare it first (with ***procedure_declaration***) and then define it later in the same block, subprogram, or package (with ***procedure_definition***) or declare and define it at the same time (with ***procedure_definition***).

A procedure declaration is also called a **procedure specification** or **procedure spec**.

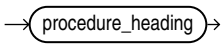
Note: This topic applies to nested procedures. For information about standalone stored procedures, see "CREATE PROCEDURE Statement" on page 14-50. For information about package procedures, see "CREATE PACKAGE Statement" on page 14-43.

Topics

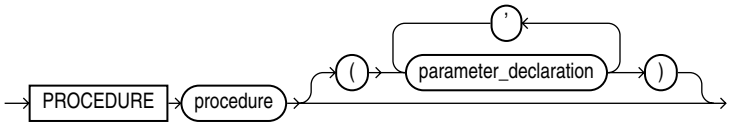
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

procedure_declaration ::=

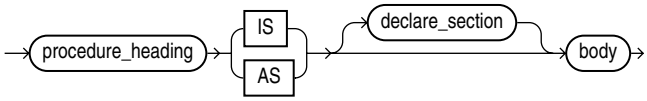


procedure_heading ::=



See "parameter_declaration ::=" on page 13-80.

procedure_definition ::=



See:

- "body ::=" on page 13-12
- "call_spec ::=" on page 14-51

- "declare_section ::=" on page 13-9

Semantics

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.

procedure_heading

procedure

Name of the procedure that you are declaring or defining.

procedure_definition

Either defines a procedure that was declared earlier or both declares and defines a procedure.

declare_section

Declares items that are local to the procedure, can be referenced in `body`, and cease to exist when the procedure completes execution.

body

Required executable part and optional exception-handling part of the procedure.

call_spec, EXTERNAL

See "call_spec" on page 14-52 and "EXTERNAL" on page 14-52.

Restriction on *call_spec*, EXTERNAL These clauses can appear only in a package specification or package body.

Examples

- Example 8–1, "Declaring, Defining, and Invoking a Simple PL/SQL Procedure" on page 8-3

Related Topics

In this chapter:

- "Formal Parameter Declaration" on page 13-80
- "Function Declaration and Definition" on page 13-83

In other chapters:

- Chapter 8, "PL/SQL Subprograms"
- "CREATE PROCEDURE Statement" on page 14-50

RAISE Statement

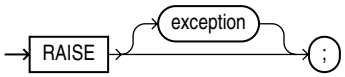
The `RAISE` statement explicitly raises an exception. Outside an exception handler, you must specify the exception name. Inside an exception handler, if you omit the exception name, the `RAISE` statement reraises the current exception.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***raise_statement* ::=**



Semantics

exception

Name of an exception, either predefined (see Table 11–3) or user-declared (see "Exception Declaration" on page 13-48).

`exception` is optional only in an exception handler, where the default is the current exception (see "Reraising Current Exception with RAISE Statement" on page 11-15).

Examples

- Example 11–9, "Declaring, Raising, and Handling User-Defined Exception" on page 11-13
- Example 11–10, "Explicitly Raising Predefined Exception" on page 11-14
- Example 11–11, "Reraising Exception" on page 11-15

Related Topics

In this chapter:

- "Exception Declaration" on page 13-48
- "Exception Handler" on page 13-50

In other chapters:

- "Raising Exceptions Explicitly" on page 11-13

Record Variable Declaration

A **record variable** is a composite variable whose internal components, called fields, can have different data types. The value of a record variable and the values of its fields can change.

You reference an entire record variable by its name. You reference a record field with the syntax `record.field`.

You can create a record variable in any of these ways:

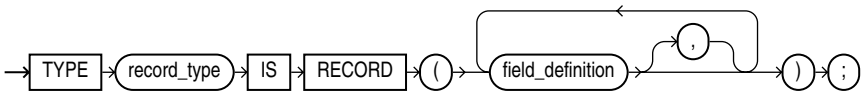
- Define a record type and then declare a variable of that type.
- Use `%ROWTYPE` to declare a record variable that represents either a full or partial row of a database table or view.
- Use `%TYPE` to declare a record variable of the same type as a previously declared record variable.

Topics

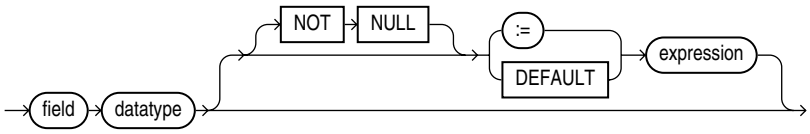
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

record_type_definition ::=



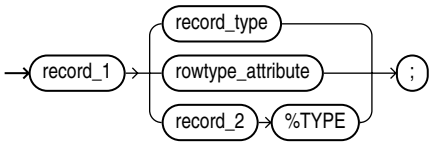
field_definition ::=



See:

- "datatype ::=" on page 13-26
- "expression ::=" on page 13-61

record_variable_declaration ::=



See "rowtype_attribute ::=" on page 13-121.

Semantics

record_type_definition

record_type

Name of the record type that you are defining.

field_definition

field

Name of the field that you are defining.

datatype

Data type of the field that you are defining.

NOT NULL

Imposes the `NOT NULL` constraint on the field that you are defining. For information about this constraint, see "NOT NULL Constraint" on page 2-13.

expression

Expression whose data type is compatible with *`datatype`*. When *`record_variable_declaration`* is elaborated, the value of *`expression`* is assigned to *`record.field`*. This value is the initial value of the field.

record_variable_declaration

record_1

Name of the record variable that you are declaring.

record_type

Name of a previously defined record type. *`record_type`* is the data type of *`record_1`*.

rowtype_attribute

See "%ROWTYPE Attribute" on page 13-121.

record_2

Name of a previously declared record variable.

%TYPE

See "%TYPE Attribute" on page 13-134.

Examples

- Example 5–32, "RECORD Type Definition and Variable Declaration" on page 5-39
- Example 5–33, "RECORD Type with RECORD Field (Nested Record)" on page 5-40
- Example 5–34, "RECORD Type with Varray Field" on page 5-40

Related Topics

In this chapter:

- "Collection Variable Declaration" on page 13-25
- "%ROWTYPE Attribute" on page 13-121

In other chapters:

- "Record Topics" on page 5-2

RESTRICT_REFERENCES Pragma

> **Note:** The `RESTRICT_REFERENCES` pragma is deprecated. Oracle recommends using `DETERMINISTIC` and `PARALLEL_ENABLE` (explained in "Function Declaration and Definition" on page 13-83) 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 invocation of the subprogram might produce different results or encounter errors.

Restriction on RESTRICT_REFERENCES Pragma

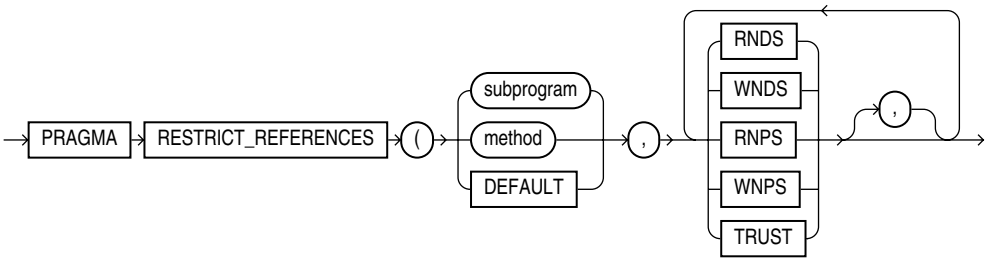
This pragma can appear only in a package specification or ADT specification. Typically, this pragma is specified for functions. If a function invokes procedures, specify this pragma for those procedures also.

Topics

- Syntax
- Semantics

Syntax

***restrict_references_pragma* ::=**



Semantics

subprogram

Name of a user-defined subprogram, typically a function. If *`subprogram`* is overloaded, the pragma applies only to the most recent subprogram declaration.

method

Name of a `MEMBER` subprogram (see "CREATE TYPE Statement" on page 14-68, specifically "element_spec ::=" on page 14-70).

DEFAULT

Applies the pragma 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 package variables)

Restriction on RNPS 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 package variables).

Restriction on WNPS 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`, PL/SQL does not check the subprogram body for violations of the constraints listed in the pragma. 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.

Note: 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 ends the execution of the subprogram or anonymous block that contains it.

In a function, the RETURN statement assigns a specified value to the function identifier and returns control to the invoker, where execution resumes immediately after the invocation (possibly inside the invoking statement). Every execution path in a function must lead to a RETURN statement (otherwise, the PL/SQL compiler issues compile-time warning PLW-05005).

In a procedure, the RETURN statement returns control to the invoker, where execution resumes immediately after the invocation.

In an anonymous block, the RETURN statement exits its own block and all enclosing blocks.

A subprogram or anonymous block can contain multiple RETURN statements.

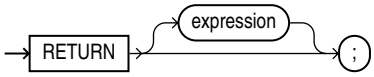
> **Note:** The RETURN statement differs from the RETURN clause in a function heading, which specifies the data type of the return value.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***return_statement* ::=**



See "expression ::=" on page 13-61.

Semantics

expression

Optional when the RETURN statement is in a pipelined table function. Required when the RETURN statement is in any other function. Not allowed when the RETURN statement is in a procedure or anonymous block.

The RETURN statement assigns the value of *expression* to the function identifier. Therefore, the data type of *expression* must be compatible with the data type in the RETURN clause of the function. For information about expressions, see "Expression" on page 13-61.

Examples

- Example 8–3, "Execution Resumes After RETURN Statement in Function" on page 8-6
- Example 8–4, "Function Where Not Every Execution Path Leads to RETURN Statement" on page 8-6
- Example 8–5, "Function Where Every Execution Path Leads to RETURN Statement" on page 8-7
- Example 8–6, "Execution Resumes After RETURN Statement in Procedure" on page 8-7
- Example 8–7, "Execution Resumes After RETURN Statement in Anonymous Block" on page 8-8

Related Topics

In this chapter:

- "Block" on page 13-9
- "Function Declaration and Definition" on page 13-83
- "Procedure Declaration and Definition" on page 13-108

In other chapters:

- "RETURN Statement" on page 8-5

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 affects no rows, then 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.

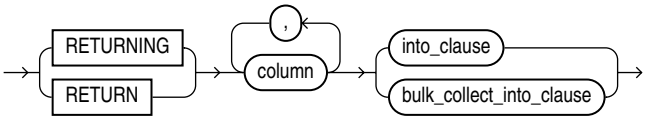
Note: You cannot use the `RETURNING INTO` clause for remote or parallel deletes.

Topics

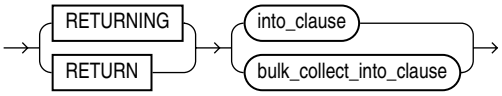
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

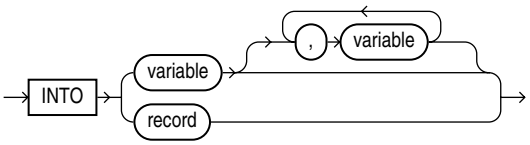
***static_returning_clause* ::=**



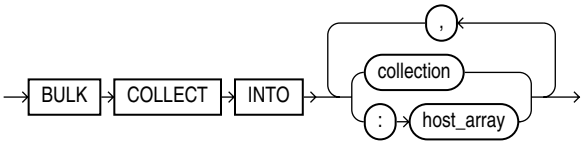
***dynamic_returning_clause* ::=**



***into_clause* ::=**



***bulk_collect_into_clause* ::=**



Semantics

static_returning_clause

column

Expression whose value is the name of a column of a database table.

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 13-52) returns a single row.

record

The name of a record variable 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.

variable

Either the name of a scalar variable in which to store a column that the statement returns or the name of a host 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. The data type of a host cursor variable is compatible with the return type of any PL/SQL cursor variable.

Restriction on *variable* *variable* cannot have the data type BOOLEAN.

bulk_collect_into_clause

Specifies one or more existing collections or host 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* or *host_array*.

For the reason to use this clause, see "Bulk SQL and Bulk Binding" on page 12-11.

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 13-52) can return multiple rows.

collection

Name of a collection variable in which to store the rows that the statement returns.

Restrictions on *collection*

- *collection* cannot be the name of an associative array that is indexed by a string.
- When the statement requires implicit data type conversions, *collection* cannot be the name of a collection of a composite type.

:host_array

Name of an array 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*.

Examples

- Example 5–45, "UPDATE Statement Assigns Values to Record Variable" on page 5-49
- Example 6–1, "Static SQL Statements" on page 6-2
- Example 12–25, "Returning Deleted Rows in Two Nested Tables" on page 12-36
- Example 12–26, "DELETE with RETURN BULK COLLECT INTO in FORALL Statement" on page 12-36

Related Topics

In this chapter:

- "DELETE Statement Extension" on page 13-45
- "EXECUTE IMMEDIATE Statement" on page 13-52
- "FETCH Statement" on page 13-71
- "SELECT INTO Statement" on page 13-125
- "UPDATE Statement Extensions" on page 13-136

In other chapters:

- "SQL Statements that Return Rows in PL/SQL Record Variables" on page 5-49
- "EXECUTE IMMEDIATE Statement" on page 7-2
- "RETURNING INTO Clause with BULK COLLECT Clause" on page 12-35

%ROWTYPE Attribute

The `%ROWTYPE` attribute lets you declare a record that represents either a full or partial row of a database table or view. For every column of the full or partial row, the record has a field with the same name and data type. If the structure of the row changes, then the structure of the record changes accordingly.

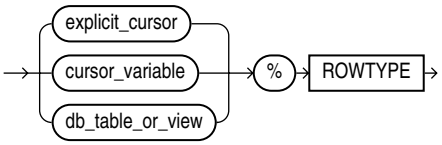
The record fields do not inherit the constraints or initial values of the corresponding columns.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***rowtype_attribute* ::=**



Semantics

explicit_cursor

Name of an explicit cursor. For every column selected by the query associated with *`explicit_cursor`*, the record has a field with the same name and data type.

cursor_variable

Name of a strong cursor variable. For every column selected by the query associated with *`cursor_variable`*, the record has a field with the same name and data type.

db_table_or_view

Name of a database table or view that is accessible when the declaration is elaborated. For every column of *`db_table_or_view`*, the record has a field with the same name and data type.

Examples

- Example 5–36, "%ROWTYPE Variable Represents Full Database Table Row" on page 5-42
- Example 5–37, "%ROWTYPE Variable Does Not Inherit Initial Values or Constraints" on page 5-43
- Example 5–38, "%ROWTYPE Variable Represents Partial Database Table Row" on page 5-44
- Example 5–39, "%ROWTYPE Variable Represents Join Row" on page 5-44

- Example 5–41, "Assigning %ROWTYPE Record to RECORD Type Record" on page 5-45

Related Topics

In this chapter:

- "Cursor Variable Declaration" on page 13-42
- "Explicit Cursor Declaration and Definition" on page 13-57
- "Record Variable Declaration" on page 13-111
- "%TYPE Attribute" on page 13-134

In other chapters:

- "%ROWTYPE Attribute" on page 5-42

Scalar Variable Declaration

A scalar variable stores a value with no internal components. The value can change.

A scalar variable declaration specifies the name and data type of the variable and allocates storage for it. The declaration can also assign an initial value and impose the `NOT NULL` constraint.

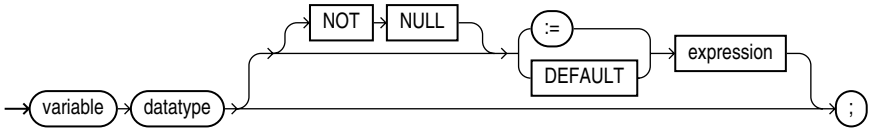
You reference a scalar variable by its name.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***variable_declaration* ::=**



See "expression ::=" on page 13-61.

Semantics

variable

Name of the variable that you are declaring.

datatype

Name of a scalar data type, including any qualifiers for size, precision, and character or byte semantics. For information about scalar data types, see Chapter 3, "PL/SQL Data Types".

NOT NULL

Imposes the `NOT NULL` constraint on the variable. For information about this constraint, see "NOT NULL Constraint" on page 2-13.

expression

Value to be assigned to the variable when the declaration is elaborated. *`expression`* and *`variable`* must have compatible data types.

Examples

- Example 2–9, "Scalar Variable Declarations" on page 2-12
- Example 2–11, "Variable and Constant Declarations with Initial Values" on page 2-13

- Example 2–12, "Variable Initialized to NULL by Default" on page 2-13
- Example 2–13, "Variable Declaration with NOT NULL Constraint" on page 2-14

Related Topics

In this chapter:

- "Assignment Statement" on page 13-3
- "Collection Variable Declaration" on page 13-25
- "Constant Declaration" on page 13-36
- "Expression" on page 13-61
- "Record Variable Declaration" on page 13-111
- "%ROWTYPE Attribute" on page 13-121
- "%TYPE Attribute" on page 13-134

In other chapters:

- "Variable Declarations" on page 2-12
- Chapter 3, "PL/SQL Data Types"

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 variables (which the SQL `SELECT` statement does not do).

Caution: The `SELECT INTO` statement with the `BULK COLLECT` clause is vulnerable to aliasing, which can cause unexpected results. For details, see "SELECT BULK COLLECT INTO Statements and Aliasing" on page 12-26.

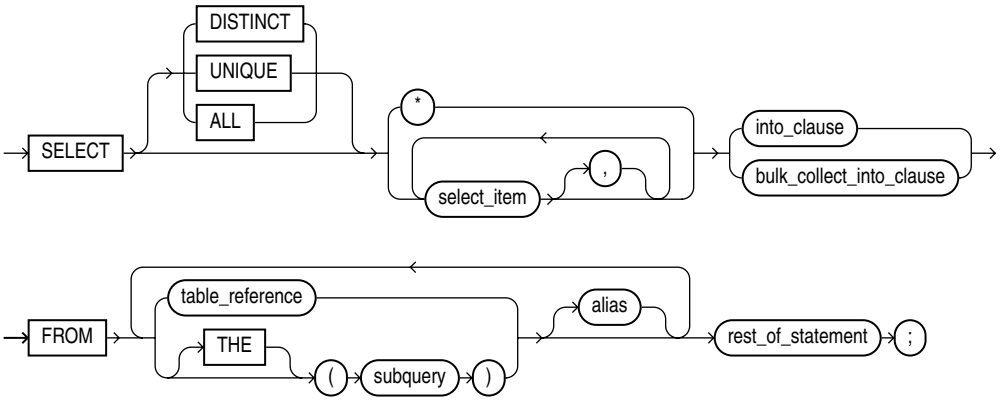
See Also: *Oracle Database SQL Language Reference* for the syntax of the SQL `SELECT` statement

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

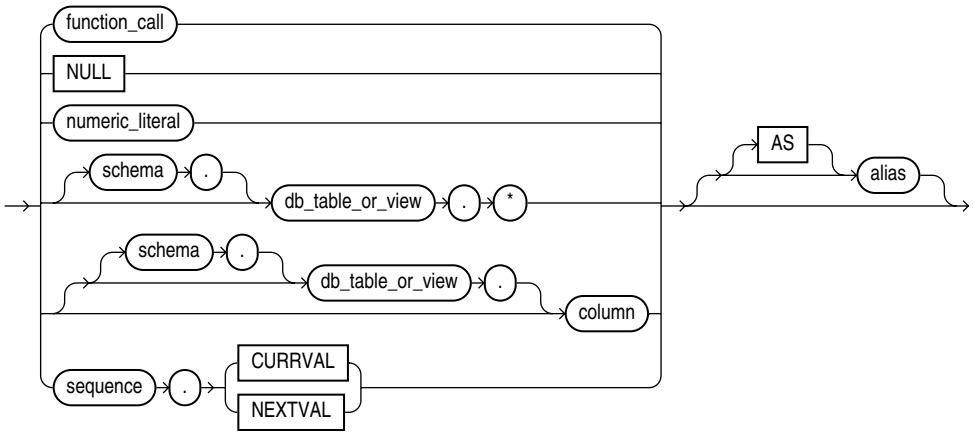
***select_into_statement* ::=**



See:

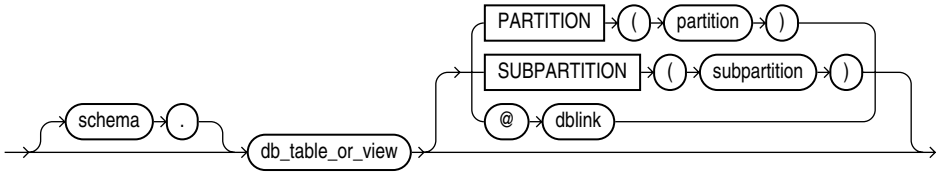
- "bulk_collect_into_clause ::=" on page 13-118
- "into_clause ::=" on page 13-118
- "table_reference ::=" on page 13-126

***select_item* ::=**



See "function_call ::=" on page 13-65.

***table_reference* ::=**



Semantics

select_into_statement

DISTINCT or UNIQUE

Causes the database to return only one copy of each set of duplicate rows selected. Duplicate rows are those with matching values for each *select_item*. These two keywords are synonymous.

Restrictions on DISTINCT and UNIQUE

- The total number of bytes in all *select_item* expressions is limited to the size of a data block minus some overhead. This size is specified by the initialization parameter DB_BLOCK_SIZE.
- No *select_item* expression can contain a LOB column.

ALL

(Default) Causes the database to return all rows selected, including all copies of duplicates.

*

Selects all columns.

into_clause

With this clause, the SELECT INTO statement retrieves one or more columns from a single row and stores them in either one or more scalar variables or one record variable. For more information, see "into_clause" on page 13-119.

bulk_collect_into_clause

With this clause, the `SELECT INTO` statement retrieves an entire result set and stores it in one or more collection variables. For more information, see "bulk_collect_into_clause" on page 13-119.

subquery

SQL `SELECT` statement (not a PL/SQL `SELECT INTO` statement).

alias

Another (usually short) name for the referenced column, table, or view.

rest_of_statement

Anything that can follow the `FROM` clause in a SQL `SELECT` statement, described in *Oracle Database SQL Language Reference*.

select_item

If the `SELECT INTO` statement returns no rows, PL/SQL raises the predefined 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.

numeric_literal

Literal of a numeric data type.

schema

Name of the schema that contains the table or view. **Default:** your schema.

db_table_or_view

Name of a database table or view.

column

Name of a column of *`db_table_or_view`*.

*

Selects all columns of *`db_table_or_view`*.

sequence

Name of a sequence.

CURRVAL

Current value in *`sequence`*.

NEXTVAL

Next value in *`sequence`*.

alias

Another (usually short) name for the referenced column, table, or view.

table_reference

Reference to a table or view for which you have the `SELECT` privilege, which is accessible when you run the `SELECT INTO` statement.

schema

Name of the schema that contains the table or view. **Default:** your schema.

table

Name of a database table.

view

Name of a database view.

PARTITION *partition* or SUBPARTITION *subpartition*

See *Oracle Database SQL Language Reference*.

@ dblink

Database link, described in *Oracle Database SQL Language Reference*. Do not put space between @ and `dblink`.

