

Oracle® Database
PL/SQL Language Reference
11g Release 1 (11.1)
B28370-01

July 2007

Oracle Database PL/SQL Language Reference, 11g Release 1 (11.1)

B28370-01

Copyright © 1996, 2007, Oracle. All rights reserved.

Primary Author: Sheila Moore

Contributing Author: E. Belden

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

The Programs (which include both the software and documentation) contain proprietary information; they are provided under a license agreement containing restrictions on use and disclosure and are also protected by copyright, patent, and other intellectual and industrial property laws. Reverse engineering, disassembly, or decompilation of the Programs, except to the extent required to obtain interoperability with other independently created software or as specified by law, is prohibited.

The information contained in this document is subject to change without notice. If you find any problems in the documentation, please report them to us in writing. This document is not warranted to be error-free. Except as may be expressly permitted in your license agreement for these Programs, no part of these Programs may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose.

If the Programs are delivered to the United States Government or anyone licensing or using the Programs on behalf of the United States 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, use, duplication, disclosure, modification, and adaptation of the Programs, including documentation and technical data, shall be subject to the licensing restrictions set forth in the applicable Oracle license agreement, and, to the extent applicable, the additional rights set forth in FAR 52.227-19, Commercial Computer Software—Restricted Rights (June 1987). Oracle USA, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

The Programs are not intended for use in any nuclear, aviation, mass transit, medical, or other inherently dangerous applications. It shall be the licensee's responsibility to take all appropriate fail-safe, backup, redundancy and other measures to ensure the safe use of such applications if the Programs are used for such purposes, and we disclaim liability for any damages caused by such use of the Programs.

Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.

The Programs may provide links to Web sites and access to content, products, and services from third parties. Oracle is not responsible for the availability of, or any content provided on, third-party Web sites. You bear all risks associated with the use of such content. If you choose to purchase any products or services from a third party, the relationship is directly between you and the third party. Oracle is not responsible for: (a) the quality of third-party products or services; or (b) fulfilling any of the terms of the agreement with the third party, including delivery of products or services and warranty obligations related to purchased products or services. Oracle is not responsible for any loss or damage of any sort that you may incur from dealing with any third party.

Contents

Preface	xix
Audience	xix
Documentation Accessibility	xix
Related Documents	xx
Conventions	xx
Syntax Descriptions.....	xxi
What's New in PL/SQL?	xxiii
New PL/SQL Features for 11g Release 1 (11.1)	xxiii
1 Overview of PL/SQL	
Advantages of PL/SQL	1-1
Tight Integration with SQL.....	1-1
Better Performance.....	1-2
Higher Productivity	1-3
Full Portability	1-3
Tight Security.....	1-3
Access to Predefined Packages.....	1-3
Support for Object-Oriented Programming	1-4
Support for Developing Web Applications and Pages.....	1-4
Understanding the Main Features of PL/SQL	1-4
Understanding PL/SQL Block Structure.....	1-5
Understanding PL/SQL Variables and Constants.....	1-5
Declaring Variables.....	1-5
Assigning Values to Variables	1-6
Bind Variables	1-7
Declaring Constants.....	1-8
Processing Queries with PL/SQL.....	1-8
Declaring PL/SQL Subprograms.....	1-8
Declaring Datatypes for PL/SQL Variables.....	1-8
%TYPE	1-9
%ROWTYPE	1-9
Understanding PL/SQL Control Structures	1-10
Conditional Control.....	1-10
Iterative Control	1-11

Sequential Control	1-12
Understanding Conditional Compilation	1-13
Writing Reusable PL/SQL Code.....	1-13
Subprograms (Procedures and Functions).....	1-13
Packages (APIs Written in PL/SQL).....	1-14
Inputting and Outputting Data with PL/SQL.....	1-16
Understanding PL/SQL Data Abstraction.....	1-17
Cursors	1-17
Collections.....	1-17
Records	1-18
Object Types	1-18
Understanding PL/SQL Error Handling	1-19
PL/SQL Architecture	1-20
PL/SQL Architecture in Oracle Database Server.....	1-20
Anonymous Blocks.....	1-21
Stored Subprograms.....	1-21
Database Triggers	1-22
PL/SQL Architecture in Oracle Tools.....	1-22