Examples

- Example 2–25, "Assigning Value to Variable with SELECT INTO Statement" on page 2-22
- Example 5–43, "SELECT INTO Assigns Values to Record Variable" on page 5-47
- Example 6–36, "ROLLBACK Statement" on page 6-42
- Example 6–37, "SAVEPOINT and ROLLBACK Statements" on page 6-43
- Example 6–44, "Declaring Autonomous Function in Package" on page 6-51
- Example 7–13, "Validation Checks Guarding Against SQL Injection" on page 7-16
- Example 12–16, "Bulk-Selecting Two Database Columns into Two Nested Tables" on page 12-24
- Example 12–17, "Bulk-Selecting into Nested Table of Records" on page 12-25
- Example 12–21, "Limiting Bulk Selection with ROWNUM and SAMPLE" on page 12-30

Related Topics

In this chapter:

- "Assignment Statement" on page 13-3
- "FETCH Statement" on page 13-71
- "%ROWTYPE Attribute" on page 13-121

In other chapters:

- "Assigning Values to Variables with the SELECT INTO Statement" on page 2-22
- "SELECT INTO Statement for Assigning Row to Record Variable" on page 5-47
- "Query Result Set Processing With SELECT INTO Statements" on page 6-23

- "SELECT INTO Statement with BULK COLLECT Clause" on page 12-24

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 invocation 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.

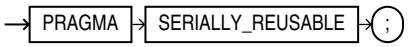
The `SERIALLY_REUSABLE` pragma can appear in the *`declare_section`* of the specification of a bodiless package, or in both the specification and body of a package, but not in only the body of a package.

Topics

- Syntax
- Examples
- Related Topics

Syntax

***serially_resuable_pragma* ::=**



Examples

- Example 10–4, "Creating SERIALLY_REUSABLE Packages" on page 10-8
- Example 10–5, "Effect of SERIALLY_REUSABLE Pragma" on page 10-9
- Example 10–6, "Cursor in SERIALLY_REUSABLE Package Open at Call Boundary" on page 10-10

Related Topics

In this chapter:

- "AUTONOMOUS_TRANSACTION Pragma" on page 13-6
- "EXCEPTION_INIT Pragma" on page 13-46
- "INLINE Pragma" on page 13-94
- "RESTRICT_REFERENCES Pragma" on page 13-114

In other chapters:

- "SERIALLY_REUSABLE Packages" on page 10-7

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 internally defined exception, 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 **(default)** or the error code associated with the exception by the `EXCEPTION_INIT` pragma.

A SQL statement cannot invoke `SQLCODE`.

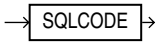
If a function invokes `SQLCODE`, and you use the `RESTRICT_REFERENCES` pragma to assert the purity of the function, then you cannot specify the constraints `WNPS` and `RNPS`.

Topics

- Syntax
- Examples
- Related Topics

Syntax

***sqlcode_function* ::=**



Examples

- Example 11–22, "Displaying SQLCODE and SQLERRM Values" on page 11-25

Related Topics

In this chapter:

- "Block" on page 13-9
- "EXCEPTION_INIT Pragma" on page 13-46
- "Exception Handler" on page 13-50
- "RESTRICT_REFERENCES Pragma" on page 13-114
- "SQLERRM Function" on page 13-132

In other chapters:

- "Error Code and Error Message Retrieval" on page 11-24

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 code.

> **Note:** The language of the error message depends on the NLS_LANGUAGE parameter. For information about this parameter, see *Oracle Database Globalization Support Guide*.

A SQL statement cannot invoke SQLERRM.

If a function invokes SQLERRM, and you use the RESTRICT_REFERENCES pragma to assert the purity of the function, then you cannot specify the constraints WNPS and RNPS.

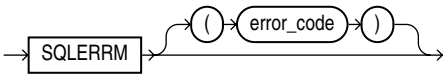
> **Note:** DBMS_UTILTY.FORMAT_ERROR_STACK is recommended over SQLERRM, unless you use the FORALL statement with its SAVE EXCEPTIONS clause. For more information, see "Error Code and Error Message Retrieval" on page 11-24.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***sqlerrm_function* ::=**



Semantics

error_code

Expression whose value is an Oracle Database error code. For a list of Oracle Database error codes, see *Oracle Database Error Messages*. **Default:** error code associated with the current value of SQLCODE.

Like SQLCODE, SQLERRM without *error_code* is useful only in an exception handler. Outside an exception handler, or if the value of *error_code* is zero, SQLERRM returns ORA-0000.

If the value of *error_code* is +100, SQLERRM returns ORA-01403.

If the value of *error_code* is a positive number other than +100, SQLERRM returns this message:

```
-error_code: non-ORACLE exception
```

If the value of *error_code* is a negative number whose absolute value is an Oracle Database error code, SQLERRM returns the error message associated with that error code. 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_code* is a negative number whose absolute value is not an Oracle Database error code, SQLERRM returns this message:

```
ORA-error_code: Message error_code 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 11–22, "Displaying SQLCODE and SQLERRM Values" on page 11-25
- Example 12–13, "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-20
- Example 12–13, "Handling FORALL Exceptions After FORALL Statement Completes" on page 12-20

Related Topics

In this chapter:

- "Block" on page 13-9
- "EXCEPTION_INIT Pragma" on page 13-46
- "RESTRICT_REFERENCES Pragma" on page 13-114
- "SQLCODE Function" on page 13-131

In other chapters:

- "Error Code and Error Message Retrieval" on page 11-24

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, collection element, record field, or subprogram parameter to be of the same data type as a previously declared variable or column (without knowing what that type is). 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 and size
- Constraints (unless the referenced item is a column)

The referencing item does not inherit the initial value of the referenced item.

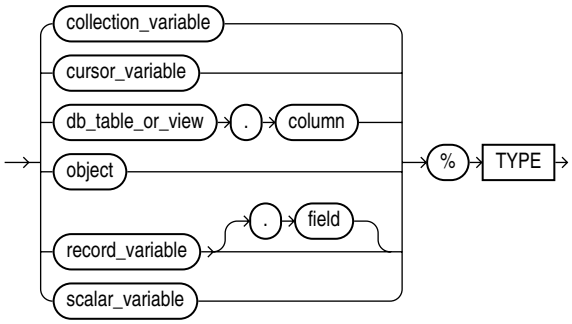
If the declaration of the referenced item changes, then the declaration of the referencing item changes accordingly.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

***type_attribute* ::=**



Semantics

collection_variable

Name of a collection variable.

Restriction on *collection_variable* In a constant declaration, *`collection_variable`* cannot be an associative array variable.

cursor_variable

Name of a cursor variable.

db_table_or_view
Name of a database table or view that is accessible when the declaration is elaborated.

column
Name of a column of *db_table_or_view*.

object
Name of an instance of an ADT.

record_variable
Name of a record variable.

field
Name of a field of *record_variable*.

scalar_variable
Name of a scalar variable.

Examples

- Example 2–15, "Declaring Variable of Same Type as Column" on page 2-15
- Example 2–16, "Declaring Variable of Same Type as Another Variable" on page 2-15

Related Topics

In this chapter:

- "Constant Declaration" on page 13-36
- "%ROWTYPE Attribute" on page 13-121
- "Scalar Variable Declaration" on page 13-123

In other chapters:

- "%TYPE Attribute" on page 2-14

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.

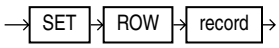
> **See Also:** *Oracle Database SQL Language Reference* for the syntax of the SQL UPDATE statement

Topics

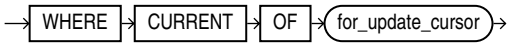
- Syntax
- Semantics
- Examples
- Related Topics

Syntax

update_set_clause ::=



where_clause ::=



Semantics

record

Name of a record variable that represents 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 of a FOR UPDATE cursor; that is, an explicit cursor associated with a FOR SELECT UPDATE statement.

Examples

- Example 5–47, "Updating Rows with Record" on page 5-51
- Example 6–40, "FOR UPDATE Cursor in CURRENT OF Clause of UPDATE Statement" on page 6-47

Related Topics

In this chapter:

- "Explicit Cursor Declaration and Definition" on page 13-57
- "Record Variable Declaration" on page 13-111
- "%ROWTYPE Attribute" on page 13-121

In other chapters:

- "Updating Rows with Records" on page 5-51
- "Restrictions on Record Inserts and Updates" on page 5-52
- "SELECT FOR UPDATE and FOR UPDATE Cursors" on page 6-46

WHILE LOOP Statement

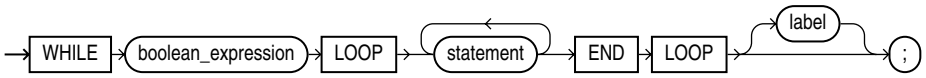
The `WHILE LOOP` statement runs one or more statements while a condition is `TRUE`. The `WHILE LOOP` statement ends when the condition becomes `FALSE` or `NULL`, or when a statement inside the loop transfers control outside the loop or raises an exception.

Topics

- Syntax
- Semantics
- Examples
- Related Topics

Syntax

while_loop_statement ::=



See:

- "boolean_expression ::=" on page 13-62
- "statement ::=" on page 13-13

Semantics

boolean_expression

Expression whose value is `TRUE`, `FALSE`, or `NULL`.

`boolean_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.

statement

To prevent an infinite loop, at least one statement must change the value of *`boolean_expression`* to `FALSE` or `NULL`, transfer control outside the loop, or raise an exception. The statements that can transfer control outside the loop are:

- "CONTINUE Statement" on page 13-38 (when it transfers control to the next iteration of an enclosing labeled loop)
- "EXIT Statement" on page 13-55
- "GOTO Statement" on page 13-87
- "RAISE Statement" on page 13-110

label

Label that identifies *`while_loop_statement`* (see "statement ::=" on page 13-13 and "label" on page 13-16). `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 in the `END LOOP` statement matches a label at the beginning of the same `LOOP` statement (the compiler does not check).

Examples

- Example 4–27, "WHILE LOOP Statements" on page 4-20

Related Topics

In this chapter:

- "Basic LOOP Statement" on page 13-7
- "CONTINUE Statement" on page 13-38
- "Cursor FOR LOOP Statement" on page 13-40
- "EXIT Statement" on page 13-55
- "Explicit Cursor Declaration and Definition" on page 13-57
- "FETCH Statement" on page 13-71
- "FOR LOOP Statement" on page 13-74
- "FORALL Statement" on page 13-77
- "OPEN Statement" on page 13-101

In other chapters:

- "WHILE LOOP Statement" on page 4-20

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 at schema level 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 1-10.

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

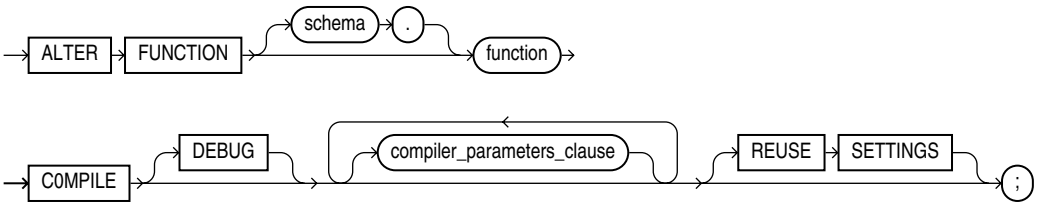
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

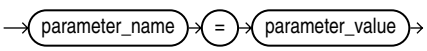
If the function is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the function must be in your schema or you must have `ALTER ANY PROCEDURE` system privilege.

Syntax

***alter_function* ::=**



***compiler_parameters_clause* ::=**



Semantics

schema

Name of the schema containing the function. **Default:** your schema.

function

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 invoke the recompiled function or package bodies that define subprograms that invoke 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`.

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.

compiler_parameters_clause

Specifies a value for a PL/SQL compilation parameter in Table 1–2. 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.

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 14-90

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 14-41 with the `OR REPLACE` clause.

Topics

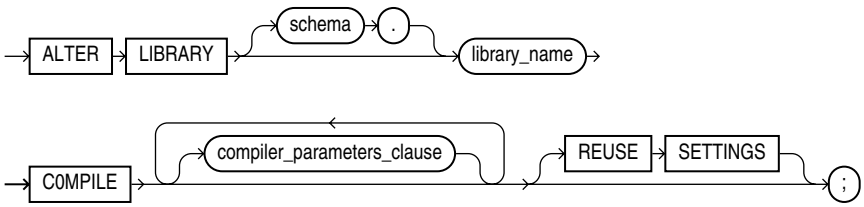
- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

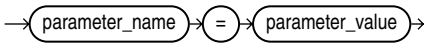
If the library is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the library must be in your schema or you must have the `ALTER ANY LIBRARY` system privilege.

Syntax

***alter_library* ::=**



***compiler_parameters_clause* ::=**



Semantics

library_name

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 `REUSE SETTINGS`.

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`.

REUSE SETTINGS

Prevents Oracle from dropping and reacquiring compiler switch settings. Preserves the existing settings and uses them for the recompilation of any parameters for which values are not specified elsewhere in this statement.

compiler_parameters_clause

Specifies a value for a PL/SQL compilation parameter in Table 1–2. 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`, and if a value was specified for the parameter in an earlier compilation of this PL/SQL unit, then 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.

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 14-41
- "DROP LIBRARY Statement" on page 14-92

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. 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 14-43, or the "CREATE PACKAGE BODY Statement" on page 14-46 with the `OR REPLACE` clause.

Topics

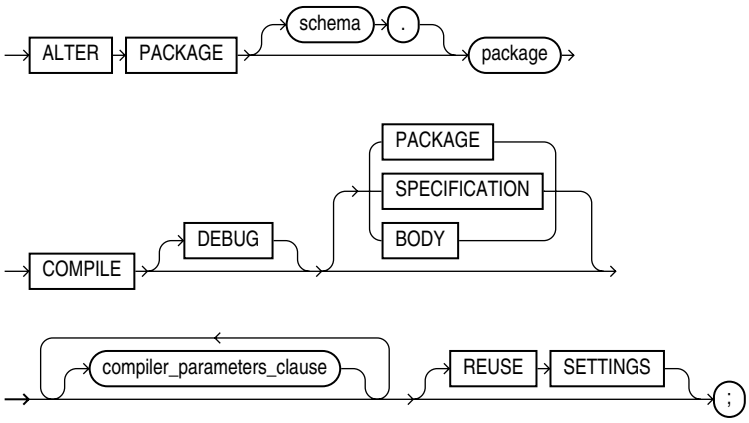
- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

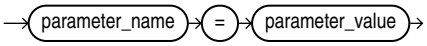
If the package is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the package must be in your schema or you must have `ALTER ANY PROCEDURE` system privilege.

Syntax

***alter_package* ::=**



***compiler_parameters_clause* ::=**



Semantics

schema

Name of the schema containing the package. **Default:** your schema.

package

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 `REUSE SETTINGS`.

> **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 invoke 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

(Default) Recompiles both the package specification and (if it exists) the package body, whether they are valid or invalid. The recompilation of the package specification and body lead to the invalidation and recompilation of dependent objects as described for `SPECIFICATION` and `BODY`.

REUSE SETTINGS

Has the same behavior for a package as it does for a function. See "REUSE SETTINGS" on page 14-4.

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.

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 14-44 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 invoke 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 invoke 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 14-43
- "DROP PACKAGE Statement" on page 14-93

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 14-50 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

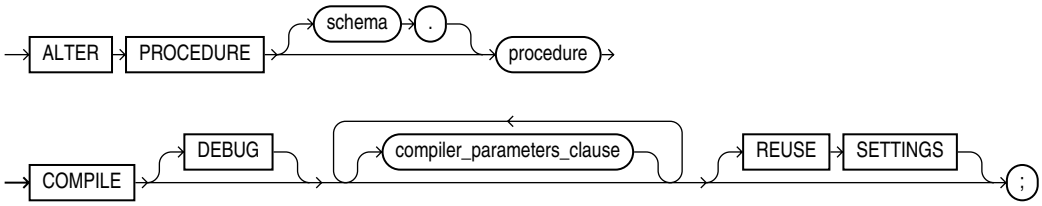
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

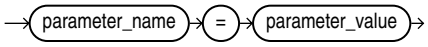
If the procedure is in the SYS schema, you must be connected as SYSDBA. Otherwise, the procedure must be in your schema or you must have ALTER ANY PROCEDURE system privilege.

Syntax

***alter_procedure* ::=**



***compiler_parameters_clause* ::=**



Semantics

schema

Name of the schema containing the procedure. **Default:** your schema.

procedure

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 invoke the recompiled procedure or package bodies that define subprograms that invoke 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 `REUSE SETTINGS`.

> **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

REUSE SETTINGS

Has the same behavior for a procedure as it does for a function. See "REUSE SETTINGS" on page 14-4.

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.

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 invoke `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 14-50

- "DROP PROCEDURE Statement" on page 14-95

ALTER TRIGGER Statement

The `ALTER TRIGGER` statement enables, disables, compiles, or renames 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 14-54 with the `OR REPLACE` clause.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

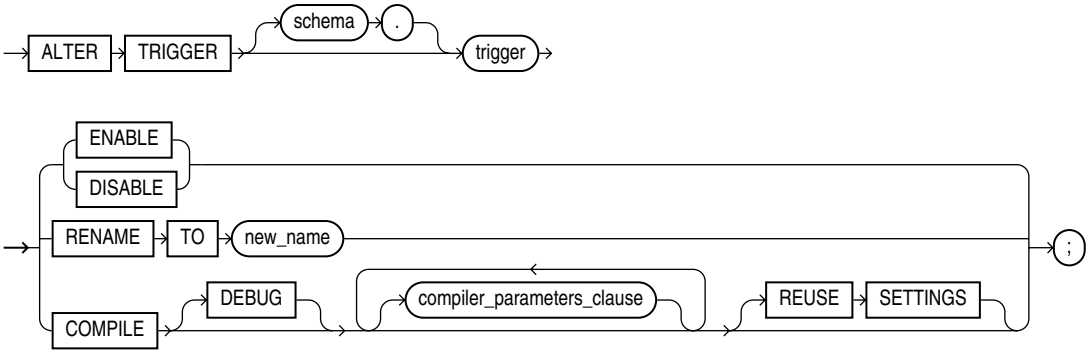
If the trigger is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the trigger must be in your schema or you must have `ALTER ANY TRIGGER` system privilege.

In addition, to alter a trigger on `DATABASE`, you must have the `ADMINISTER DATABASE TRIGGER` system privilege.

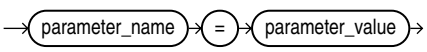
> **See Also:** "CREATE TRIGGER Statement" on page 14-54 for more information about triggers based on `DATABASE` triggers

Syntax

***alter_trigger* ::=**



***compiler_parameters_clause* ::=**



Semantics

schema

Name of the schema containing the trigger. **Default:** your schema.

trigger

Name of the trigger to be altered.

ENABLE

Enables the trigger.

DISABLE

Disables the trigger.

See Also:

- "Enabling Triggers: Example" on page 14-16
- "Disabling Triggers: Example" on page 14-16

RENAME TO *new_name*

Renames the trigger without changing its state.

When you rename a trigger, the database rebuilds the remembered source of the trigger in the `USER_SOURCE`, `ALL_SOURCE`, and `DBA_SOURCE` static 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 `REUSE SETTINGS`.

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

REUSE SETTINGS

Has the same behavior for a trigger as it does for a function. See REUSE SETTINGS on page 14-4.

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.

Examples

Disabling Triggers: Example The sample schema `hr` has a trigger named `update_job_history` created on the `employees` table. The trigger fires 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

In this chapter:

- "CREATE TRIGGER Statement" on page 14-54
- "DROP TRIGGER Statement" on page 14-97

In other chapters:

- "Trigger Compilation, Invalidation, and Recompilation" on page 9-32
- "Trigger Enabling and Disabling" on page 9-41

ALTER TYPE Statement

The `ALTER TYPE` statement does one of the following to a type that was created with "CREATE TYPE Statement" on page 14-68 and "CREATE TYPE BODY Statement" on page 14-85:

- **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

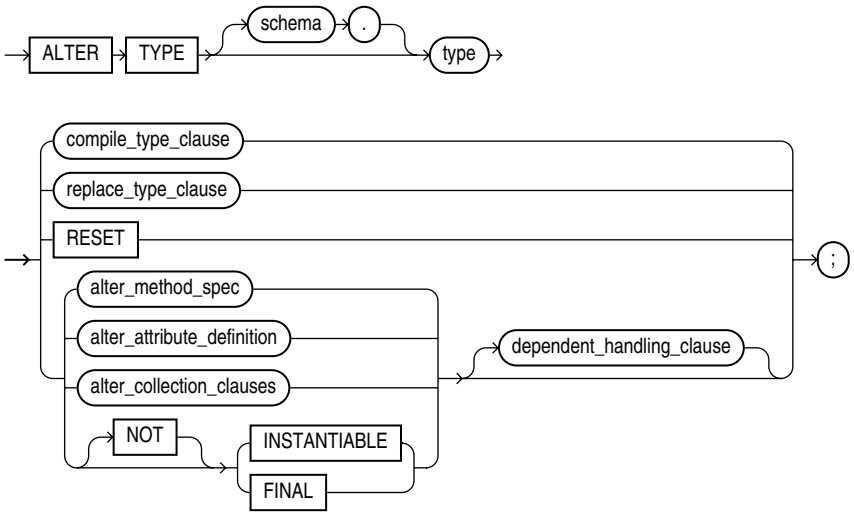
- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

If the type is in the `SYS` schema, you must be connected as `SYSDBA`. Otherwise, the type must be in your schema and you must have `CREATE TYPE` or `CREATE ANY TYPE` system privilege, or you must have `ALTER ANY TYPE` system privileges.

Syntax

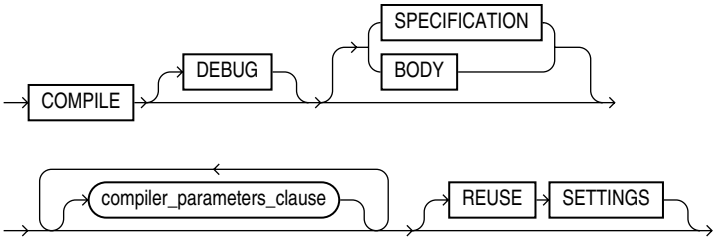
***alter_type* ::=**



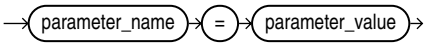
See:

- "alter_attribute_definition ::=" on page 14-20
- "alter_method_spec ::=" on page 14-20
- "alter_collections_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

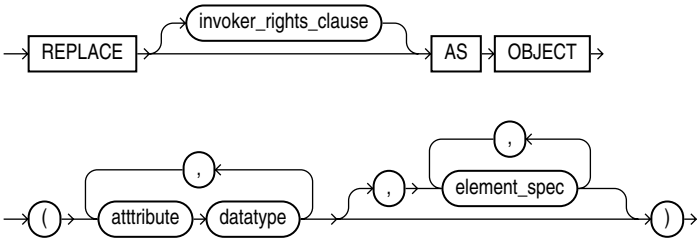
***compile_type_clause* ::=**



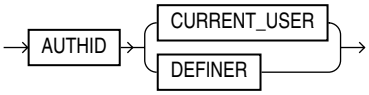
***compiler_parameters_clause* ::=**



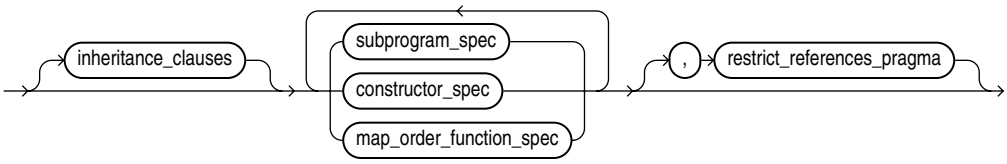
***replace_type_clause* ::=**



***invoker_rights_clause* ::=**



***element_spec* ::=**

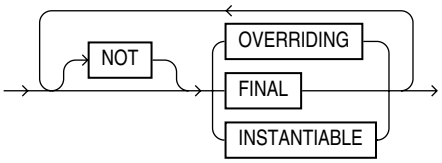


See:

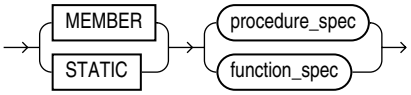
- "constructor_spec ::=" on page 14-19
- "inheritance_clauses ::=" on page 14-19
- "map_order_function_spec ::=" on page 14-19
- "pragma_clause ::=" on page 14-20

- "subprogram_spec ::=" on page 14-19

***inheritance_clauses* ::=**



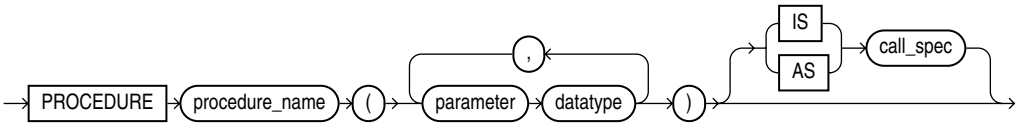
***subprogram_spec* ::=**



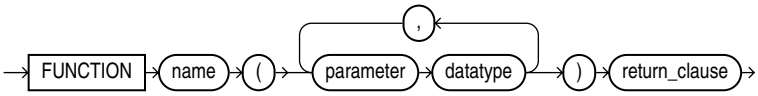
See:

- "function_spec ::=" on page 14-19
- "procedure_spec ::=" on page 14-19

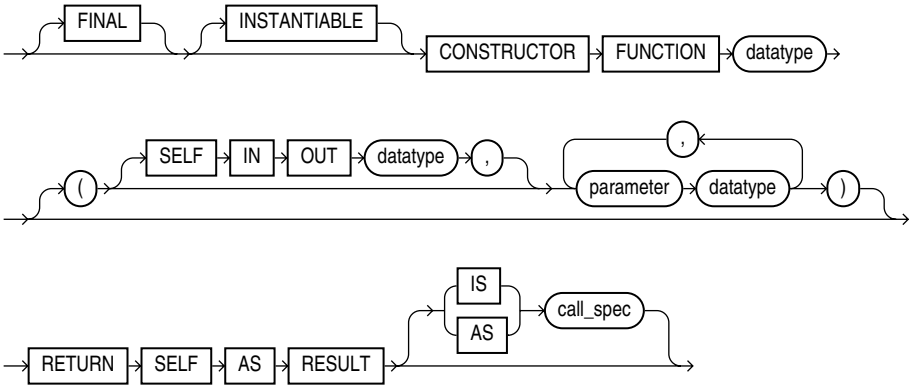
***procedure_spec* ::=**



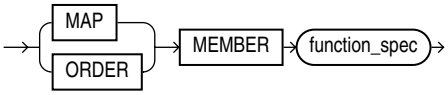
***function_spec* ::=**



***constructor_spec* ::=**

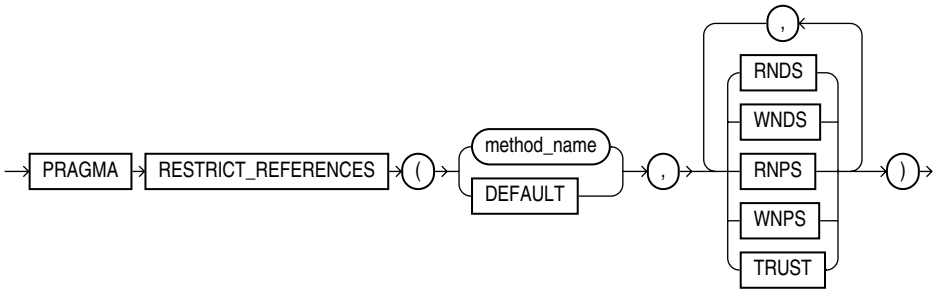


***map_order_function_spec* ::=**

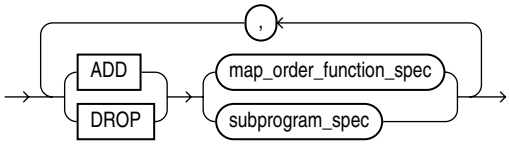


See "function_spec ::=" on page 14-19.

***pragma_clause* ::=**



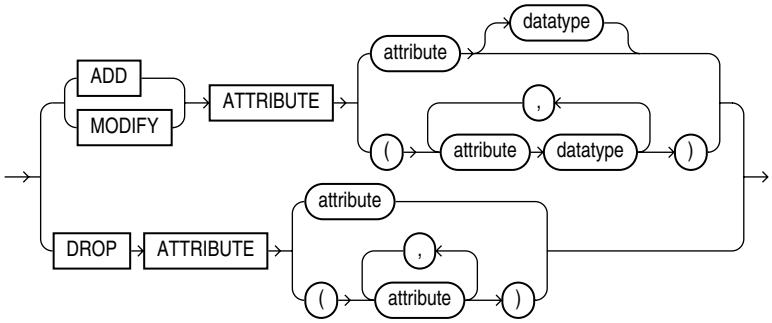
***alter_method_spec* ::=**



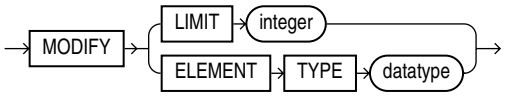
See:

- "map_order_function_spec ::=" on page 14-19
- "subprogram_spec ::=" on page 14-19

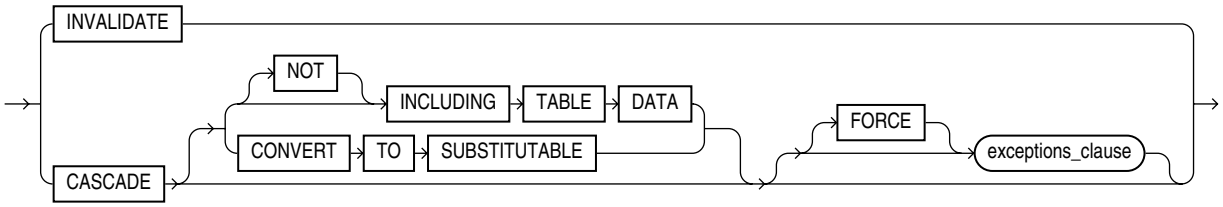
***alter_attribute_definition* ::=**



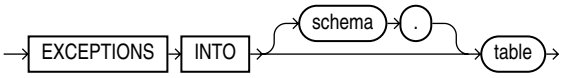
***alter_collections_clauses*::=**



***dependent_handling_clause* ::=**



***exceptions_clause* ::=**



Semantics

schema

Name of the schema containing the type. **Default:** your schema.

type

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*.

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 14-29.

Restriction on RESET You cannot specify `RESET` if the type has any table dependents (direct or indirect).

[NOT] INSTANTIABLE

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.

[NOT] FINAL

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 from FINAL to NOT FINAL, or the reverse, then you must specify the CASCADE clause of the "dependent_handling_clause" on page 14-27 to convert data in dependent columns and tables. Specifically:

- 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 if you want 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 if 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.

    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.

compile_type_clause

(Default) Recompiles the type specification and 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 REUSE SETTINGS.

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.

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 8-43.

Restriction on *invoker_rights_clause* You can specify this clause only for an ADT, not for a nested table or varray.

attribute

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

Specifies elements of the redefined object.

inheritance_clauses

Specifies the relationship between supertypes and subtypes.

subprogram_spec

Specifies a subprogram to be referenced as an ADT attribute. For each such subprogram, you must specify a corresponding method body in the ADT body.

See Also:

- "CREATE TYPE Statement" on page 14-68 for a description of the difference between member and static methods, and for examples
- "CREATE TYPE BODY Statement" on page 14-85
- "Overloaded Subprograms" on page 8-25 for information about overloading subprogram names in a package

procedure_spec

Specification of a procedure subprogram.

function_spec

Specification of a function subprogram.

restrict_references_pragma

Deprecated clause, described in "RESTRICT_REFERENCES Pragma" on page 13-114.

Restriction on *restrict_references_pragma* This 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 of 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. You must not specify a comparison method to determine the equality of two ADTs.

MAP

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`* is to 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.

ORDER

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.

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

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.

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 14-22.

INCLUDING TABLE DATA

(Default) Converts data stored in all user-defined columns to the most recent version of the column type.

> **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 14-22 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 14-79 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 14-79 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 14-79 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 14-68
- "CREATE TYPE BODY Statement" on page 14-85
- "DROP TYPE Statement" on page 14-98

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 Declaration and Definition" on page 13-83.

A **call specification** declares a Java method or a third-generation language (3GL) subprogram so that it can be invoked from PL/SQL. You can also use the SQL `CALL` statement to invoke 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 an invocation 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 8-31.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

Prerequisites

To create or replace a standalone stored function in your 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.

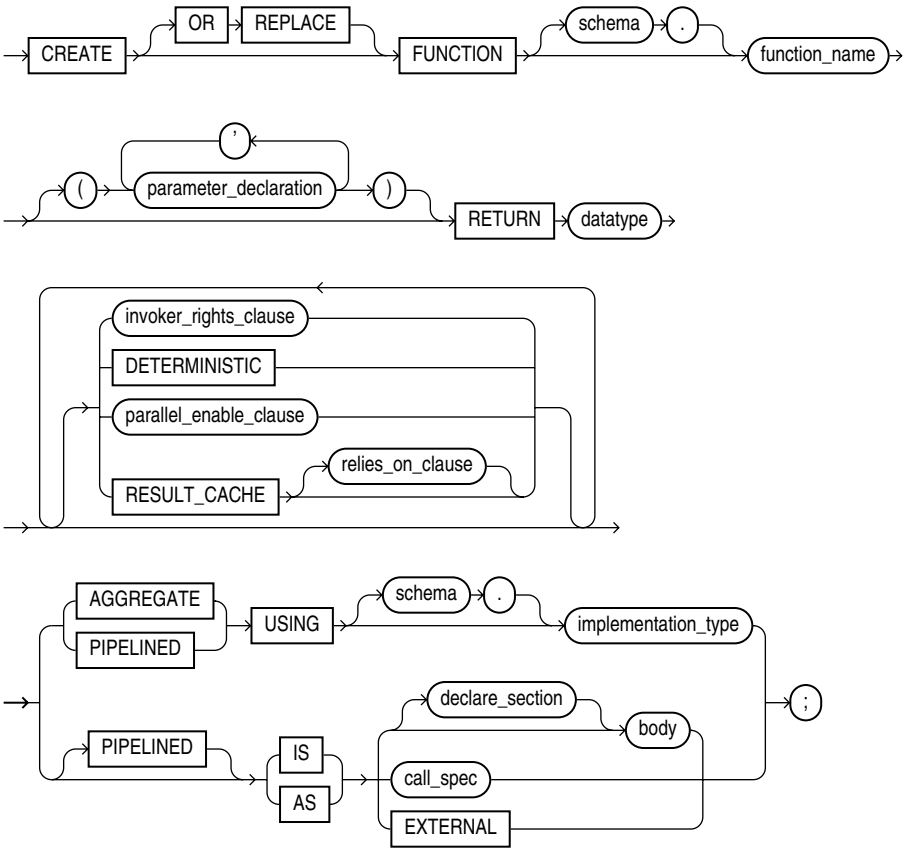
To invoke a call specification, you may need additional privileges, for example, `EXECUTE` privileges on a C library for a C call specification.

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

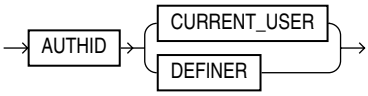
create_function ::=



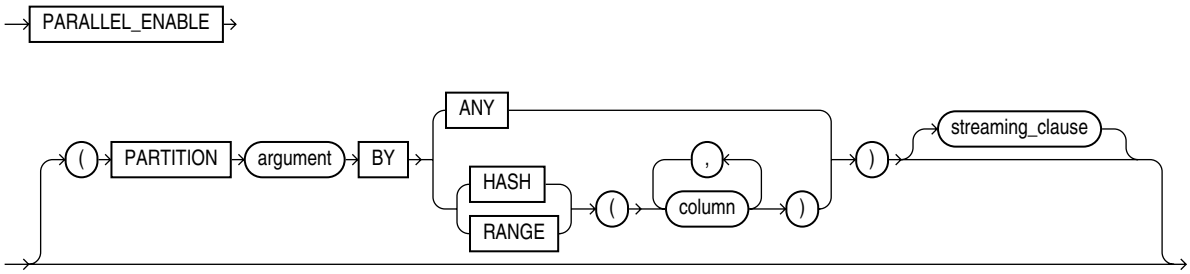
See:

- "body ::=" on page 13-12
- "call_spec ::=" on page 14-34
- "datatype ::=" on page 13-26
- "declare_section ::=" on page 13-9
- "parameter_declaration ::=" on page 13-80
- "relies_on_clause ::=" on page 13-84

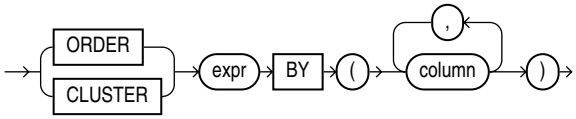
invoker_rights_clause ::=



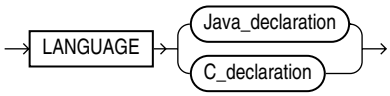
***parallel_enable_clause* ::=**



***streaming_clause* ::=**

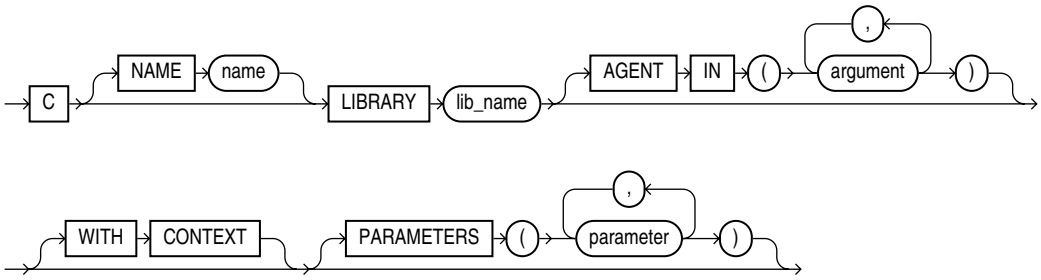


***call_spec* ::=**



***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

Name of the schema containing the function. **Default:** your schema.

function_name

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 invoking functions with BOOLEAN parameters or returns. Therefore, for SQL statements to invoke 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 8-43.

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 invoke 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 invoke 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* for information about using parallel table functions

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:

  - "Chaining Pipelined Table Functions for Multiple Transformations" on page 12-38
  - *Oracle Database Data Cartridge Developer's Guide* for information about using pipelined table functions

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 SQL 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 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 C procedure or Java 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* to learn how to write Java call specifications
- *Oracle Database Advanced Application Developer's Guide* to learn how to write C call specifications

EXTERNAL

Deprecated way of declaring a C function, supported only for backward compatibility. Oracle recommends that you use the LANGUAGE C syntax.

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 invoke 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
--------- -------------
      9450           100
  13670.74            50
     14175            80
   18742.5            90
```

Package 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 14-50
- "DROP FUNCTION Statement" on page 14-90

In other chapters:

- "Function Declaration and Definition" on page 13-83 for information about creating a function in a PL/SQL block
- "Formal Parameter Declaration" on page 13-80
- 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
- *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 invoke third-generation-language (3GL) functions and procedures.

Topics

- Prerequisites
- Syntax
- Semantics
- Examples
- Related Topics

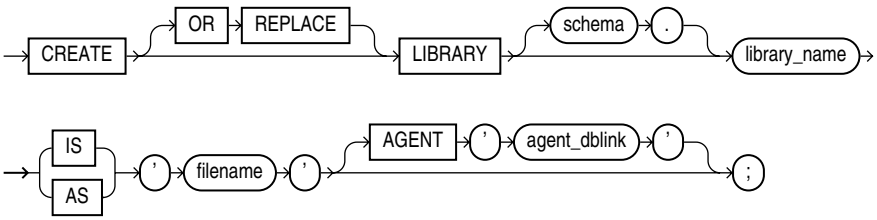
Prerequisites

To create a library in your 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

Name of the schema containing the library. **Default:** your schema.

library_name

Name that represents 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 run from a database link other than the server. Oracle Database uses the database link that *agent_dblink* specifies to run external procedures. If you omit this clause, then the default agent on the server (extproc) runs 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 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 14-92

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

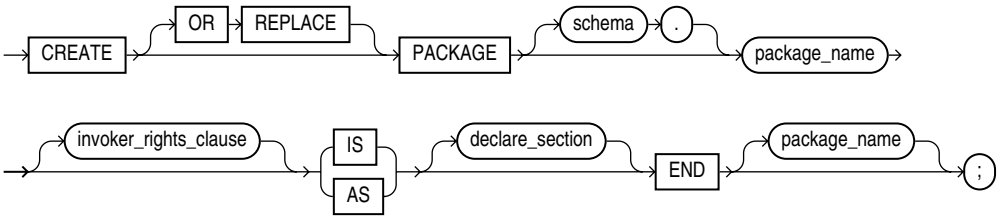
Prerequisites

To create or replace a package in your 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

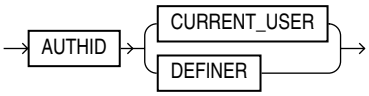
***create_package* ::=**



See:

- "declare_section ::=" on page 13-9
- "invoker_rights_clause ::=" on page 14-43

***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

Name of the schema containing the package. **Default:** your schema.

package_name

A package stored in the database. For naming conventions, see "Identifiers" on page 2-4.

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 8-43.

declare_section

Has a definition for every cursor and subprogram declaration in the package specification. The headings of corresponding subprogram declarations and definitions must match word for word, except for white space.

Can also declare and define private items that can be referenced only from inside the package.

Restriction on *declare_section* The AUTONOMOUS_TRANSACTION pragma cannot appear here.

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 invoke any of these public procedures or functions or raise any of the public exceptions of the package.

Before you can invoke 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 14-46.

Related Topics

In this chapter:

- "ALTER PACKAGE Statement" on page 14-8
- "CREATE PACKAGE Statement" on page 14-43
- "CREATE PACKAGE BODY Statement" on page 14-46
- "DROP PACKAGE Statement" on page 14-93

In other chapters:

- Chapter 10, "PL/SQL Packages"
- "Package Specification" on page 10-3
- "Function Declaration and Definition" on page 13-83
- "Procedure Declaration and Definition" on page 13-108

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

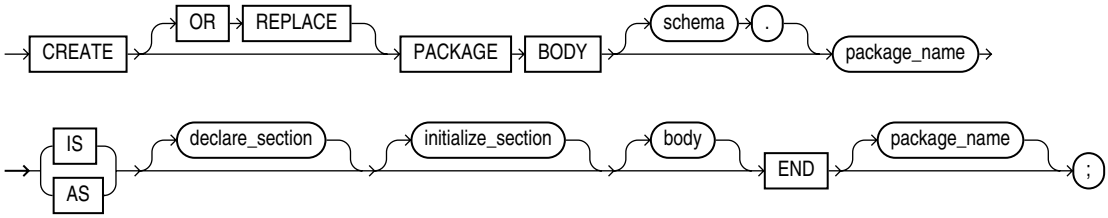
Prerequisites

To create or replace a package in your 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.

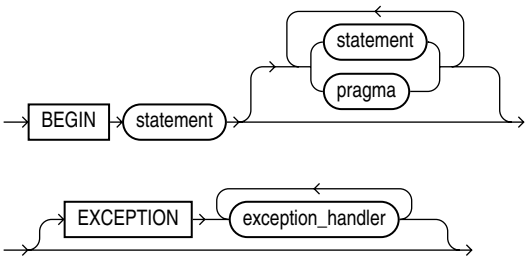
Syntax

***create_package_body* ::=**



See "declare_section ::=" on page 13-9.

***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

Name of the schema containing the package. **Default:** your schema.

package_name

Name of the package to be created.

declare_section

Has a definition for every cursor and subprogram declaration in the package specification. The headings of corresponding subprogram declarations and definitions must match word for word, except for white space.

Can also declare and define private items that can be referenced only from inside the package.

Restriction on *declare_section* The `AUTONOMOUS_TRANSACTION` pragma cannot appear here.

initialize_section

Initializes variables and does any other one-time setup steps.

Examples

Creating a Package Body: Example This statement creates the body of the `emp_mgmt` package created in "Creating a Package: Example" on page 14-44.

```
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 invokes 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 14-43

In other chapters:

- Chapter 10, "PL/SQL Packages"
- "Package Body" on page 10-5
- "Function Declaration and Definition" on page 13-83
- "Procedure Declaration and Definition" on page 13-108

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 Declaration and Definition" on page 13-108.

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 invoke 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 an invocation 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 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.

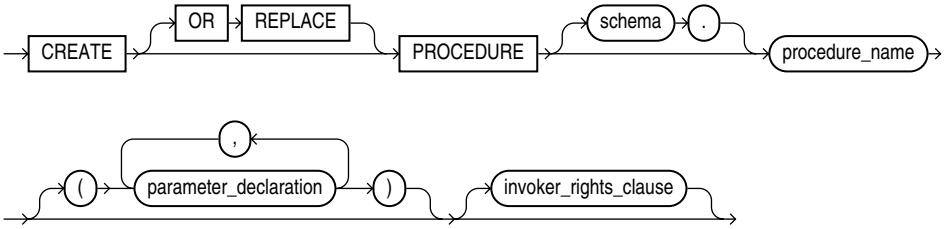
To invoke a call specification, you may need additional privileges, for example, the EXECUTE object privilege on the C library for a C call specification.

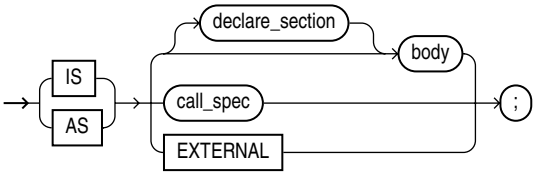
To embed a CREATE PROCEDURE statement inside an Oracle precompiler program, you must terminate the statement with the keyword END-EXEC followed by the embedded SQL statement terminator for the specific language.

> **See Also:** For more information about such prerequisites:
>
> - *Oracle Database Advanced Application Developer's Guide*
> - *Oracle Database Java Developer's Guide*

Syntax

create_procedure ::=

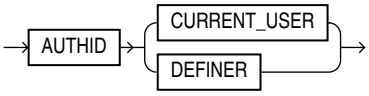




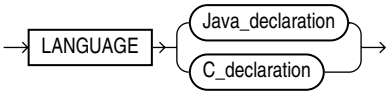
See:

- "body ::=" on page 13-12
- "declare_section ::=" on page 13-9
- "parameter_declaration ::=" on page 13-80

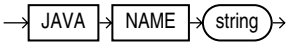
invoker_rights_clause ::=



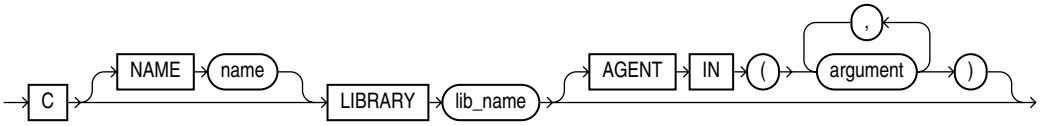
call_spec ::=

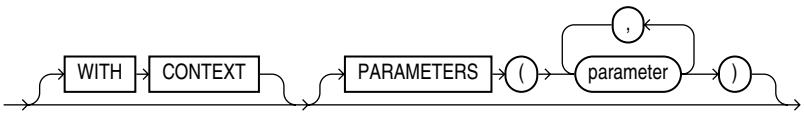


Java_declaration ::=



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

Name of the schema containing the procedure. **Default:** your schema.

procedure_name

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 8-43.

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 C procedure or Java 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* to learn how to write Java call specifications
- *Oracle Database Advanced Application Developer's Guide* to learn how to write C call specifications

EXTERNAL

Deprecated way of declaring a C procedure, supported only for backward compatibility. Oracle recommends that you use the `LANGUAGE C` syntax.

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 invoke 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 14-47 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 14-95

In other chapters:

- "Formal Parameter Declaration" on page 13-80
- "Procedure Declaration and Definition" on page 13-108
- 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 an invocation of a procedure implemented in PL/SQL or Java

The database automatically runs a trigger when specified conditions occur.

Topics

- Prerequisites
- Syntax
- Semantics
- Related Topics

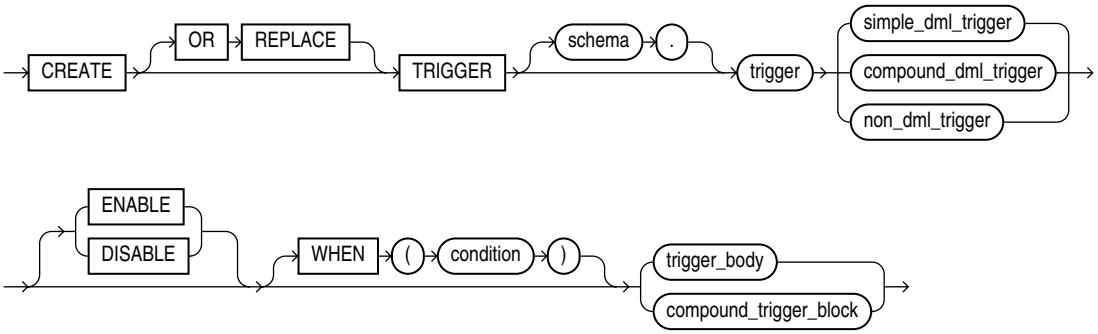
Prerequisites

- To create a trigger in your schema on a table in your schema or on your 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 invokes 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

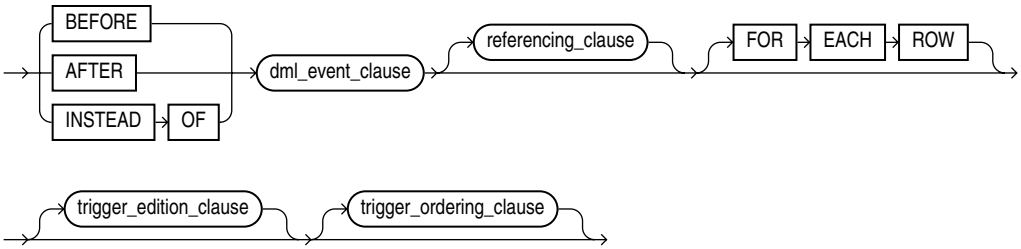
***create_trigger* ::=**



See:

- "compound_dml_trigger ::=" on page 14-55
- "compound_trigger_block ::=" on page 14-56
- "non_dml_trigger ::=" on page 14-55
- "trigger_body ::=" on page 14-55

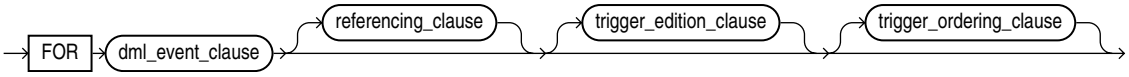
***simple_dml_trigger* ::=**



See:

- "dml_event_clause ::=" on page 14-56
- "referencing_clause ::=" on page 14-56
- "trigger_edition_clause ::=" on page 14-56
- "trigger_ordering_clause ::=" on page 14-56

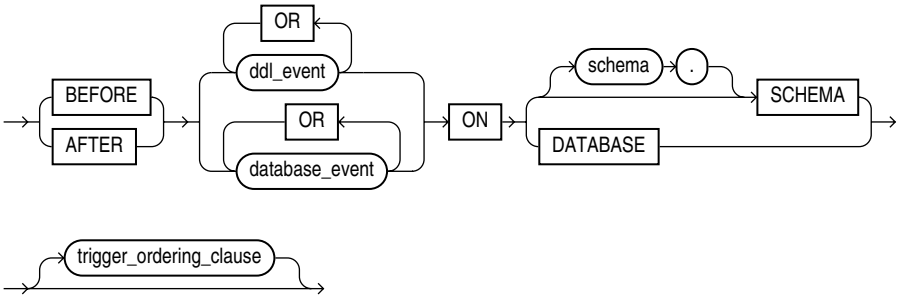
***compound_dml_trigger* ::=**



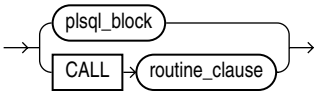
See:

- "dml_event_clause ::=" on page 14-56
- "referencing_clause ::=" on page 14-56
- "trigger_edition_clause ::=" on page 14-56
- "trigger_ordering_clause ::=" on page 14-56

***non_dml_trigger* ::=**



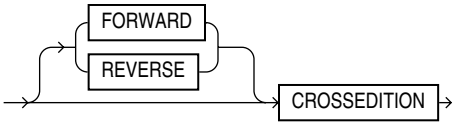
***trigger_body* ::=**



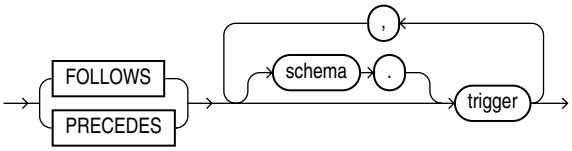
See:

- "plsql_block ::=" on page 13-9
- `routine_clause` in *Oracle Database SQL Language Reference*

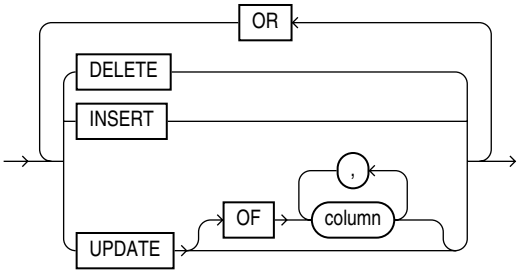
***trigger_edition_clause* ::=**

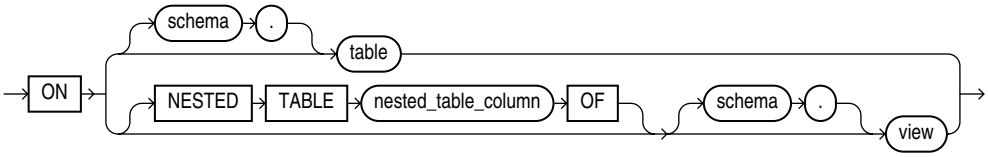


***trigger_ordering_clause* ::=**

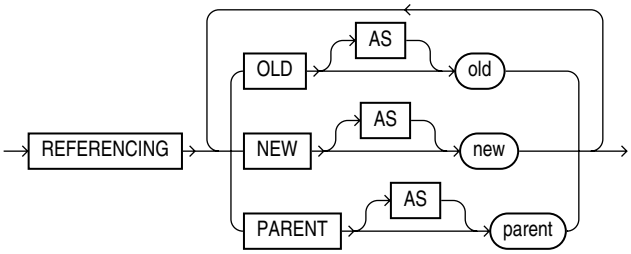


***dml_event_clause* ::=**

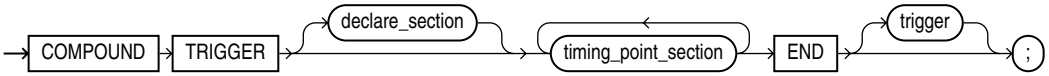




***referencing_clause* ::=**

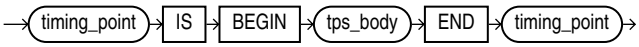


***compound_trigger_block* ::=**

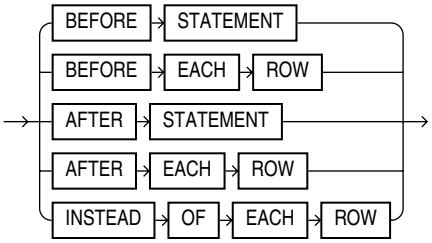


See "declare_section ::=" on page 13-9.

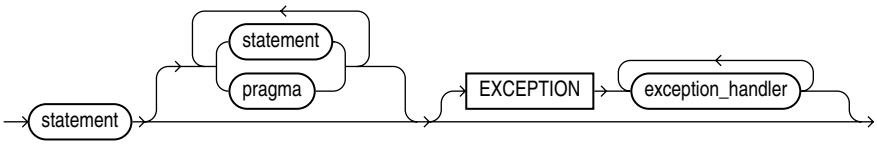
***timing_point_section* ::=**



***timing_point* ::=**



***tps_body* ::=**



See:

- "exception_handler ::=" on page 13-50
- "pragma ::=" on page 13-11
- "statement ::=" on page 13-13

Semantics

OR REPLACE

Re-creates the trigger if it exists, and recompiles it.

Users who were granted privileges on the trigger before it was redefined can still access the procedure without being regranted the privileges.

schema

Name of the schema containing the trigger. **Default:** your schema.

trigger

Name of the trigger to be created.

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.

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 database definition language (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

(Default) Creates the trigger as a forward crossedition trigger, which is the type of trigger described in **CROSSEDITION**.

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. This trigger propagates 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 a simple DML trigger (described in "DML Triggers" on page 9-3). A *`simple_dml_trigger`* must have a *`trigger_body`*, not a *`compound_trigger_block`*.

BEFORE

Causes the database to fire the trigger before running the triggering event. For row triggers, the trigger fires 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 fires 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

Creates an `INSTEAD OF` trigger (described in "INSTEAD OF Triggers" on page 9-10).

Restrictions on INSTEAD OF

- You can create an `INSTEAD OF` trigger only on a noneditioning view (not an editioning view or table), or on a nested table column of a noneditioning view (see "NESTED TABLE nested_table_column" on page 14-61).
- An `INSTEAD OF` trigger cannot have a `WHEN` clause.
- An `INSTEAD OF` trigger cannot have a column list.
- An `INSTEAD OF` trigger can read the `OLD` and `NEW` values, but cannot change them.

Note:

- If the view is inherently updatable and has `INSTEAD OF` triggers, the triggers take precedence: The database fires the triggers instead of performing DML on the view.
- If the view belongs to a hierarchy, then the subviews do not inherit the trigger.
- The `WITH 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. For information about `WITH CHECK OPTION`, see *Oracle Database SQL Language Reference*.
- 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.

dml_event_clause

Specifies the triggering statements for a DML trigger. The database fires the trigger in the existing user transaction.

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 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. For more information, see "INSTEAD OF Triggers on Nested Table Columns of Views" on page 9-12.

Restriction on NESTED TABLE You can specify `NESTED TABLE` only for `INSTEAD OF` triggers.

referencing_clause

Specifies correlation names, which refer to old, new, and parent values of the current row. **Defaults:** `OLD`, `NEW`, and `PARENT`.

If your trigger is associated with a table named `OLD`, `NEW`, or `PARENT`, then use this clause to specify different correlation names to avoid confusion between the table names and the correlation names.

If the trigger is defined on a nested table, then `OLD` and `NEW` refer to the current row of the nested table, and `PARENT` refers to the current row of the parent table. If the trigger is defined on a database table or view, then `OLD` and `NEW` refer to the current row of the database table or view, and `PARENT` is undefined.

You can use correlation names in any trigger body and in the `WHEN` condition of a row-level simple DML trigger or a compound DML trigger.

Restrictions on *referencing_clause* The *`referencing_clause`* is not valid with:

- An `INSTEAD OF` trigger on a `CREATE` DDL event
- A DML trigger whose body is `CALL` *`routine`*

FOR EACH ROW

Creates the trigger as a row trigger. The database fires a row trigger 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 when the triggering statement is issued if the optional trigger constraint is met.

`INSTEAD OF` trigger statements are implicitly activated for each row.

compound_dml_trigger

Creates a compound DML trigger (described in "Compound DML Triggers" on page 9-14). A *`compound_dml_trigger`* must have a *`compound_trigger_block`*, not a *`trigger_body`*.

non_dml_trigger

Defines a system trigger (described in "System Triggers" on page 9-30). A *`non_dml_trigger`* must have a *`trigger_body`*, not a *`compound_trigger_block`*.

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.

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 an `AUDIT` statement is issued.

- `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 a NOAUDIT statement is issued.

- 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:** "Triggers for Publishing Events" on page 9-43 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 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-00018: maximum number of sessions exceeded
  - ORA-00020: maximum number of processes (string) exceeded
  - ORA-01034: ORACLE not available
  - 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-04030: out of process memory when trying to allocate *string* bytes (*string, string*)

- `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 SUSPEND`

  Causes the database to fire the trigger whenever a server error causes a transaction to be suspended.

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.

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

(Default) Creates the trigger in an enabled state.

DISABLE

Creates the trigger in a disabled state, which lets you ensure that the trigger compiles without errors before you enable it.

Note: `DISABLE` is especially useful if you are creating a crossedition trigger, which affects the online application being redefined if compilation errors occur.

WHEN (*condition*)

Specifies a SQL condition that the database evaluates for each row that the triggering statement affects. If the value of *`condition`* is `TRUE` for an affected row, then

trigger_body or *tps_body* runs for that row; otherwise, *trigger_body* or *tps_body* does not run for that row. The triggering statement runs regardless of the value of *condition*.

In a DML trigger, the *condition* can contain correlation names (see "referencing_clause ::=" on page 14-56). In *condition*, do not put a colon (:) before the correlation name NEW, OLD, or PARENT (in this context, it is not a placeholder for a bind argument).

> **See Also:** *Oracle Database SQL Language Reference* for information about SQL conditions

Restrictions on WHEN (*condition*)

- You cannot specify this clause for a STARTUP, SHUTDOWN, or DB_ROLE_CHANGE trigger.
- If you specify this clause for a SERVERERROR trigger, then *condition* must be ERRNO = *error_code*.
- If you specify this clause for a simple DML trigger, then you must also specify FOR EACH ROW and you cannot specify INSTEAD OF.
- The *condition* cannot include a subquery or a PL/SQL expression (for example, an invocation of a user-defined function).

trigger_body

The PL/SQL block or CALL subprogram that the database runs to fire either a *simple_dml_trigger* or *non_dml_trigger*. A CALL subprogram is either a PL/SQL subprogram or a Java subprogram in a PL/SQL wrapper.

If *trigger_body* is a PL/SQL block and it contains errors, then the CREATE [OR REPLACE] statement fails.

Restrictions on *trigger_body*

- It cannot appear in a compound DML trigger.
- Its *declare_section* cannot declare variables of the data type LONG or LONG RAW.

In *trigger_body* or *tps_body*, *declare_section* cannot declare variables of the data type LONG or LONG RAW.

compound_trigger_block

Can appear only in a compound DML trigger.

If the trigger is created on a noneditioning view, then *compound_trigger_block* must have only one timing point section, whose *timing_point* must be INSTEAD OF EACH ROW.

If the trigger is created on a table or editioning view, then timing point sections can be in any order, but no timing point section can be repeated.

Restriction on *compound_trigger_block* The *declare_section* of *compound_trigger_block* cannot include PRAGMA AUTONOMOUS_TRANSACTION.

> **See Also:** "Compound DML Trigger Restrictions" on page 9-16

Related Topics

In this chapter:

- "ALTER TRIGGER Statement" on page 14-14
- "DROP TRIGGER Statement" on page 14-97

In other chapters:

- Chapter 9, "PL/SQL Triggers"

> **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)
- Standalone stored varying array (varray) type
- Standalone stored 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, allowing you 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:

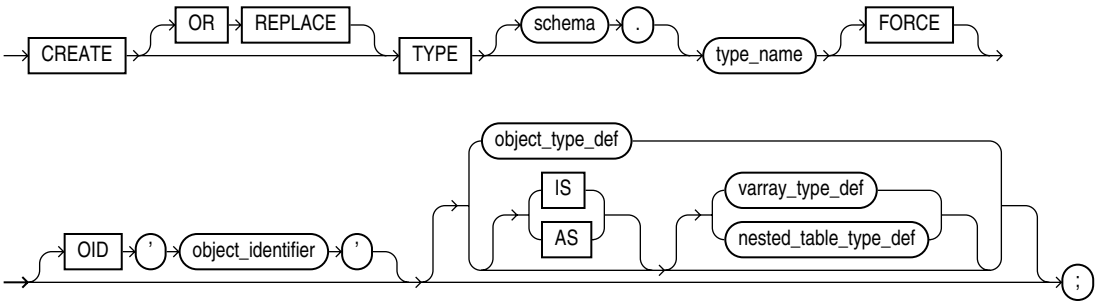
- If you create a type whose specification declares only attributes but no methods, then you need not specify a type body.
- If you create a SQLJ object type, then you cannot specify a type body. The implementation of the type is specified as a Java class.
- A standalone stored collection type that you create with the `CREATE TYPE` statement differs from a collection type that you define with the keyword `TYPE` in a PL/SQL block or package. For information about the latter, see "Collection Variable Declaration" on page 13-25.
- 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

Syntax

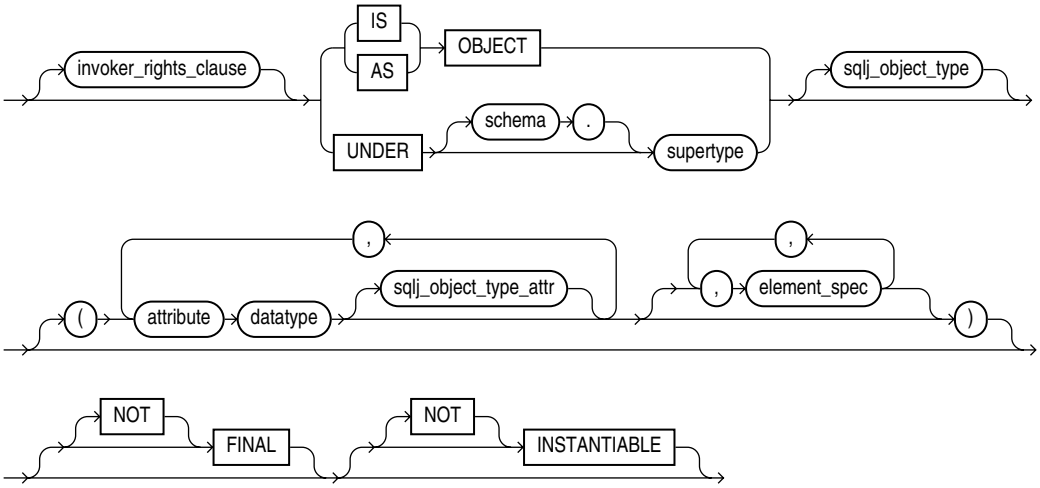
create_type ::=



See:

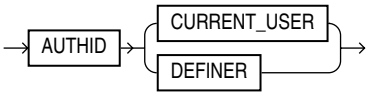
- "nested_table_type_def ::=" on page 13-26
- "varray_type_def ::=" on page 13-26

object_type_def ::=

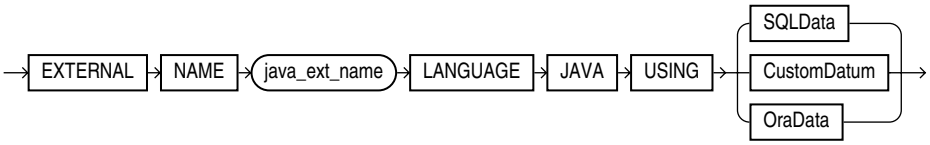


See "element_spec ::=" on page 14-70.

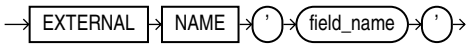
invoker_rights_clause ::=



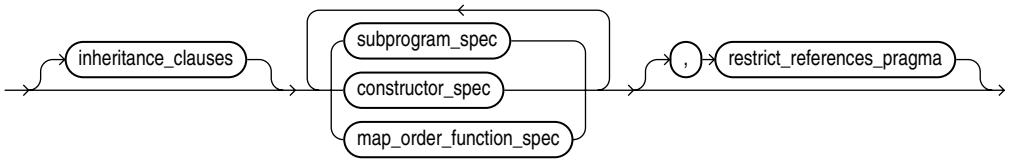
sqlj_object_type ::=



sqlj_object_type_attr ::=



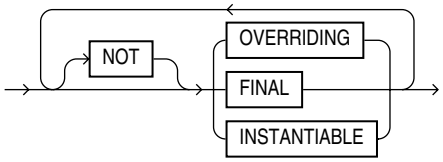
element_spec ::=



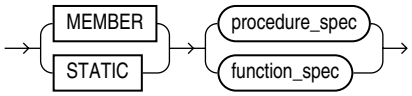
See:

- "constructor_spec ::=" on page 14-71
- "map_order_function_spec ::=" on page 14-71
- "restrict_references_pragma ::=" on page 13-114

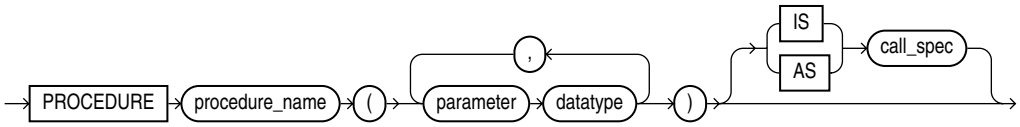
inheritance_clauses ::=



subprogram_spec ::=

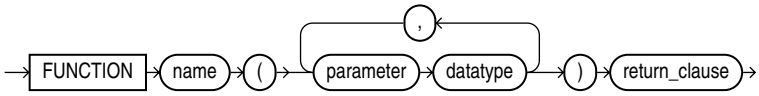


procedure_spec ::=



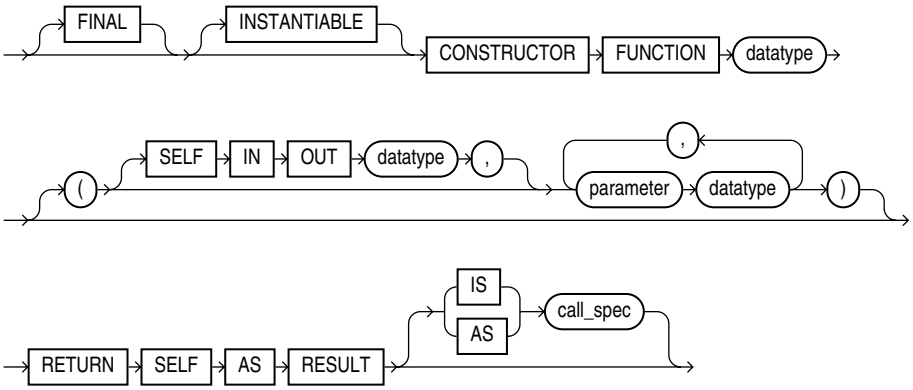
See "call_spec ::=" on page 14-71.

function_spec ::=



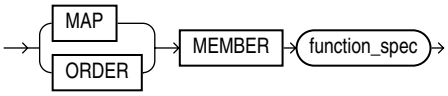
See "return_clause ::=" on page 14-71.

***constructor_spec* ::=**



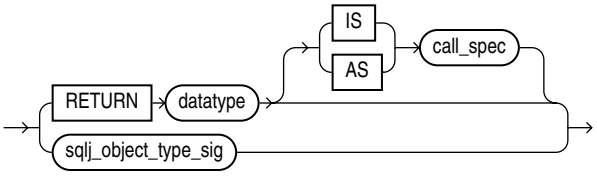
See "call_spec ::=" on page 14-34.

***map_order_function_spec* ::=**



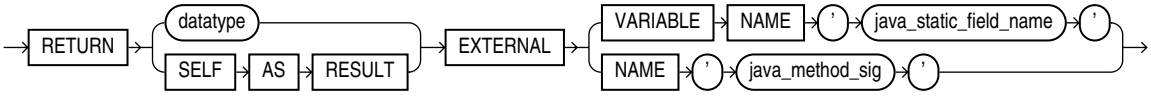
See "function_spec ::=" on page 14-70.

***return_clause* ::=**

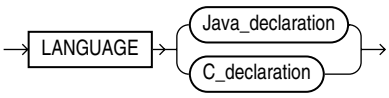


See "call_spec ::=" on page 14-34.

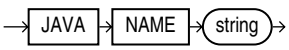
***sqlj_object_type_sig* ::=**



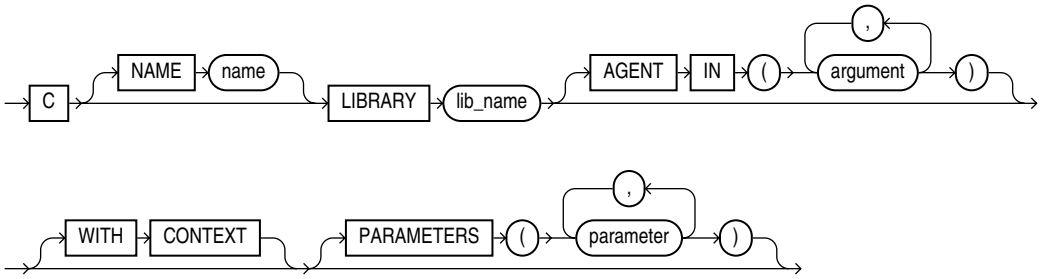
***call_spec* ::=**



***Java_declaration* ::=**



***C_declaration* ::=**



Prerequisites

To create a type in your 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.

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

Name of the schema containing the type. **Default:** your schema.

type_name

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 14-79

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 8-43.

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 14-80 and "Type Hierarchy Example" on page 14-81

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

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* for information about these three interfaces and "SQLJ Object Type Example" on page 14-80

element_spec

Specifies each attribute of the ADT.

attribute

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 data type of an ADT attribute. This data type must be stored in the database; that is, either a predefined data type or a user-defined standalone stored collection type. For information about predefined data types, see Chapter 3, "PL/SQL Data Types." For information about user-defined standalone stored collection types, see "Collection Types" on page 5-2.

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`.

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 14-82

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 14-83

[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:

- **(Default)** Specify `FINAL` if no further subtypes can be created for this type.
- 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:

- **(Default)** Specify INSTANTIABLE if object instances of this type can be constructed.
- 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. **Default:** NOT OVERRIDING.

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. **Default:** NOT FINAL.

NOT INSTANTIABLE

Specifies that the type does not provide an implementation for this method. **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 14-77

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 Invocation" on page 13-31 for information about method invocation and methods
- "CREATE PROCEDURE Statement" on page 14-50 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

Maps a C procedure or Java method name, parameter types, and return type to their SQL counterparts. In *Java_declaration*, *string* identifies the Java implementation of the method.

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.

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

EXTERNAL

Deprecated way of declaring a C method, supported only for backward compatibility. Oracle recommends that you use the LANGUAGE C syntax.

restrict_references_pragma

Deprecated clause, described in "RESTRICT_REFERENCES Pragma" on page 13-114.

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 14-82

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`* is to 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 column of such a VARRAY type.

> **See Also:** "Varray Type Example" on page 14-81

nested_table_type_def

Creates a named nested table of type *datatype*.

> **See Also:**
>
> - "Nested Table Type Example" on page 14-82
> - "Nested Table Type Containing a Varray" on page 14-82

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 14-81.

```
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 14-85
- "DROP TYPE Statement" on page 14-98

In other chapters:

- "Abstract Data Types" on page 1-8
- "Conditional Compilation Directive Restrictions" on page 2-52
- "Collection Variable Declaration" on page 13-25

> **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 14-68.

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

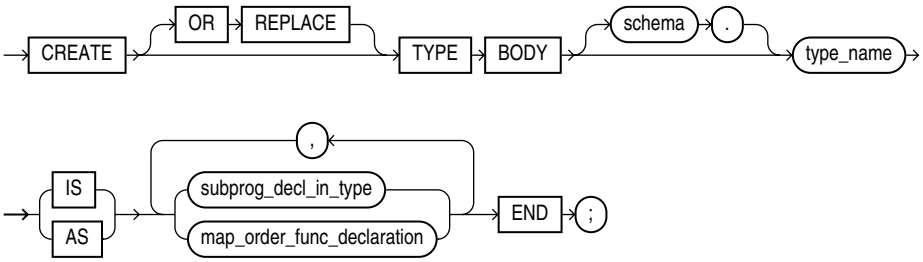
Prerequisites

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 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

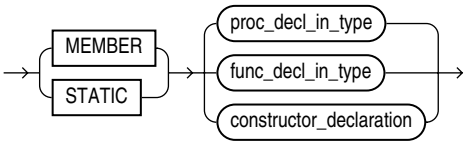
***create_type_body* ::=**



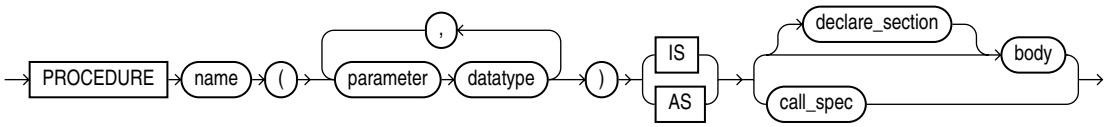
See:

- "map_order_func_declaration ::=" on page 14-86
- "subprog_decl_in_type ::=" on page 14-85

***subprog_decl_in_type* ::=**



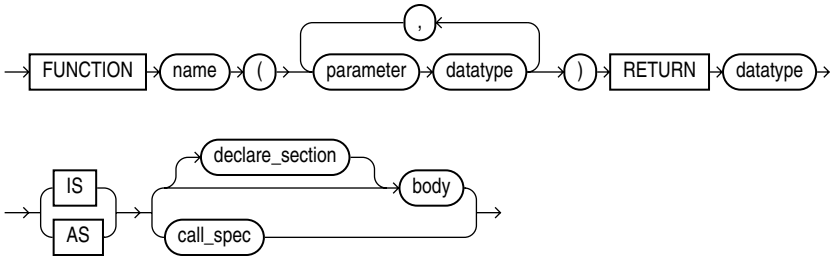
proc_decl_in_type ::=



See:

- "body ::=" on page 13-12
- "call_spec ::=" on page 14-87
- "declare_section ::=" on page 13-9

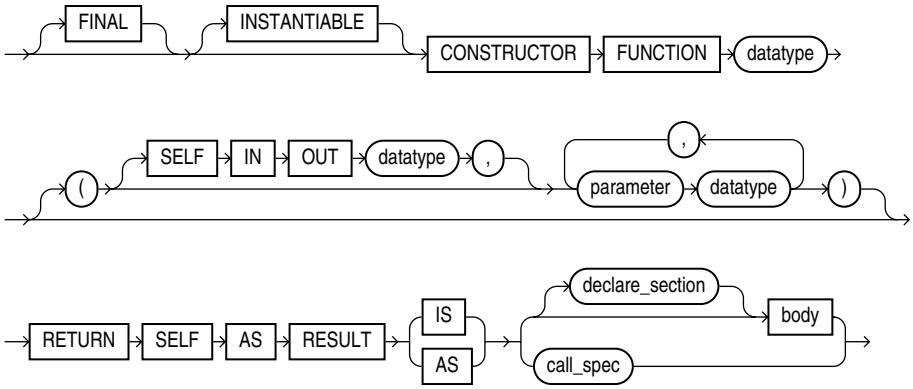
func_decl_in_type ::=



See:

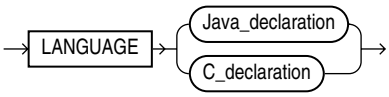
- "body ::=" on page 13-12
- "call_spec ::=" on page 14-87
- "declare_section ::=" on page 13-9

constructor_declaration ::=

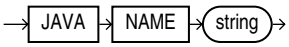


map_order_func_declaration ::=

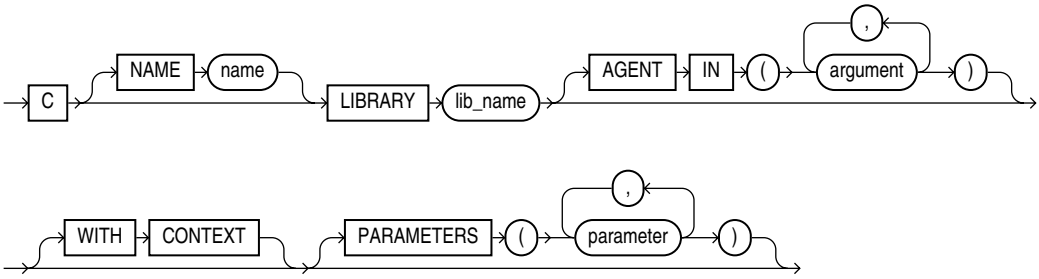
***call_spec* ::=**



***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

Name of the schema containing the type body. **Default:** your schema.

type_name

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 14-68 for a list of restrictions on user-defined functions
- "Overloaded Subprograms" on page 8-25 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

Maps a C procedure or Java 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* to learn how to write Java call specifications
- *Oracle Database Advanced Application Developer's Guide* to learn how to write C call specifications

EXTERNAL

Deprecated way of declaring a C method, supported only for backward compatibility. Oracle recommends that you use the `LANGUAGE C` syntax.

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 14-50 and "CREATE FUNCTION Statement" on page 14-32 for the full syntax with all possible clauses.

EXTERNAL

Deprecated way of declaring a C method, supported only for backward compatibility. Oracle recommends that you use the `LANGUAGE C` syntax.

Examples

Several examples of creating type bodies appear in the Examples section of "CREATE TYPE Statement" on page 14-68. 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 14-68
- "DROP TYPE BODY Statement" on page 14-100
- "CREATE FUNCTION Statement" on page 14-32
- "CREATE PROCEDURE Statement" on page 14-50

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 14-93 or redefine the package without the function using the "CREATE PACKAGE Statement" on page 14-43 with the `OR REPLACE` clause.

Topics

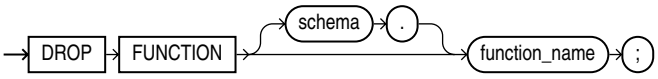
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

The function must be in your schema or you must have the `DROP ANY PROCEDURE` system privilege.

Syntax

***drop_function* ::=**



Semantics

schema

Name of the schema containing the function. **Default:** your schema.

function_name

Name of the function to be dropped.

The database invalidates any local objects that depend on, or invoke, 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 14-39 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

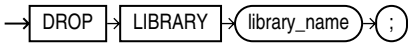
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

You must have the `DROP ANY LIBRARY` system privilege.

Syntax

***drop_library* ::=**



Semantics

library_name

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 14-42:

```
DROP LIBRARY ext_lib;
```

Related Topics

- "ALTER LIBRARY Statement" on page 14-6
- "CREATE LIBRARY Statement" on page 14-41

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 14-43 and "CREATE PACKAGE BODY Statement" on page 14-46 with the OR REPLACE clause.

Topics

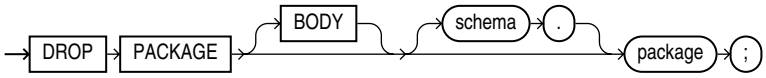
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

The package must be in your 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 invoke a procedure or stored function declared in the package specification until you re-create the package body.

schema

Name of the schema containing the package. **Default:** your schema.

package

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 14-47, 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 14-43
- "CREATE PACKAGE BODY Statement" on page 14-46

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 14-93, or redefine the package without the procedure using the "CREATE PACKAGE Statement" on page 14-43 with the OR REPLACE clause.

Topics

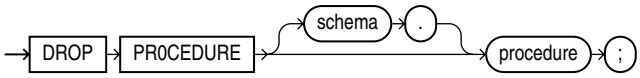
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

The procedure must be in your schema or you must have the DROP ANY PROCEDURE system privilege.

Syntax

***drop_procedure* ::=**



Semantics

schema

Name of the schema containing the procedure. **Default:** your schema.

procedure

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 14-50

DROP TRIGGER Statement

The `DROP TRIGGER` statement drops a database trigger from the database.

Topics

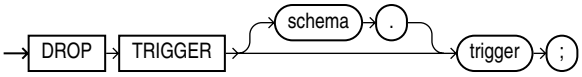
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

The trigger must be in your 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.

Syntax

***drop_trigger* ::=**



Semantics

schema

Name of the schema containing the trigger. **Default:** your schema.

trigger

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 14-54

DROP TYPE Statement

The `DROP TYPE` statement drops the specification and body of an ADT, `VARRAY` type, or nested table type.

Topics

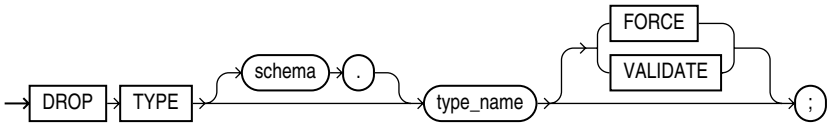
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

The ADT, `VARRAY` type, or nested table type must be in your schema or you must have the `DROP ANY TYPE` system privilege.

Syntax

***drop_type* ::=**



Semantics

schema

Name of the schema containing the type. **Default:** your schema.

type_name

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 14-81 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 14-68
- "CREATE TYPE BODY Statement" on page 14-85

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 invoke its member functions.

Topics

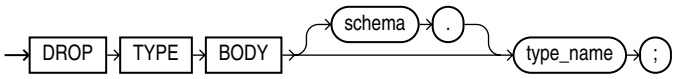
- Prerequisites
- Syntax
- Semantics
- Example
- Related Topics

Prerequisites

The type body must be in your schema or you must have the `DROP ANY TYPE` system privilege.

Syntax

***drop_type_body* ::=**



Semantics

schema

Name of the schema containing the type. **Default:** your schema.

type_name

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 14-79 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 14-68
- "CREATE TYPE BODY Statement" on page 14-85

A

PL/SQL Source Text Wrapping

You can wrap the PL/SQL source text for any of these stored PL/SQL units, thereby preventing anyone from displaying or editing that text:

- Package specification
- Package body
- Type specification
- Type body
- Function
- Procedure

A file containing wrapped PL/SQL source text is called a **wrapped file**. A wrapped file can be moved, backed up, or processed by SQL*Plus or the Import and Export utilities.

To produce a wrapped file, use either the PL/SQL Wrapper utility or a `DBMS_DDL` subprogram. The PL/SQL Wrapper utility wraps the source text of every wrappable PL/SQL unit created by a specified SQL file. The `DBMS_DDL` subprograms wrap the source text of single dynamically generated wrappable PL/SQL units.

Both the PL/SQL Wrapper utility and `DBMS_DDL` subprograms detect tokenization errors (for example, runaway strings), but not syntax or semantic errors (for example, nonexistent tables or views).

Wrapped files are upward-compatible between Oracle Database releases. For example, you can load files produced by the V8.1.5 PL/SQL Wrapper utility into a V8.1.6 Oracle Database.

Topics

- PL/SQL Source Text Wrapping Limitations
- PL/SQL Source Text Wrapping Guidelines
- PL/SQL Source Text Wrapping with PL/SQL Wrapper Utility
- PL/SQL Source Text Wrapping with DBMS_DDL Subprograms

PL/SQL Source Text Wrapping Limitations

- Wrapped files are not downward-compatible between Oracle Database releases.

  For example, you cannot load files produced by the V8.1.6 PL/SQL Wrapper utility into a V8.1.5 Oracle Database.

- Wrapping PL/SQL source text is not a secure way to hide passwords or table names.

  For high-assurance security, use Oracle Database Vault, described in *Oracle Database Vault Administrator's Guide*.

- You cannot wrap the PL/SQL source text of triggers.

  To hide the implementation details of a trigger, put them in a stored subprogram, wrap the subprogram, and write a one-line trigger that invokes the subprogram.

PL/SQL Source Text Wrapping Guidelines

- Wrap only the body of a package or type, not the specification.

  Leaving the specification unwrapped allows other developers to see the information needed to use the package or type (see Example A–5). Wrapping the body prevents them from seeing the package or type implementation.

- Wrap files only after you have finished editing them.

  You cannot edit wrapped files. If a wrapped file needs changes, you must edit the original unwrapped file and then wrap it.

- Before distributing a wrapped file, view it in a text editor and ensure that all important parts are wrapped.

PL/SQL Source Text Wrapping with PL/SQL Wrapper Utility

The PL/SQL Wrapper utility takes a single SQL file (such as a SQL*Plus script) and produces an equivalent text file in which the PL/SQL source text of each wrappable PL/SQL unit is wrapped. (For the list of wrappable PL/SQL units, see the introduction to "PL/SQL Source Text Wrapping" on page A-1.)

The PL/SQL Wrapper utility cannot connect to Oracle Database. To run the PL/SQL Wrapper utility, enter this command at the operating system prompt (with no spaces around the equal signs):

```
wrap iname=input_file [ oname=output_file ]
```

input_file is the name of an existing file that contains any combination of SQL statements. *output_file* is the name of the file that the PL/SQL Wrapper utility creates—the wrapped file.

> **Note:** *input_file* cannot include substitution variables specified with the SQL*Plus `DEFINE` notation, because *output_file* is parsed by the PL/SQL compiler, not by SQL*Plus.

The PL/SQL Wrapper utility deletes from the wrapped file all comments except:

- Comments in `CREATE` statement headers (that is, between `CREATE` and *plsql_source* in the syntax diagram in *Oracle Database SQL Language Reference*)
- Comments delimited by `/*` and `*/`

> **Note:** If *input_file* is a wrapped file, then *input_file* and *output_file* have identical contents.

The default file extension for *input_file* is `sql`. The default name of *output_file* is *input_file*`.plb`. Therefore, these commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql oname=/mydir/myfile.plb
```

This example specifies a different file extension for *input_file* and a different name for *output_file*:

```
wrap iname=/mydir/myfile.src oname=/yourdir/yourfile.out
```

You can run *output_file* as a script in SQL*Plus. For example:

```
SQL> @myfile.plb;
```

Example A–1 shows the text of a SQL file, `wraptest2.sql`, that contains two wrappable PL/SQL units—the procedure `wraptest` and the function `fibonacci`. The file also contains two comments and a SQL `SELECT` statement.

Example A–1 SQL File with Two Wrappable PL/SQL Units

```
-- The following statement will not change. This comment will be deleted.

SELECT COUNT(*) FROM EMPLOYEES
/

/* The PL/SQL source text of the following two CREATE statements
will be wrapped. This commment will not be deleted. */

CREATE PROCEDURE wraptest AUTHID DEFINER 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;
/

CREATE OR REPLACE FUNCTION fibonacci (
  n PLS_INTEGER
) RETURN PLS_INTEGER
AUTHID DEFINER
IS
  fib_1 PLS_INTEGER := 0;
  fib_2 PLS_INTEGER := 1;
BEGIN
  IF n = 1 THEN                              -- terminating condition
    RETURN fib_1;
  ELSIF n = 2 THEN
    RETURN fib_2;                           -- terminating condition
  ELSE
    RETURN fibonacci(n-2) + fibonacci(n-1);  -- recursive invocations
  END IF;
END;
/
```

Example A–2 uses the PL/SQL Wrapper utility to wrap `wraptest2.sql` and shows the wrapped file, `wraptest2.plb`. The wrapped file shows that the utility deleted the comment that begins with `--` and wrapped (made unreadable) the PL/SQL source

text of the procedure `wraptest` and the function `fibonacci`, but did not change the `SELECT` statement or the comment delimited by `/*` and `*/`.

Example A–2 Wrapping File with PL/SQL Wrapper Utility

Assume that the operating system prompt is >. Wrap the file `wraptest.sql`:

```
> wrap iname=wraptest2.sql
```

Result:

```
PL/SQL Wrapper: Release 11.2.0.1.0- Production on Wed Sep 15 08:10:15 2010

Copyright (c) 1993, 2009, Oracle.  All rights reserved.

Processing wraptest2.sql to wraptest2.plb
```

Contents of `wraptest.plb`:

```
SELECT COUNT(*) FROM EMPLOYEES
/
/* The PL/SQL source text of the following two CREATE statements
will be wrapped. This commment will not be deleted. */
CREATE PROCEDURE wraptest wrapped
a000000
b2
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
7
121 134
Pf3/wD+9ncRZhp3XxTMUO3yIRvswg+nQ7UhqfHRG2vg+SD7x9XzsDUFWbdwCJVEOLKBBRuH6
VMoRHfX6apzfyMkvWhzQLCYvAcq6Zu7++E7PrXNxUJzk/FZW8P9eRgyyyMFnDj53aP1cDje9
ZdGr2VmJHIw0ZNHBYhDdR+du5U5Yy47a6dJHXFW9eNyxBHtXZDuiWYTUtlnueHQV9iYDwE+r
jFn+eZm4jgDcTLTEzfmIVtPDRNhYCY3xhPo7vJeS8M1AvP+4xh9+uO35XsRIsRl1PTFVrGwg
6iuxETwA5Pu2mwx3

/
CREATE OR REPLACE FUNCTION fibonacci wrapped
a000000
b2
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
```

```
abcd
abcd
abcd
abcd
abcd
abcd
abcd
8
14a fb
e1Yq3QQJoEoNKIeJlbgLoLdSgogwgxDcf8vWfHSKbuowFOXFKoj9MqYGqWyRxeeCUVqNVIO1
ICqJa3yPr6e7z8GZpMH3J0Cx0uQ0B1JuysymdNDlzfTvb7QWsrLU4jGs3h8Mm49/L9nyO4Xh
Ae06nawFpOJIAYpBf9wBVC+ZrjU/nuEtokBqCce6HWIoF6rYgz0V0W/47x5KpOnQ2i7X3kFe
FR8K7jT7X58k8xK9uYlZv5LhV71a7A==

/
```

In SQL*Plus, Example A–3 runs the wrapped file `wraptest.plb`, creating the procedure `wraptest` and the function `fibonacci`; selects the text of the subprograms (which is wrapped and therefore unreadable), and then invokes the subprograms.

Example A–3 Running Wrapped File and Viewing Wrapped PL/SQL Units

```
SQL> -- Run wrapped file:
SQL>
SQL> @wraptest2.plb
SQL> SELECT COUNT(*) FROM EMPLOYEES
  2  /

  COUNT(*)
----------
       107

1 row selected.

SQL> /* The PL/SQL source text of the following two CREATE statements
SQL> will be wrapped. This commment will not be deleted. */
SQL> CREATE PROCEDURE wraptest wrapped
  2  a000000
  3  b2
  4  abcd
  5  abcd
  6  abcd
  7  abcd
  8  abcd
  9  abcd
 10  abcd
 11  abcd
 12  abcd
 13  abcd
 14  abcd
 15  abcd
 16  abcd
 17  abcd
 18  abcd
 19  7
 20  121 134
 21  Pf3/wD+9ncRZhp3XxTMUO3yIRvswg+nQ7UhqfHRG2vg+SD7x9XzsDUFWbdwCJVEOLKBBRuH6
 22  VMoRHfX6apzfyMkvWhzQLCYvAcq6Zu7++E7PrXNxUJzk/FZW8P9eRgyyyMFnDj53aP1cDje9
 23  ZdGr2VmJHIw0ZNHBYhDdR+du5U5Yy47a6dJHXFW9eNyxBHtXZDuiWYTUtlnueHQV9iYDwE+r
```

```
 24  jFn+eZm4jgDcTLTEzfmIVtPDRNhYCY3xhPo7vJeS8M1AvP+4xh9+uO35XsRIsRl1PTFVrGwg
 25  6iuxETwA5Pu2mwx3
 26
 27  /

Procedure created.

SQL> CREATE OR REPLACE FUNCTION fibonacci wrapped
  2  a000000
  3  b2
  4  abcd
  5  abcd
  6  abcd
  7  abcd
  8  abcd
  9  abcd
 10  abcd
 11  abcd
 12  abcd
 13  abcd
 14  abcd
 15  abcd
 16  abcd
 17  abcd
 18  abcd
 19  8
 20  14a fb
 21  e1Yq3QQJoEoNKIeJlbgLoLdSgogwgxDcf8vWfHSKbuowFOXFKoj9MqYGqWyRxeeCUVqNVIO1
 22  ICqJa3yPr6e7z8GZpMH3J0Cx0uQ0B1JuysymdNDlzfTvb7QWsrLU4jGs3h8Mm49/L9nyO4Xh
 23  Ae06nawFpOJIAYpBf9wBVC+ZrjU/nuEtokBqCce6HWIoF6rYgz0V0W/47x5KpOnQ2i7X3kFe
 24  FR8K7jT7X58k8xK9uYlZv5LhV71a7A==
 25
 26  /

Function created.

SQL>
SQL> -- Try to display procedure source text:
SQL>
SQL> SELECT text FROM USER_SOURCE WHERE name='WRAPTEST';

TEXT
--------------------------------------------------------------------------------
PROCEDURE wraptest wrapped
a000000
b2
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
```

```
abcd
7
121 134
Pf3/wD+9ncRZhp3XxTMUO3yIRvswg+nQ7UhqfHRG2vg+SD7x9XzsDUFWbdwCJVEOLKBBRuH6
VMoRHfX6apzfyMkvWhzQLCYvAcq6Zu7++E7PrXNxUJzk/FZW8P9eRgyyyMFnDj53aP1cDje9
ZdGr2VmJHIw0ZNHBYhDdR+du5U5Yy47a6dJHXFW9eNyxBHtXZDuiWYTUtlnueHQV9iYDwE+r
jFn+eZm4jgDcTLTEzfmIVtPDRNhYCY3xhPo7vJeS8M1AvP+4xh9+uO35XsRIsRl1PTFVrGwg
6iuxETwA5Pu2mwx3


1 row selected.

SQL>
SQL> -- Try to display function source text:
SQL>
SQL> SELECT text FROM USER_SOURCE WHERE name='FIBONACCI';

TEXT
--------------------------------------------------------------------------------
FUNCTION fibonacci wrapped
a000000
b2
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
8
14a fb
e1Yq3QQJoEoNKIeJlbgLoLdSgogwgxDcf8vWfHSKbuowFOXFKoj9MqYGqWyRxeeCUVqNVIO1
ICqJa3yPr6e7z8GZpMH3J0Cx0uQ0B1JuysymdNDlzfTvb7QWsrLU4jGs3h8Mm49/L9nyO4Xh
Ae06nawFpOJIAYpBf9wBVC+ZrjU/nuEtokBqCce6HWIoF6rYgz0V0W/47x5KpOnQ2i7X3kFe
FR8K7jT7X58k8xK9uYlZv5LhV71a7A==


1 row selected.

SQL>
SQL> BEGIN
  2    wraptest;  -- invoke procedure
  3    DBMS_OUTPUT.PUT_LINE('fibonacci(5) = ' || fibonacci(5));
  4  END;
  5  /
Emp Id: 198
Emp Id: 199
Emp Id: 200
Emp Id: 201
Emp Id: 202
Emp Id: 203
Emp Id: 204
```

```
Emp Id: 205
Emp Id: 206
Emp Id: 100
fibonacci(5) = 3

PL/SQL procedure successfully completed.

SQL>
```

PL/SQL Source Text Wrapping with DBMS_DDL Subprograms

The DBMS_DDL package provides WRAP functions and CREATE_WRAPPED procedures, each of which wraps the PL/SQL source text of a single dynamically generated wrappable PL/SQL unit. The DBMS_DDL package also provides the exception MALFORMED_WRAP_INPUT (ORA-24230), which is raised if the input to WRAP or CREATE_WRAPPED is not a valid wrappable PL/SQL unit. (For the list of wrappable PL/SQL units, see the introduction to "PL/SQL Source Text Wrapping" on page A-1.)

Each WRAP function takes as input a single CREATE statement that creates a wrappable PL/SQL unit and returns an equivalent CREATE statement in which the PL/SQL source text is wrapped. For more information about the WRAP functions, see *Oracle Database PL/SQL Packages and Types Reference*.

Caution: If you pass the statement that DBMS_DDL.WRAP returns to the DBMS_SQL.PARSE procedure whose formal parameter statement has data type VARCHAR2A, then you must set the lfflg parameter of DBMS_SQL.PARSE to FALSE. Otherwise, DBMS_SQL.PARSE adds lines to the wrapped PL/SQL unit, corrupting it. (For the syntax of DBMS_SQL.PARSE, see *Oracle Database PL/SQL Packages and Types Reference*.)

Each CREATE_WRAPPED procedure does what its corresponding WRAP function does and then runs the returned CREATE statement, creating the specified PL/SQL unit. For more information about the CREATE_WRAPPED procedures, see *Oracle Database PL/SQL Packages and Types Reference*.

Tip: When invoking a DBMS_DDL subprogram, use the fully qualified package name, SYS.DBMS_DDL, to avoid name conflict if someone creates a local package named DBMS_DDL or defines the public synonym DBMS_DDL.

Note: The CREATE statement that is input to a WRAP function or CREATE_WRAPPED procedure runs with the privileges of the user who invoked the subprogram.

Example A–4 dynamically creates a package specification (using the EXECUTE IMMEDIATE statement) and a wrapped package body, using a CREATE_WRAPPED procedure.

Example A–4 Creating Wrapped Package Body with CREATE_WRAPPED Procedure

```
DECLARE
  package_text  VARCHAR2(32767); -- text for creating package spec and body
```

```
  FUNCTION generate_spec (pkgname VARCHAR2) RETURN VARCHAR2 AS
  BEGIN
    RETURN 'CREATE PACKAGE ' || pkgname || ' AUTHID DEFINER 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
  package_text := generate_spec('emp_actions');  -- Generate package spec
  EXECUTE IMMEDIATE package_text;                -- Create package spec
  package_text := generate_body('emp_actions');  -- Generate package body
  SYS.DBMS_DDL.CREATE_WRAPPED(package_text);     -- Create wrapped package body
END;
/
```

Example A–5 selects the text of the package that Example A–4 created, `emp_actions`, and then invokes the procedure `emp_actions.raise_salary`. If the package specification were wrapped, then the information needed to invoke the procedure would be unreadable, like the PL/SQL source text of the package body.

Example A–5 Viewing Package with Wrapped Body and Invoking Package Procedure

Select text of package:

```
SELECT text FROM USER_SOURCE WHERE name = 'EMP_ACTIONS';
```

Result:

```
TEXT
----------------------------------------------------------------------

PACKAGE emp_actions AUTHID DEFINER AS
      PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
      PROCEDURE fire_employee (emp_id NUMBER);
      END emp_actions;
PACKAGE BODY emp_actions wrapped
a000000
369
abcd
abcd
abcd
abcd
abcd
abcd
```

```
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
abcd
b
180 113
1fOVodewm7j9dBOmBsiEQz0BKCgwg/BKoZ4VZy/pTBIYo8Uj1sjpbEz08Ck3HMjYq/Mf0XZn
u9D0Kd+i89g9ZO61I6vZYjw2AuBidnLESyR63LHZpFD/7lyDTfF1eDY5vmNwLTXrFaxGy243
0lHKAzmOlwwfBWylkZZNi2UnpmSIe6z/BU2nhbwfpqd224p69FwYVXmFX2H5IMsdZ2/vWsK9
cDMCD1KEqOnPpbU2yXdpW3GIbGD8JFIbKAfpJLkoLfVxoRPXQfj0h1k=
```

Invoke `raised_salary` and show its effect:

```
DECLARE
  s employees.salary%TYPE;
BEGIN
  SELECT salary INTO s FROM employees WHERE employee_id=130;
  DBMS_OUTPUT.PUT_LINE('Old salary: ' || s);
  emp_actions.raise_salary(130, 100);
  SELECT salary INTO s FROM employees WHERE employee_id=130;
  DBMS_OUTPUT.PUT_LINE('New salary: ' || s);
END;
/
```

Result:

```
Old salary: 2800
New salary: 2900

PL/SQL procedure successfully completed.
```

B

PL/SQL Name Resolution

This appendix explains PL/SQL **name resolution**; that is, how the PL/SQL compiler resolves ambiguous references to identifiers.

An unambiguous identifier reference can become ambiguous if you change identifiers in its compilation unit (that is, if you add, rename, or delete identifiers).

> **Note:** The `AUTHID` property of a stored PL/SQL unit affects the name resolution of SQL statements that the unit issues at run time. For more information, see "Invoker's Rights and Definer's Rights (AUTHID Property)" on page 8-43.

Topics

- Qualified Names and Dot Notation
- Column Name Precedence
- Differences Between PL/SQL and SQL Name Resolution Rules
- Resolution of Names in Static SQL Statements
- What is Capture?
- Avoiding Inner Capture in SELECT and DML Statements

Qualified Names and Dot Notation

When one named item belongs to another named item, you can (and sometimes must) qualify the name of the "child" item with the name of the "parent" item, using dot notation. For example:

When referencing ...	You must qualify its name with ...	Using this syntax ...
Field of a record	Name of the record	*record_name.field_name*
Method of a collection	Name of the collection	*collection_name.method*
Pseudocolumn `CURRVAL`	Name of a sequence	*sequence_name*`.CURRVAL`
Pseudocolumn `NEXTVAL`	Name of a sequence	*sequence_name*`.NEXTVAL`

If an identifier is declared in a named PL/SQL unit, you can qualify its simple name (the name in its declaration) with the name of the unit (block, subprogram, or package), using this syntax:

unit_name.simple_identifier_name

If the identifier is not visible, then you *must* qualify its name (see "Scope and Visibility of Identifiers" on page 2-16).

If an identifier belongs to another schema, then you must qualify its name with the name of the schema, using this syntax:

schema_name.package_name

A simple name can be qualified with multiple names, as Example B–1 shows.

Example B–1 Qualified 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 invocation followed by component name
     n := f2(10)(10).a;  -- Function invocation followed by indexed name
                         --  and followed by component name
     n := hr.pkg1.f2(10)(10).a;  -- Schema name, package name,
                                 -- function invocation, 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;
/
```

Some examples of possibly ambiguous qualified names 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()
```

Column Name Precedence

If a SQL statement references a name that belongs to both a column and either a local variable or formal parameter, then the column name takes precedence.

> **Caution:** When a variable or parameter name is interpreted as a column name, data can be deleted, changed, or inserted unintentionally.

In Example B–2, the name `last_name` belongs to both a local variable and a column (names are not case-sensitive). Therefore, in the `WHERE` clause, both references to `last_name` resolve to the column, and all rows are deleted.

Example B–2 Variable Name Interpreted as Column Name Causes Unintended Result

```
DROP TABLE employees2;
CREATE TABLE employees2 AS
  SELECT LAST_NAME FROM employees;

DECLARE
  last_name  VARCHAR2(10) := 'King';
BEGIN
  DELETE FROM employees2 WHERE LAST_NAME = last_name;
  DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
END;
/
```

Result:

```
Deleted 107 rows.
```

Example B–3 solves the problem in Example B–2 by giving the variable a different name.

Example B–3 Fixing Example B–2 with Different Variable Name

```
DECLARE
  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.
```

Example B–4 solves the problem in Example B–2 by labeling the block and qualifying the variable name with the block name.

Example B–4 Fixing Example B–2 with Block Label

```
<<main>>
DECLARE
  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.
```

In Example B–5, the the function `dept_name` has a formal parameter and a local variable whose names are those of columns of the table `DEPARTMENTS`. The parameter and variable name are qualified with the function name to distinguish them from the column names.

Example B–5 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
    ORDER BY department_name) LOOP

      DBMS_OUTPUT.PUT_LINE ('Department: ' || dept_name(item.department_id));
  END LOOP;
END;
/
```

Result:

```
Department: Accounting
Department: Administration
Department: Benefits
```

```
Department: Construction
Department: Contracting
Department: Control And Credit
Department: Corporate Tax
Department: Executive
Department: Finance
Department: Government Sales
Department: Human Resources
Department: IT
Department: IT Helpdesk
Department: IT Support
Department: Manufacturing
Department: Marketing
Department: NOC
Department: Operations
Department: Payroll
Department: Public Relations
Department: Purchasing
Department: Recruiting
Department: Retail Sales
Department: Sales
Department: Shareholder Services
Department: Shipping
Department: Treasury
```

Differences Between PL/SQL and SQL Name Resolution Rules

PL/SQL and SQL name resolution rules are very similar. However:

- PL/SQL rules are less permissive than SQL rules.

  Because most SQL rules are context-sensitive, they recognize as legal more situations than PL/SQL rules do.

- PL/SQL and SQL resolve qualified names differently.

  For example, when resolving the table name `HR.JOBS`:

  – PL/SQL searches first for packages, types, tables, and views named `HR` in the current schema, and then for objects named `JOBS` in the `HR` schema.

  – SQL searches first for objects named `JOBS` in the `HR` schema, and then for packages, types, tables, and views named `HR` in the current schema.

To avoid problems caused by the few differences between PL/SQL and SQL name resolution rules, follow the recommendations in "Avoiding Inner Capture in SELECT and DML Statements" on page B-7.

> **Note:** When the PL/SQL compiler processes a static SQL statement, it sends that statement to the SQL subsystem, which uses SQL rules to resolve names in the statement. For details, see "Resolution of Names in Static SQL Statements" on page B-5.

Resolution of Names in Static SQL Statements

Static SQL is described in Chapter 6, "PL/SQL Static SQL".

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 names in the scope of the SQL statement.
4. If the SQL subsystem cannot resolve a name in the scope of the SQL statement, then it sends the name back to the PL/SQL compiler. The name 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.
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 computer code.
7. At run time, the PL/SQL run-time system invokes 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.

What is Capture?

When a declaration or definition prevents the compiler from correctly resolving a reference in another scope, the declaration or definition is said to **capture** the reference. Capture is usually the result of migration or schema evolution.

Topics

- Outer Capture
- Same-Scope Capture
- Inner Capture

Note: Same-scope and inner capture occur only in SQL scope.

Outer Capture

Outer capture occurs when a name in an inner scope, which had resolved to an item in an inner scope, now resolves to an item in an outer scope. Both PL/SQL and SQL are designed to prevent outer capture; you need not be careful to avoid it.

Same-Scope Capture

Same-scope capture occurs when a column is added to one of two tables used in a join, and the new column has the same name as a column in the other table. When only one table had a column with that name, the name could appear in the join

unqualified. Now, to avoid same-scope capture, you must qualify the column name with the appropriate table name, everywhere that the column name appears in the join.

Inner Capture

Inner capture occurs when a name in an inner scope, which had resolved to an item in an outer scope, now either resolves to an item in an inner scope or cannot be resolved. In the first case, the result might change. In the second case, an error occurs.

In Example B–6, a new column captures a reference to an old column with the same name. Before new column `col2` is added to table `tab2`, `col2` resolves to `tab1.col2`; afterward, it resolves to `tab2.col2`.

Example B–6 Inner Capture of Column Reference

Table `tab1` has a column named `col2`, but table `tab2` does not:

```
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);
```

Therefore, in the inner `SELECT` statement, the reference to `col2` resolves to column `tab1.col2`:

```
CREATE OR REPLACE PROCEDURE proc AS
  CURSOR c1 IS
    SELECT * FROM tab1
    WHERE EXISTS (SELECT * FROM tab2 WHERE col2 = 10);
BEGIN
  OPEN c1;
  CLOSE c1;
END;
/
```

Add a column named `col2` to table `tab2`:

```
ALTER TABLE tab2 ADD (col2 NUMBER);
```

Now procedure `proc` is invalid. At its next invocation, the database automatically recompiles it, and the reference to `col2` in the inner `SELECT` statement resolves to column `tab2.col2`.

To avoid inner capture, follow the rules in "Avoiding Inner Capture in SELECT and DML Statements" on page B-7.

Avoiding Inner Capture in SELECT and DML Statements

Avoid inner capture of references in `SELECT`, `SELECT INTO`, and DML statements by following these recommendations:

- Specify a unique alias for each table in the statement.
- Do not specify a table alias that is the name of a schema that owns an item referenced in the statement.
- Qualify each column reference in the statement with the appropriate table alias.

In Example B–7, schema hr owns tables tab1 and tab2. Table tab1 has a column named tab2, whose Abstract Data Type (ADT) has attribute a. Table tab2 does not have a column named a. Against recommendation, the query specifies alias hr for table tab1 and references table tab2. Therefore, in the query, the reference hr.tab2.a resolves to table tab1, column tab2, attribute a. Then the example adds column a to table tab2. Now the reference hr.tab2.a in the query resolves to schema hr, table tab2, column a. Column a of table tab2 captures the reference to attribute a in column tab2 of table tab1.

Example B–7 Inner Capture of Attribute Reference

```
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.
```

Add a column named a to table tab2 (which belongs to schema hr):

```
ALTER TABLE tab2 ADD (a NUMBER);
```

Now, when the query runs, hr.tab2.a resolves to schema hr, table tab2, column a. To avoid this inner capture, apply the recommendations to the query:

```
SELECT * FROM hr.tab1 p1
WHERE EXISTS (SELECT * FROM hr.tab2 p2 WHERE p2.x = p1.tab2.a);
```

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 element, you must give the table an alias and use the alias to qualify the reference to the attribute or method.

In Example B–8, table tbl1 has column col1 of data type t1, an ADT with attribute x. The example shows several correct and incorrect references to tbl1.col1.x.

Example B–8 Qualifying ADT Attribute References

```
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 to the attribute `x` 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 to the attribute `x`, with table aliases, are correct:

```
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. A row expression can appear in the `SET` clause of an `UPDATE` statement or be the parameter of the SQL function `REF` or `VALUE`.

In Example B–9, table `ot1` is a standalone nested table of elements of data type `t1`, an ADT with attribute `x`. The example shows several correct and incorrect references to row expressions.

Example B–9 Qualifying References to Row Expressions

```
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 text 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_INTEGER`value	-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
size of a trigger	32 K

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' ORDER BY type;
```

Result:

```
NAME TYPE         SOURCE_SIZE PARSED_SIZE CODE_SIZE ERROR_SIZE
---- ------------ ----------- ----------- --------- ----------
PKG1 PACKAGE              112         464       262         79
PKG1 PACKAGE BODY         233         103       314          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."

D

PL/SQL Reserved Words and Keywords

Reserved words (listed in Table D–1) and **keywords** (listed in Table D–2) are identifiers that have special meaning in PL/SQL. They are case-insensitive. For more information about them, see "Reserved Words and Keywords" on page 2-5.

> **Note:** 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, CURSOR
D	DECLARE, DEFAULT, DESC, DISTINCT, DROP
E	ELSE, END, EXCEPTION, EXCLUSIVE
F	FETCH, FOR, FROM, FUNCTION
G	GOTO, GRANT, GROUP
H	HAVING
I	IDENTIFIED, IF, IN, INDEX, INDEXES, INSERT, INTERSECT, INTO, IS
L	LIKE, LOCK
M	MINUS, MODE
N	NOCOMPRESS, NOT, NOWAIT, NULL
O	OF, ON, OPTION, OR, ORDER, OVERLAPS
P	PROCEDURE, PUBLIC
R	RESOURCE, REVOKE
S	SELECT, SHARE, SIZE, SQL, START, SUBTYPE
T	TABAUTH, TABLE, THEN, TO, TYPE
U	UNION, UNIQUE, UPDATE
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, CHARSET, CHARSETFORM, CHARSETID, CLOB_BASE, CLOSE, COLLECT, COMMENT, COMMIT, COMMITTED, COMPILED, CONSTANT, CONSTRUCTOR, CONTEXT, CONTINUE, CONVERT, COUNT, CURRENT, CUSTOMDATUM
D	DANGLING, DATA, DATE, DATE_BASE, DAY, DEFINE, DELETE, DETERMINISTIC, DOUBLE, DURATION
E	ELEMENT, ELSIF, EMPTY, ESCAPE, EXCEPT, EXCEPTIONS, EXECUTE, EXISTS, EXIT, EXTERNAL
F	FINAL, FIXED, FLOAT, FORALL, FORCE
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, OCIDATE, OCIDATETIME, OCIDURATION, OCIINTERVAL, OCILOBLOCATOR, OCINUMBER, OCIRAW, OCIREF, OCIREFCURSOR, OCIROWID, OCISTRING, OCITYPE, OLD, ONLY, OPAQUE, OPEN, OPERATOR, ORACLE, ORADATA, ORGANIZATION, ORLANY, ORLVARY, OTHERS, OUT, OVERRIDING
P	PACKAGE, PARALLEL_ENABLE, PARAMETER, PARAMETERS, PARENT, PARTITION, PASCAL, PIPE, PIPELINED, PRAGMA, PRECISION, PRIOR, 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, SUM, SYNONYM
T	TDO, THE, TIME, TIMESTAMP, TIMEZONE_ABBR, TIMEZONE_HOUR, TIMEZONE_MINUTE, TIMEZONE_REGION, TRAILING, TRANSACTION, TRANSACTIONAL, TRUSTED
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

E

PL/SQL Predefined Data Types

This appendix groups by data type family the data types and subtypes that the package STANDARD predefines.

BFILE Data Type Family

```
type BFILE is BFILE_BASE;
```

BLOB Data Type Family

```
type BLOB is BLOB_BASE;

subtype "BINARY LARGE OBJECT" is BLOB;
```

BOOLEAN Data Type Family

```
type BOOLEAN is (FALSE, TRUE);
```

CHAR Data Type Family

```
type VARCHAR2 is new CHAR_BASE;
type MLSLABEL is new CHAR_BASE;
type UROWID   is new CHAR_BASE;

subtype VARCHAR              is VARCHAR2;
subtype STRING               is VARCHAR2;
subtype LONG                 is VARCHAR2(32760);
subtype RAW                  is VARCHAR2;
subtype "LONG RAW"           is RAW(32760);
subtype ROWID                is VARCHAR2(256);
subtype CHAR                 is VARCHAR2;
subtype CHARACTER            is CHAR;
subtype "CHARACTER VARYING"  is VARCHAR;
subtype "CHAR VARYING"       is VARCHAR;
subtype "NATIONAL CHARACTER" is CHAR CHARACTER SET NCHAR_CS;
subtype "NATIONAL CHAR"      is CHAR CHARACTER SET NCHAR_CS;
subtype "NCHAR"              is CHAR CHARACTER SET NCHAR_CS;
subtype "NVARCHAR2"          is VARCHAR2 CHARACTER SET NCHAR_CS;
```

CLOB Data Type Family

```
type CLOB is CLOB_BASE;

subtype "CHARACTER LARGE OBJECT"         is CLOB;
subtype "CHAR LARGE OBJECT"              is CLOB;
subtype "NATIONAL CHARACTER LARGE OBJEC" is CLOB CHARACTER SET NCHAR_CS;
subtype "NCHAR LARGE OBJECT"             is CLOB CHARACTER SET NCHAR_CS;
subtype "NCLOB"                          is CLOB CHARACTER SET NCHAR_CS;
```

DATE Data Type Family

```
type DATE                           is     DATE_BASE;
type TIME                           is new DATE_BASE;
type TIMESTAMP                      is new DATE_BASE;
type "TIME WITH TIME ZONE"          is new DATE_BASE;
type "TIMESTAMP WITH TIME ZONE"     is new DATE_BASE;
type "INTERVAL YEAR TO MONTH"       is new DATE_BASE;
type "INTERVAL DAY TO SECOND"       is new DATE_BASE;
type "TIMESTAMP WITH LOCAL TIME ZONE" is new DATE_BASE;

subtype TIME_UNCONSTRAINED          is TIME(9);
subtype TIME_TZ_UNCONSTRAINED       is TIME(9) WITH TIME ZONE;
subtype TIMESTAMP_UNCONSTRAINED     is TIMESTAMP(9);
subtype TIMESTAMP_TZ_UNCONSTRAINED  is TIMESTAMP(9) WITH TIME ZONE;
subtype YMINTERVAL_UNCONSTRAINED    is INTERVAL YEAR(9) TO MONTH;
subtype DSINTERVAL_UNCONSTRAINED    is INTERVAL DAY(9) TO SECOND (9);
subtype TIMESTAMP_LTZ_UNCONSTRAINED is TIMESTAMP(9) WITH LOCAL TIME ZONE;
```

NUMBER Data Type Family

```
type NUMBER is NUMBER_BASE;

subtype FLOAT              is NUMBER; -- NUMBER(126)
subtype REAL               is FLOAT;   -- FLOAT(63)
subtype "DOUBLE PRECISION" is FLOAT;

subtype INTEGER  is NUMBER(38,0);
subtype INT      is INTEGER;
subtype SMALLINT is NUMBER(38,0);

subtype DECIMAL is NUMBER(38,0);
subtype NUMERIC is DECIMAL;
subtype DEC     is DECIMAL;

subtype BINARY_INTEGER is INTEGER range '-2147483647'..2147483647;
subtype NATURAL        is BINARY_INTEGER range 0..2147483647;
subtype NATURALN       is NATURAL not null;
subtype POSITIVE       is BINARY_INTEGER range 1..2147483647;
subtype POSITIVEN      is POSITIVE not null;
subtype SIGNTYPE       is BINARY_INTEGER range '-1'..1;  -- for SIGN functions
subtype PLS_INTEGER    is BINARY_INTEGER;

type BINARY_FLOAT  is NUMBER;
type BINARY_DOUBLE is NUMBER;

subtype SIMPLE_INTEGER is BINARY_INTEGER NOT NULL;
subtype SIMPLE_FLOAT   is BINARY_FLOAT   NOT NULL;
subtype SIMPLE_DOUBLE  is BINARY_DOUBLE  NOT NULL;
```

See Also:

- Chapter 3, "PL/SQL Data Types" for more information about PL/SQL data types
- "User-Defined PL/SQL Subtypes" on page 3-11 for information that also applies to predefined subtypes

Index

Symbols

$$PLSQL_LINE inquiry directive, 2-44
$$PLSQL_UNIT inquiry directive, 2-44
% wildcard character, 2-34
%BULK_EXCEPTIONS cursor attribute, 12-19
%BULK_ROWCOUNT cursor attribute, 12-21
%FOUND cursor attribute
  for implicit cursor, 6-6
  for named cursor, 6-20
%ISOPEN cursor attribute
  for implicit cursor, 6-6
  for named cursor, 6-19
%NOTFOUND cursor attribute
  for implicit cursor, 6-7
  for named cursor, 6-21
%ROWCOUNT cursor attribute
  for implicit cursor, 6-7
  for named cursor, 6-22
%ROWTYPE attribute
  in general, 5-42
  column alias and, 6-14
  explicit cursor and, 6-14
  syntax diagram, 13-121
%TYPE attribute
  in general, 2-14
  initial value and, 2-15
  NOT NULL constraint and, 2-15
  syntax diagram, 13-134
_ wildcard character, 2-34

A

Abstract Data Type (ADT)
  in general, 1-8
  creating, 14-68
  editioned, 14-21
  for use in any schema, 8-47
accent-insensitive comparison, 2-34
ACCESS_INTO_NULL exception, 11-10
ADT
  See Abstract Data Type (ADT)
aggregate function
  in general, 14-37
  in PL/SQL expression, 2-40
  pipelined table function as, 12-45
  SQL%NOTFOUND attribute and, 6-7
alias
  column
    in cursor FOR LOOP, 6-24
    in explicit cursor, 6-14
  table
    for avoiding inner capture, B-7
    for row expression, B-9
    for table element attribute or method, B-8
aliasing (problem)
  SELECT BULK COLLECT INTO statement and, 12-26
  subprogram parameter, 8-15
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, 2-27
anonymous block
  in general, 1-5
  AUTHID property and, 8-44
ANYDATA data type, 12-40
ANYDATASET data type, 12-40
ANYTYPE data type, 12-40
architecture of PL/SQL, 1-10
array
  associative
    See associative array
  non-PL/SQL, 5-4
assignment of value
  to composite variable
    collection, 5-14
    record, 5-44
  to scalar variable, 2-20
assignment statement
  in general, 2-21
  syntax diagram, 13-3
associative array
  in general, 5-4
  as standalone stored subprogram parameter, 5-7
  characteristics of, 5-3
  comparisons, 5-18
  FIRST and LAST methods for, 5-29
  in FORALL statement, 13-78

NLS parameters and, 5-6
pipelined table function and, 12-40
See also collection
atomic (lexical) unit, 2-3
atomically null collection
See null collection
attribute
%ROWTYPE
See %ROWTYPE attribute
%TYPE
See %TYPE attribute
cursor
See cursor attribute
AUTHID property, 8-43
autonomous transaction
in general, 6-49
pipelined table function in, 13-106
autonomous trigger, 6-53
AUTONOMOUS_TRANSACTION pragma
in general, 13-6
for pipelined table function, 12-40

B

bag data structure, 5-4
base type, 3-1
basic LOOP statement
in general, 4-9
syntax diagram, 13-7
BETWEEN operator, 2-35
BINARY_DOUBLE data type
predefined constants for, 3-2
subtype of, 3-3
tuning code and, 12-9
BINARY_FLOAT data type
predefined constants for, 3-2
subtype of, 3-3
tuning code and, 12-9
BINARY_INTEGER data type
See PLS_INTEGER data type
bind argument
avoiding SQL injection with, 7-15
placeholder for
See placeholder for bind argument
blank-padding
in assignment, 3-5
in comparison, 3-6
block syntax, 13-9
BOOLEAN data type, 3-7
BOOLEAN expression, 2-37
BOOLEAN static expression, 2-47
BOOLEAN variable, 2-23
bounded collection, 5-3
built-in function
See SQL function
bulk binding, 12-11
BULK COLLECT clause
in general, 12-23
aliasing and, 12-26
of FETCH statement, 12-31
of RETURNING INTO clause
in general, 12-35
FORALL statement and, 12-36
of SELECT INTO statement, 12-24
query result set processing and, 6-23
that returns no rows, 12-24
bulk SQL
in general, 12-11
in compound DML trigger, 9-16

C

C procedure, invoking, 8-49
cache, function result, 8-31
calculated column, 6-14
call specification
in general, 8-49
in CREATE FUNCTION statement, 14-32
in CREATE PROCEDURE statement, 14-50
in package, 10-2
call stack, AUTHID property and, 8-44
capture, B-6
cascading triggers, 9-41
CASE expression
searched, 2-39
simple, 2-38
case sensitivity
character comparison and, 2-34
character literal and, 2-9
identifier and
in general, 2-5
quoted user-defined identifier, 2-6
keyword and, D-1
LIKE operator and, 2-34
reserved word and, D-1
CASE statement
in general, 4-1
searched
in general, 4-7
syntax diagram, 13-20
simple
in general, 4-6
IF THEN ELSIF statement and, 4-6
syntax diagram, 13-20
CASE_NOT_FOUND exception, 11-10
case-insensitive comparison, 2-34
CHAR data type, 3-3
CHAR data type family, E-1
character code, 2-1
character literal, 2-9
See also string
character set, 2-1
CLOB data type and comparison operator, 2-32
CLOB data type family, E-1
CLOSE statement, 13-23
collating sequence, 2-33
collection
in general, 5-1
as public package item, 5-37
assigning one to another, 5-14

bounded, 5-3
comparing one to another, 5-18
cursor variable and, 13-42
declaration syntax, 13-25
empty
in general, 5-3
creating with constructor, 5-13
index
See index collection
internal size of
DELETE method and, 5-22
EXTEND method and, 5-27
TRIM method and, 5-25
multidimensional, 5-17
null
in general, 5-3
assigning to collection variable, 5-15
pipelined table function and, 12-39
retrieving query results into, 12-23
types of, 5-2
unbounded, 5-3
collection constructor, 5-13
collection method
in general, 5-21
as subprogram parameter, 5-22
invocation syntax, 13-31
null collection and, 5-21
COLLECTION_IS_NULL exception, 11-10
column alias
in cursor FOR LOOP, 6-24
in explicit cursor, 6-14
comment
in general, 2-9
nested, 2-10
PL/SQL Wrapper utility and, A-2
syntax diagram, 13-34
COMMIT statement
in general, 6-40
FOR UPDATE cursor and, 6-48
in autonomous transaction, 6-53
comparison
of collections, 5-18
of records, 5-49
comparison operator
in general, 2-32
cursor variable and, 13-42
compatible data type
for collection variables, 5-14
for scalar variables, 2-20
compilation
conditional, 2-41
for native execution, 12-51
interpreted, 12-53
compilation parameter
in general, 1-11
displaying value of, 2-45
predefined inquiry directive for, 2-44
compiler directive
See pragma
compile-time warning, 11-2
composite data type, 5-1
composite variable, 5-1
compound DML trigger, 9-14
computation-intensive code, 12-8
concatenation operator (||), 2-24
concurrent transactions, 6-53
condition, SQL multiset, 5-20
conditional compilation, 2-41
conditional compilation directive
in general, 2-42
error, 2-43
inquiry, 2-43
restrictions on, 2-52
selection, 2-42
conditional predicate, 9-4
conditional selection statement, 4-1
conditional trigger, 9-2
constant
declaring
in general, 2-12
syntax diagram, 13-36
initial value of, 2-13
predefined, 3-2
static
in general, 2-48
in DBMS_DB_VERSION package, 2-49
constrained subtype
in general, 3-12
in performance-critical code, 12-9
subprogram parameter and, 8-10
constraint
cursor parameter and, 13-59
NOT NULL
See NOT NULL constraint
trigger compared to, 9-3
constructor
See collection constructor
context of transaction, 6-50
CONTINUE statement
in general, 4-11
syntax diagram, 13-38
CONTINUE WHEN statement
in general, 4-12
syntax diagram, 13-38
control statement, 4-1
control token, 2-42
correlated subquery, 6-27
correlation name
in general, 9-5
with LONG or LONG RAW column, 9-37
COUNT collection method, 5-32
CREATE FUNCTION statement, 14-32
CREATE LIBRARY statement, 14-41
CREATE PACKAGE statement, 14-43
CREATE TRIGGER statement, 14-54
CREATE TYPE BODY statement, 14-85
CREATE TYPE statement, 14-68
CREATE_WRAPPED procedure, A-8
crossedition trigger, 9-4
CURRENT OF clause

in general, 6-47
FOR UPDATE cursor and, 6-47
ROWID pseudocolumn instead of, 6-48
CURRENT_USER, 8-45
CURRVAL pseudocolumn, 6-3
cursor
in general, 6-5
explicit
See explicit cursor
FOR UPDATE
in general, 6-47
after COMMIT or ROLLBACK, 6-48
implicit
See implicit cursor
in SERIALLY_REUSABLE package, 10-9
named
in general, 6-8
pipelined table function and, 12-43
See also explicit cursor *and* cursor variable
nested, 6-38
cursor attribute
for cursor variable, 6-35
for explicit cursor
in general, 6-19
%FOUND, 6-20
%ISOPEN, 6-19
%NOTFOUND, 6-21
%ROWCOUNT, 6-22
for implicit cursor
in general, 6-5
DBMS_SQL package and, 7-7
native dynamic SQL and, 7-2
SQL%BULK_EXCEPTIONS, 12-19
SQL%BULK_ROWCOUNT, 12-21
SQL%FOUND, 6-6
SQL%ISOPEN, 6-6
SQL%NOTFOUND, 6-7
SQL%ROWCOUNT, 6-7
where you can use, 6-5
CURSOR expression
in general, 6-38
passing to pipelined table function, 12-44
cursor FOR LOOP statement
query result set processing with, 6-24
recursive invocation in, 8-31
syntax diagram, 13-40
cursor parameter, 6-15
cursor specification, 13-57
cursor variable
in general, 6-28
declaration syntax diagram, 13-42
CURSOR_ALREADY_OPEN exception, 11-10

D

data abstraction, 1-7
data definition language statement
See DDL statement
data manipulation language statement
See DML statement
Data Pump Import and triggers, 9-42
data type
collection
See collection
compatible
for collection variables, 5-14
for scalar variables, 2-20
composite, 5-1
object
See Abstract Data Type (ADT)
of expression, 2-24
predefined, E-1
RECORD
See record
scalar, 3-1
SQL, 3-2
user-defined
See Abstract Data Type (ADT)
what it determines, 3-1
See also subtype
data type conversion
in general, 3-2
implicit
See implicit data type conversion
SQL injection and, 7-13
data type family
in general, 3-1
overloaded subprogram and, 8-25
predefined data types grouped by, E-1
subtypes with base types in same, 3-14
database character set, 2-1
database link, IR, 8-46
DATABASE trigger, 9-31
DATE data type family, E-2
DBMS_ASSERT package, 7-16
DBMS_DB_VERSION package, 2-49
DBMS_DDL package, A-8
DBMS_PARALLEL_EXECUTE package, 12-48
DBMS_PREPROCESSOR package, 2-51
DBMS_PROFILE package, 12-50
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_STANDARD package, 2-16
DBMS_TRACE package, 12-50
DBMS_WARNING package, 11-3
dbmsupgin.sql script, 12-53
dbmsupgnv.sql script, 12-53
DDL statement
dynamic SQL for, 7-1
in trigger, 6-53
subprogram side effects and, 8-43
deadlock
autonomous transaction and, 6-53
implicit rollback and, 6-45
declaration
in general, 2-11
exception raised in, 11-20

default value
of cursor parameter, 6-16
of subprogram parameter, 8-18
See also initial value
DEFINE
binding category, 12-11
PL/SQL Wrapper utility and, A-2
DEFINER, 8-45
definer's rights unit
See DR unit
DELETE collection method
in general, 5-22
COUNT method and, 5-32
EXISTS method and, 5-28
EXTEND method and, 5-27
FIRST method and, 5-28
LAST method and, 5-28
NEXT method and, 5-35
PRIOR method and, 5-35
TRIM method and, 5-25
DELETE statement
BEFORE statement trigger and, 9-35
PL/SQL extension to, 13-45
See also DML statement
DELETING conditional predicate, 9-4
delimiter, 2-3
dense collection, 5-3
Descriptive Intermediate Attributed Notation for Ada (DIANA), C-1
DETERMINISTIC option
in general, 13-84
for pipelined table function, 12-40
directive
compiler
See pragma
error, 2-43
inquiry, 2-43
selection, 2-42
See also conditional compilation directive
DML statement
in FORALL statement, 12-12
inside pipelined table function, 12-40
on pipelined table function result, 12-46
PL/SQL syntax of, 6-1
repeating efficiently, 12-12
DML statement statement
avoiding inner capture in, B-7
DML trigger, 9-3
dot notation
for collection method, 5-22
for identifier in named PL/SQL unit, 2-16
for pseudocolumn, 6-3
for record field, 5-1
name resolution and, B-1
double quotation mark ("), 2-2
DR unit
call stack and, 8-44
dynamic SQL and, 8-44
name resolution and, 8-44
privilege checking and, 8-44
SET ROLE command and, 8-45
static SQL and, 8-44
See also AUTHID property
DROP FUNCTION statement, 14-90
DROP LIBRARY statement, 14-92
DROP PACKAGE statement, 14-93
DROP PROCEDURE statement, 14-95
DROP TRIGGER statement, 14-97
DROP TYPE BODY statement, 14-100
DUP_VAL_ON_INDEX exception, 11-10
dynamic SQL
in general, 7-1
AUTHID property and, 8-44
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, 12-4

E

editioned Abstract Data Type (ADT), 14-21
element of collection, 5-1
embedded SQL
See static SQL
empty collection
in general, 5-3
creating with constructor, 5-13
error directive, 2-43
error handling, 11-1
error-reporting function
SQLCODE, 13-131
SQLERRM
in general, 13-132
SQL%BULK_EXCEPTIONS and, 12-20
escape character, 2-35
escaped identifier, B-6
evaluation order, 2-24
events publication, 9-43
evolution of type, 14-17
exception
handling
overview of, 11-4
in FORALL statement, 12-18
in trigger, 9-33
See also exception handler
internally defined
See internally defined exception
predefined
See predefined exception
raised in cursor FOR LOOP statement, 6-26
raised in declaration, 11-20
raised in exception handler, 11-20
raising explicitly, 11-13
reraising, 11-15
unhandled
in general, 11-24
in FORALL statement, 12-17

OUT and IN OUT parameters and, 8-13
user-defined
See user-defined exception
exception handler
in general, 11-4
continuing execution after, 11-25
exception raised in, 11-20
for NO_DATA_NEEDED, 12-47
GOTO statement and, 13-87
locator variables for, 11-7
retrieving error code and message in, 11-24
retrying transaction after, 11-27
syntax diagram, 13-50
EXCEPTION_INIT pragma
in general, 13-46
for giving error code to user-defined exception, 11-16
for giving name to internally defined exception, 11-9
EXECUTE IMMEDIATE statement
in general, 7-2
syntax diagram, 13-52
EXISTS collection method, 5-28
EXIT statement
in general, 4-10
syntax diagram, 13-55
EXIT WHEN statement
in general, 4-10
in basic LOOP statement, 4-10
syntax diagram, 13-55
exiting a loop, 4-9
explicit cursor
in general, 6-8
declaration syntax diagram, 13-57
in package
declaring, 10-11
opening and closing, 10-10
query result processing with
in FOR LOOP statement, 6-24
with OPEN, FETCH, and CLOSE statements, 6-26
explicit format model, 7-17
expression
in general, 2-23
CURSOR
in general, 6-38
passing to pipelined table function, 12-44
data type of, 2-24
in explicit cursor, 6-14
SQL function in PL/SQL, 2-40
static, 2-46
syntax diagram, 13-61
EXTEND collection method, 5-26
external subprogram, 8-49

F

FETCH statement
across COMMIT, 6-48
record variable and, 5-48
syntax diagram, 13-71
that returns no row, 6-10
with BULK COLLECT clause, 12-31
with cursor variable, 6-30
with explicit cursor, 6-10
field of record, 5-1
FIRST collection method, 5-28
FOR LOOP statement
in general, 4-13
bounds of, 4-17
FORALL statement and, 12-11
STEP clause and, 4-14
syntax diagram, 13-74
See also cursor FOR LOOP statement
FOR UPDATE cursor
in general, 6-47
after COMMIT or ROLLBACK, 6-48
FORALL statement
in general, 12-11
associative array in, 13-78
bulk binding and, 12-11
BULK COLLECT clause and, 12-36
for sparse collection
in general, 12-14
SQL%BULK_EXCEPTIONS and, 12-21
handling exception raised in
after FORALL completes, 12-19
immediately, 12-18
number of rows affected by, 12-21
syntax diagram, 13-77
unhandled exception in, 12-17
format model, 7-17
forward declaration of subprogram, 8-8
function
aggregate
See aggregate function
built-in
See SQL function
declaration syntax diagram, 13-83
error-reporting
SQLCODE, 13-131
SQLERRM, 12-20
invoking
in general, 8-3
in SQL statement, 8-43
options for, 8-5
SQL
See SQL function
structure of, 8-4
table
See table function
See also subprogram
function result cache, 8-31
function specification, 13-83

G

global identifier, 2-16
GOTO statement
in general, 4-21

restrictions on, 13-87
syntax diagram, 13-87

H

hardware arithmetic, 12-9
hash table, 5-4
hiding PL/SQL source text
See wrapping PL/SQL source text
host variable
bulk-binding, 12-38
cursor variable as, 6-37
packages and, 10-3

I

identifier
in general, 2-4
ambiguous reference to, B-1
escaped, B-6
global, 2-16
in static SQL, 6-2
local, 2-16
reference to, 2-15
scope of, 2-16
user-defined
in general, 2-5
collecting data about, 12-49
visibility of, 2-16
See also name
IDL, C-1
IF statement
in general, 4-1
IF THEN form, 4-2
IF THEN ELSE form, 4-3
IF THEN ELSIF form
in general, 4-5
nested IF THEN ELSE statement and, 4-6
simple CASE statement and, 4-6
nested, 4-4
syntax diagram, 13-89
imp and triggers, 9-43
implicit cursor
in general, 6-5
CURSOR expression with, 6-38
declaration syntax, 13-91
dynamic SQL and, 7-6
query result processing with
with cursor FOR LOOP statement, 6-24
with SELECT INTO statement, 6-23
implicit data type conversion
minimizing, 12-9
of subprogram parameter, 8-11
of subtypes
constrained, 3-13
unconstrained, 3-12
with base types in same family, 3-14
implicit ROLLBACK statement, 6-44
Import and triggers, 9-43
IN operator, 2-36
IN OUT parameter mode, 8-12
IN parameter mode, 8-12
in-bind, 12-11
independent transaction
See autonomous transaction
index collection
in general, 12-14
representing subset with, 12-14
index-by table
See associative array
infinite loop, 4-10
INFORMATIONAL compile-time warning, 11-2
initial value
%TYPE attribute and, 2-15
NOT NULL constraint and, 2-14
of constant, 2-13
of variable
associative array, 5-4
nested table, 5-10
record, 5-39
scalar, 2-13
varray, 5-8
See also default value
initialization parameter, 1-11
INLINE pragma
in general, 12-2
syntax diagram, 13-94
inner capture
in general, B-7
avoiding, B-7
input, 1-6
inquiry directive, 2-43
INSERT statement
inserting record with
in general, 5-50
restrictions on, 5-52
PL/SQL extension to, 13-96
See also DML statement
INSERTING conditional predicate, 9-4
instance method, IR, 8-48
INSTEAD OF trigger
in general, 9-10
compound, 9-15
for pipelined table function result, 12-46
on nested table column, 9-12
Interface Definition Language (IDL), C-1
internally defined exception
in general, 11-9
giving name to, 11-9
raising explicitly, 11-14
interpreted compilation, 12-53
INVALID_CURSOR exception, 11-10
INVALID_NUMBER exception, 11-10
invoker's rights unit
See IR unit
IR unit
Abstract Data Type (ADT), 8-47
call stack and, 8-44
database link, 8-46
dynamic SQL and, 8-44

instance method, 8-48
invoked by trigger, 8-46
invoked by view, 8-46
name resolution and, 8-44
overriding name resolution in, 8-46
privilege checking and, 8-44
static SQL and, 8-44
template objects in, 8-46
See also AUTHID property
IS [NOT] NULL operator
in general, 2-34
collections and, 5-19
isolation level of transaction, 6-51

J

Java class method invocation, 8-49

K

key-value pair
See associative array
keywords
in general, 2-5
list of, D-1

L

labeled LOOP statement, 4-11
LAST collection method, 5-28
LEVEL pseudocolumn, 6-3
lexical unit, 2-3
library
creating, 14-41
dropping, 14-92
explicitly recompiling, 14-6
library arithmetic, 12-9
LIKE operator, 2-34
LIMIT clause, 12-34
LIMIT collection method, 5-34
literal, 2-8
local identifier, 2-16
locator variable, 11-7
lock mode, 6-46
LOCK TABLE statement, 6-46
locking
overriding default, 6-46
result set row, 6-46
table, 6-46
logical operator, 2-27
logical value, 3-7
LOGIN_DENIED exception, 11-10
LONG data type
in general, 3-6
in trigger, 9-37
LONG RAW data type
in general, 3-6
in trigger, 9-37
LOOP statement
exiting, 4-9
kinds of, 4-9
labeled, 4-9, 4-11
nested, 4-11
optimizing, 12-8
LOOP UNTIL structure, 4-20

M

MALFORMED_WRAP_INPUT exception, A-8
manageability, 1-3
membership test, 2-36
Method 4, 7-7
method, collection
See collection method
mixed parameter notation, 8-21
mode
lock, 6-46
subprogram parameter, 8-12
multibyte character set
as database character set, 2-2
variables for values from, 3-4
multidimensional collection, 5-17
multiline comment, 2-10
multiple data transformations, 12-38
multiset condition, 5-20
mutating table, 9-37
mutating-table error
for function, 8-43
for trigger, 9-37

N

name
qualified
See dot notation
qualified remote, 2-16
remote, 2-16
simple, 2-15
See also identifier
name resolution
in general, B-1
AUTHID property and, 8-44
in static SQL, B-5
overriding in IR unit, 8-46
PL/SQL and SQL differences, B-5
named cursor
in general, 6-8
pipelined table function and, 12-43
See also explicit cursor *and* cursor variable
named parameter notation, 8-21
national character set, 2-3
native dynamic SQL
in general, 7-2
switching to DBMS_SQL package from, 7-7
native execution, compilation for, 12-51
NATURAL subtype, 3-9
NATURALN subtype, 3-9
nested comment, 2-10
nested cursor, 6-38
nested IF statement
in general, 4-4

IF THEN ELSIF form and, 4-6
nested LOOP statement, 4-11
nested record
assignment example, 5-46
declaration example, 5-40
nested subprogram
in general, 8-2
declaration and definition of, 8-2
forward declaration for, 8-8
nested table
in general, 5-10
assigning null value to, 5-15
assigning set operation result to, 5-16
characteristics of, 5-3
column in view, trigger on, 9-12
comparing to NULL, 5-19
comparing two, 5-19
correlation names and, 9-5
COUNT method for, 5-33
FIRST and LAST methods for, 5-31
returned by function, 12-39
SQL multiset conditions and, 5-20
See also collection
nested transaction, 6-50
NEW correlation name
in general, 9-5
with LONG or LONG RAW column, 9-37
new features, xxxi
NEXT collection method, 5-35
NEXTVAL pseudocolumn, 6-3
NLS parameters
associative array and, 5-6
character comparison and, 2-33
SQL injection and, 7-13
NO_DATA_FOUND exception, 11-10
NO_DATA_NEEDED exception
in general, 12-46
SQLCODE for, 11-10
NOCOPY hint
in general, 13-81
subprogram parameter aliasing and, 8-16
tuning subprogram invocation with, 12-7
nonpadded comparison semantics, 3-6
no-op (no operation) statement, 4-23
NOT NULL constraint
in general, 2-13
%TYPE attribute and, 2-15
EXTEND method and, 5-27
NOT operator, 2-29
NOT_LOGGED_ON exception, 11-10
null collection
in general, 5-3
assigning to collection variable, 5-15
collection method and, 5-21
NULL statement
syntax diagram, 13-100
uses for, 4-23
null string, 2-9
NULL value
comparing to collection
associative array, 5-18
nested table, 5-19
varray, 5-19
comparison operator and, 2-32
concatenation operator and, 2-24
for $$PLSQL_UNIT inquiry directive, 2-44
for collection variable, 5-15
for subprogram parameter, 8-18
for unresolvable inquiry directive, 2-46
in control statement, 2-30
IN operator and, 2-36
in set, 2-36
simple CASE expression and, 2-38
simple CASE statement and, 4-7
NUMBER data type family
inefficiency of, 12-9
members of, E-2

O

obfuscating PL/SQL source text
See wrapping PL/SQL source text
object type
See Abstract Data Type (ADT)
OBJECT_VALUE pseudocolumn, 9-9
OCI
associative array and, 5-7
cursor variable and, 6-38
OLD correlation name, 9-5
OPEN FOR statement
in general, 13-103
recursive invocation and, 8-31
OPEN statement
in general, 13-101
recursive invocation and, 8-31
operation, 2-24
operator
comparison
in general, 2-32
cursor variable and, 13-42
logical, 2-27
relational
in general, 2-32
collection and, 5-18
operator precedence, 2-24
optimizer
PL/SQL, 12-1
SQL, 12-43
OR operator, 2-28
Oracle Call Interface (OCI)
associative array and, 5-7
cursor variable and, 6-38
Oracle RAC environment, result caches in, 8-41
ORA-*n* error
See internally defined exception
ordinary user-defined identifier, 2-5
Original Import and triggers, 9-43
OUT parameter mode, 8-12
out-bind, 12-11
outer capture, B-6

output, 1-6
overloaded subprogram
in general, 8-25
INLINE pragma and, 12-2

P

package
in general, 10-1
as application, 10-1
body of
See package body
DBMS_STANDARD, 2-16
explicitly recompiling, 14-8
features of, 10-2
guidelines for writing, 10-11
initialization of, 10-7
of static constants, 2-48
private items in, 10-6
product-specific, 10-1
public items in
See public package item
reasons to use, 10-2
SERIALLY_REUSABLE, 10-7
specification of
See package specification
STANDARD
See STANDARD package
state of, 10-7
supplied by Oracle, 10-1
wrapping
in general, A-1
guideline for, A-2
package body
in general, 10-1
creating, 14-46
dropping, 14-93
initialization part of
in general, 10-6
assigning initial values in, 10-11
replacing, 14-46
package specification
in general, 10-1
creating, 14-43
cursor variable in, 13-42
dropping, 14-93
replacing, 14-43
See also public package item
package subprogram, 8-2
parallel DML
bulk binding and, 12-11
for large table, 12-48
PARALLEL_ENABLE option
in general, 13-85
for pipelined table function, 12-40
for table function, 12-39
parameter
compilation
See compilation parameter
explicit cursor, 6-15
initialization, 1-11
subprogram
See subprogram parameter
parameter mode, 8-12
PARENT correlation name
in general, 9-5
with LONG or LONG RAW column, 9-37
parentheses
nested, 2-25
to control evaluation order, 2-25
to improve readability, 2-25
pattern matching, 2-34
percent sign (%) wildcard character, 2-34
PERFORMANCE compile-time warning, 11-2
PIPE ROW statement, 12-41
PIPELINED option
in general, 12-39
where to specify, 12-40
pipelined table function
in general, 12-39
as aggregate function, 12-45
in autonomous transaction, 13-106
See also table function
placeholder for bind argument
in conditional compilation directive, 2-52
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
in trigger body, 9-5
PLS_INTEGER data type
in general, 3-8
tuning code and, 12-9
PLS_INTEGER static expression, 2-46
PL/Scope tool, 12-49
PL/SQL architecture, 1-10
PL/SQL block syntax, 13-9
PL/SQL engine, 1-10
PL/SQL function result cache, 8-31
PL/SQL language
advantages of, 1-1
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, xxxi
portability of, 1-3
program limits of, C-1
scalability of, 1-3
SQL integration in, 1-1
syntax and semantics, 13-1
PL/SQL optimizer, 12-1
PL/SQL table
See associative array
PL/SQL unit
in general, 1-10

stored
See stored PL/SQL unit
PL/SQL Wrapper utility, A-2
PLSQL_CCFLAGS compilation parameter, 2-45
PLSQL_OPTIMIZE_LEVEL compilation parameter, 12-1
PLSQL_WARNINGS compilation parameter
displaying value of
with ALL_PLSQL_OBJECT_SETTINGS view, 11-3
with DBMS_WARNING subprogram, 11-3
setting value of
with ALTER statements, 11-2
with PLSQL_WARNINGS subprogram, 11-3
portability, 1-3
positional parameter notation, 8-21
POSITIVE subtype, 3-9
POSITIVEN subtype, 3-9
post-processed source text, 2-51
pragma
in general, 2-41
AUTONOMOUS_TRANSACTION
in general, 13-6
for pipelined table function, 12-40
EXCEPTION_INIT, 13-46
INLINE
in general, 12-2
syntax diagram, 13-94
RESTRICT_REFERENCES, 13-114
SERIALLY_REUSABLE, 13-130
precedence, operator, 2-24
predefined constant, 3-2
predefined data type, E-1
predefined exception
in general, 11-10
raising explicitly, 11-14
redeclared, 11-12
predefined inquiry directive, 2-44
predefined subtype, E-1
preprocessor control token, 2-42
PRIOR collection method, 5-35
privilege checking and AUTHID property, 8-44
procedure
declaration syntax, 13-108
invoking, 8-3
structure of, 8-4
See also subprogram
procedure specification, 13-108
product-specific package, 10-1
Profiler API, 12-50
profiling and tracing programs, 12-49
program limits, C-1
PROGRAM_ERROR exception, 11-10
pseudocolumn, 6-3
pseudoinstruction
See pragma
pseudorecord, 9-5
See also correlation name
public package item
appropriate, 10-3
collection type as, 5-37
cursor variable as, 13-42
declaring, 10-3
RECORD type as, 5-39, 5-41
referencing, 10-3
remote variable, 10-4
scope of, 10-3
visibility of, 10-3
publishing events, 9-43
purity rules for subprograms, 8-43

Q

qualified name
See dot notation
qualified remote name, 2-16
query
invoking function in, 12-5
processing result set of
in general, 6-22
multiple-row dynamic query, 7-4
See also SELECT INTO statement
quotation mark, single or double, 2-2
quoted user-defined identifier, 2-6

R

RAISE statement
in general, 11-13
syntax diagram, 13-110
RAISE_APPLICATION_ERROR procedure, 11-16
raising exception explicitly, 11-13
range test, 2-35
read-only transaction, 6-45
read-write transaction, 6-45
recompiling stored PL/SQL unit, 14-1
record
in general, 5-1
as public package item, 5-41
assigning value to, 5-44
creating
in general, 5-38
syntax diagram, 13-111
nested
See nested record
representing row, 5-42
types of, 5-39
recursive subprogram
in general, 8-29
result-cached, 8-36
recursive trigger, 9-35
REF CURSOR
See cursor variable
REF CURSOR type, 6-28
relational operator
in general, 2-32
collection and, 5-18
RELEASE constant, 2-49
remote exception handling
subprograms and, 11-19

triggers and, 9-34
remote name, 2-16
remote public package variable, 10-4
REPEAT UNTIL structure, 4-20
replacing stored PL/SQL unit, 14-1
reraising exception, 11-15
reserved preprocessor control token, 2-42
reserved words
information about, 2-5
list of, D-1
RESTRICT_REFERENCES pragma, 13-114
result cache, 8-31
RESULT_CACHE clause, 8-32
RESULT_CACHE option for function, 13-85
RETURN clause of function, 8-4
RETURN INTO clause
See RETURNING INTO clause
RETURN statement, 8-5
RETURNING INTO clause
in general, 13-118
BULK COLLECT clause of
in general, 12-35
FORALL statement and, 12-36
REUSE SETTINGS clause, 1-12
ROLLBACK statement
in general, 6-41
FOR UPDATE cursor and, 6-48
implicit, 6-44
in autonomous transaction, 6-53
transparent, 9-35
rowid, 3-7
ROWID data type, 3-7
ROWID pseudocolumn
in general, 6-3
instead of CURRENT OF clause, 6-48
row-level trigger, 9-3
ROWNUM pseudocolumn
bulk SELECT operation and, 12-30
single-row result set and, 6-23
ROWTYPE_MISMATCH exception, 11-10
run-time error
See exception

S

same-scope capture, B-6
SAMPLE clause, 12-30
SAVEPOINT statement
in general, 6-43
in autonomous transaction, 6-53
scalability
SERIALLY_REUSABLE packages and, 10-7
subprograms and, 1-3
scalar data type, 3-1
scalar variable
assigning value to, 2-20
declaration
in general, 2-12
syntax diagram, 13-123
initial value of, 2-13
SCHEMA trigger, 9-31
scope of identifier, 2-16
searched CASE expression, 2-39
searched CASE statement
in general, 4-7
syntax diagram, 13-20
security mechanism
against SQL injection, 7-9
PL/SQL source text wrapping
benefit of, A-1
limitations of, A-2
trigger as, 9-3
SELECT FOR UPDATE statement, 6-46
SELECT INTO statement
assigning values with
to record variable, 5-47
to scalar variables, 2-22
avoiding inner capture in, B-7
query result set processing with, 6-23
SQL%NOTFOUND attribute and, 6-7
SQL%ROWCOUNT attribute and, 6-8
syntax diagram, 13-125
with BULK COLLECT clause, 12-24
See also query
selection directive, 2-42
selector
in simple CASE expression, 2-38
in simple CASE statement, 4-6
SELF_IS_NULL exception, 11-10
sequence, 6-3
sequential control statement, 4-20
SERIALLY_REUSABLE package, 10-7
SERIALLY_REUSABLE pragma, 13-130
session cursor, 6-5
set data structure, 5-4
set membership test, 2-36
SET ROLE command and AUTHID property, 8-45
SET TRANSACTION statement, 6-45
SEVERE compile-time warning, 11-2
short-circuit evaluation
how it works, 2-31
tuning code and, 12-10
side effects of subprogram, 8-31
SIGNTYPE subtype, 3-9
simple CASE expression, 2-38
simple CASE statement
in general, 4-6
IF THEN ELSIF statement and, 4-6
syntax diagram, 13-20
simple DML trigger, 9-3
simple name, 2-15
SIMPLE_DOUBLE subtype
in general, 3-3
tuning code and, 12-9
SIMPLE_FLOAT subtype
in general, 3-3
tuning code and, 12-9
SIMPLE_INTEGER subtype
in general, 3-10
tuning code and, 12-9

single quotation mark (’), 2-2
single-line comment, 2-9
sparse collection
in general, 5-3
FORALL statement for, 12-14
SQL%BULK_EXCEPTIONS and, 12-21
traversing, 5-35
specification
cursor, 13-57
function, 13-83
package
See package specification
procedure, 13-108
SQL
bulk
in general, 12-11
in compound DML trigger, 9-16
dynamic
See dynamic SQL
static
See static SQL
SQL cursor
See implicit cursor
SQL data type, 3-2
SQL function
in PL/SQL expression, 2-40
tuning and, 12-10
SQL injection, 7-9
SQL integration in PL/SQL, 1-1
SQL multiset condition, 5-20
SQL MULTISET operator, 5-16
SQL optimizer, 12-43
SQL statement
for stored PL/SQL unit, 14-1
in trigger, 9-2
invoking collection method in, 5-22
invoking PL/SQL function in, 8-43
tuning, 12-5
See also anonymous block
SQL%BULK_EXCEPTIONS cursor attribute, 12-19
SQL%BULK_ROWCOUNT cursor attribute, 12-21
SQL%FOUND cursor attribute, 6-6
SQL%NOTFOUND cursor attribute, 6-7
SQL%ROWCOUNT cursor attribute, 6-7
SQL*Loader and triggers, 9-42
SQLCODE function, 13-131
SQLERRM function
in general, 13-132
SQL%BULK_EXCEPTIONS and, 12-20
SQLJ object type, creating, 14-68
standalone stored subprogram
in general, 8-2
function
creating, 14-32
dropping, 14-90
explicitly recompiling, 14-3
replacing, 14-32
procedure
creating, 14-50
dropping, 14-95
explicitly recompiling, 14-11
replacing, 14-50
STANDARD package
data type defined in
See predefined data type
exception defined in
See predefined exception
how it defines PL/SQL environment, 10-16
listing identifiers defined in, 2-5
referencing item defined in, 2-16
statement injection, 7-11
statement modification, 7-10
statement-level trigger, 9-3
static constant
in general, 2-48
in DBMS_DB_VERSION package, 2-49
static expression, 2-46
static SQL
in general, 6-1
AUTHID property and, 8-44
name resolution in, B-5
placeholder for bind argument in
in general, 6-2
OPEN FOR statement and, 6-30
PL/SQL identifier in, 6-2
STORAGE_ERROR exception
in general, 11-10
recursive invocation and, 8-30
store table, 5-13
stored PL/SQL unit
in general, 1-12
creating, 14-1
recompiling, 14-1
replacing, 14-1
wrapping, A-1
stored subprogram
in general, 8-2
unhandled exception in, 11-24
wrapping, A-1
string
null, 2-9
zero-length, 2-9
See also character literal
STRING subtype, 3-5
strong REF CURSOR type
creating, 6-29
FETCH statement and, 6-30
subprogram
in general, 8-1
inlining, 12-1
invoked by trigger, 9-32
unhandled exception in, 11-24
subprogram invocation
optimization of, 12-1
resolution of, 8-23
syntax of, 8-2
tuning, 12-7
subprogram parameter
in general, 8-9
collection as, 5-22

CURSOR expression as actual, 6-38
cursor variable as, 6-35
optional, 8-18
pseudorecord as, 9-6
query result as, 6-35
required, 8-18
subquery
correlated, 6-27
result set processing with, 6-26
subscript of collection, 5-1
SUBSCRIPT_BEYOND_COUNT exception, 11-10
SUBSCRIPT_OUTSIDE_LIMIT exception, 11-10
subtype
in general, 3-1
constrained
in general, 3-12
subprogram parameter and, 8-10
of BINARY_DOUBLE data type, 3-3
of BINARY_FLOAT data type, 3-3
of PLS_INTEGER data type, 3-9
predefined, E-1
unconstrained, 3-12
user-defined, 3-11
See also data type
synonym, 2-16
SYS_INVALID_ROWID exception, 11-10
SYS_REFCURSOR type, 6-28
system trigger, 9-30

T

table
hash, 5-4
index-by
See associative array
mutating, 9-37
nested
in general, 5-10
characteristics of, 5-3
parallel DML for large, 12-48
PL/SQL
See associative array
store, 5-13
unordered, 5-4
updating large in parallel, 12-48
table alias
for avoiding inner capture, B-7
for row expression, B-9
for table element attribute or method, B-8
table function
in general, 12-39
pipelined
See pipelined table function
weak cursor variable argument to, 6-29
TCL statement
in general, 6-2
in subprogram invoked by trigger, 9-32
in trigger, 6-53
TIMEOUT_ON_RESOURCE exception, 11-10
timing point
of compound DML trigger, 9-15
of simple DML trigger, 9-3
of system trigger, 9-30
trigger firing order and, 9-40
TO_NUMBER function, 7-8
TO_REFCURSOR function, 7-7
TOO_MANY_ROWS exception, 11-10
Trace API, 12-50
trace file, 11-1
tracing and profiling programs, 12-49
transaction
autonomous
in general, 6-49
pipelined table function in, 13-106
ending
with COMMIT statement, 6-40
with ROLLBACK statement, 6-41
nested, 6-50
read-only, 6-45
read-write, 6-45
retrying after handling exception, 11-27
SQL%ROWCOUNT cusor attribute and, 6-8
Transaction Control Language
See TCL statement
TRANSACTIONS initialization parameter, 6-53
trigger
in general, 9-1
as security mechanism, 9-3
AUTHID property and, 8-44
autonomous, 6-53
cascading, 9-41
DDL statement in, 6-53
hiding implementation details of, A-2
IR subprogram invoked by, 8-46
recursive, 9-35
TCL statement in, 6-53
TRIM collection method, 5-25
tri-state logic, 2-27
tuning PL/SQL code, 12-1
type
See data type
type-compatible data type
for collection variables, 5-14
for scalar variables, 2-20

U

unbounded collection, 5-3
unconstrained subtype, 3-12
underscore (_) wildcard character, 2-34
unhandled exception
in general, 11-24
in FORALL statement, 12-17
OUT and IN OUT parameters and, 8-13
unordered table, 5-4
UPDATE statement
BEFORE statement trigger and, 9-35
PL/SQL extensions to, 13-136
with values in record
in general, 5-51

restrictions on, 5-52
See also DML statement
UPDATING conditional predicate, 9-4
UROWID data type, 3-7
USER_DUMP_DEST initialization parameter, 11-1
user-defined exception
in general, 11-11
giving error code to, 11-16
raising
with RAISE statement, 11-13
with RAISE_APPLICATION_ERROR procedure, 11-16
user-defined identifier
in general, 2-5
collecting data about, 12-49
user-defined subtype, 3-11
user-defined type
See Abstract Data Type (ADT)
utlrp.sql script, 12-55

V

V$RESERVED_WORDS view, D-1
validation check for avoiding SQL injection, 7-16
VALUE_ERROR exception, 11-10
VARCHAR subtype, 3-5
VARCHAR2 data type, 3-3
VARCHAR2 static expression, 2-47
variable
binding of, 12-11
BOOLEAN, 2-23
collection
See collection
composite, 5-1
cursor
See cursor variable
host
cursor variable as, 6-37
packages and, 10-3
in cursor variable query, 6-33
in explicit cursor query, 6-12
locator, 11-7
record
See record
remote public package, 10-4
scalar
See scalar variable
with undefined value, 6-3
variable-size array
See varray
varray
in general, 5-8
assigning null value to, 5-15
characteristics of, 5-3
comparing to NULL, 5-19
COUNT method for, 5-33
FIRST and LAST methods for, 5-30
returned by function, 12-39
See also collection
VERSION constant, 2-49
view
AUTHID property and, 8-44
IR subprogram invoked by, 8-46
virtual column
See calculated column
visibility
of identifier, 2-16
of transaction, 6-50

W

warning, compile-time, 11-2
weak REF CURSOR type
creating, 6-29
FETCH statement and, 6-30
WHILE LOOP statement
in general, 4-20
syntax diagram, 13-138
whitespace character
between lexical units, 2-11
in character literal, 2-9
in database character set, 2-2
wildcard character, 2-34
WRAP function, A-8
wrap utility
See PL/SQL Wrapper utility
wrapping PL/SQL source text
in general, A-1
inquiry directives and, 2-46

Z

ZERO_DIVIDE exception, 11-11
zero-length string, 2-9