2 PL/SQL Language Fundamentals

Character Sets and Lexical Units	2-1
Delimiters	2-2
Identifiers	2-3
Reserved Words	2-4
Predefined Identifiers.....	2-4
Quoted Identifiers.....	2-4
Literals	2-5
Numeric Literals.....	2-5
Character Literals.....	2-6
String Literals.....	2-6
BOOLEAN Literals	2-7
Date and Time Literals.....	2-7
Comments	2-8
Single-Line Comments.....	2-8
Multiline Comments.....	2-8
Restrictions on Comments.....	2-9
Declarations	2-9
Constants.....	2-10
Using DEFAULT	2-10
Using NOT NULL.....	2-10
Using the %TYPE Attribute.....	2-10
Using the %ROWTYPE Attribute.....	2-12
Aggregate Assignment.....	2-13
Using Aliases	2-13
Restrictions on Declarations	2-14
Naming Conventions	2-14
Synonyms	2-14

Scoping	2-15
Case Sensitivity.....	2-15
Name Resolution.....	2-15
Scope and Visibility of PL/SQL Identifiers	2-16
Assigning Values to Variables	2-19
Assigning BOOLEAN Values.....	2-20
Assigning SQL Query Results to PL/SQL Variables.....	2-20
PL/SQL Expressions and Comparisons.....	2-21
Operator Precedence.....	2-21
Logical Operators.....	2-22
Order of Evaluation	2-23
Short-Circuit Evaluation	2-23
Comparison Operators.....	2-24
Relational Operators.....	2-24
IS NULL Operator.....	2-24
LIKE Operator	2-25
BETWEEN Operator.....	2-25
IN Operator.....	2-25
Concatenation Operator.....	2-25
BOOLEAN Expressions	2-26
BOOLEAN Arithmetic Expressions	2-26
BOOLEAN Character Expressions.....	2-26
BOOLEAN Date Expressions.....	2-27
Guidelines for BOOLEAN Expressions.....	2-27
CASE Expressions	2-28
Simple CASE Expression	2-28
Searched CASE Expression	2-29
Handling Null Values in Comparisons and Conditional Statements	2-30
NULL Values and the NOT Operator.....	2-31
NULL Values and Zero-Length Strings.....	2-31
NULL Values and the Concatenation Operator.....	2-31
NULL Values as Arguments to Built-In Functions.....	2-31
PL/SQL Error-Reporting Functions.....	2-33
Using SQL Functions in PL/SQL.....	2-33
Conditional Compilation.....	2-34
How Does Conditional Compilation Work?.....	2-34
Conditional Compilation Control Tokens.....	2-34
Using Conditional Compilation Selection Directives.....	2-35
Using Conditional Compilation Error Directives	2-35
Using Conditional Compilation Inquiry Directives	2-35
Using Predefined Inquiry Directives with Conditional Compilation	2-36
Using Static Expressions with Conditional Compilation.....	2-36
Boolean Static Expressions	2-37
PLS_INTEGER Static Expressions.....	2-37
VARCHAR2 Static Expressions.....	2-37
Static Constants.....	2-38
Setting the PLSQL_CCFLAGS Initialization Parameter.....	2-39

Using DBMS_DB_VERSION Package Constants	2-39
Conditional Compilation Examples	2-39
Using Conditional Compilation to Specify Code for Database Versions	2-40
Using DBMS_PREPROCESSOR Procedures to Print or Retrieve Source Text	2-41
Conditional Compilation Restrictions.....	2-41
Using PL/SQL to Create Web Applications and Server Pages.....	2-42
PL/SQL Web Applications.....	2-42
PL/SQL Server Pages	2-43

3 PL/SQL Datatypes

Predefined PL/SQL Scalar Datatypes and Subtypes.....	3-1
Predefined PL/SQL Numeric Datatypes and Subtypes.....	3-2
PLS_INTEGER and BINARY_INTEGER Datatypes.....	3-2
SIMPLE_INTEGER Subtype of PLS_INTEGER	3-3
Overflow Semantics.....	3-3
Overloading Rules.....	3-4
Integer Literals	3-4
Cast Operations.....	3-5
Compiler Warnings	3-5
BINARY_FLOAT and BINARY_DOUBLE Datatypes.....	3-5
NUMBER Datatype	3-6
Predefined PL/SQL Character Datatypes and Subtypes.....	3-7
CHAR and VARCHAR2 Datatypes	3-8
Predefined Subtypes of Character Datatypes.....	3-9
Memory Allocation for Character Variables.....	3-9
Blank-Padding Shorter Character Values	3-9
Comparing Character Values.....	3-10
Maximum Sizes of Values Inserted into Character Database Columns	3-11
RAW Datatype	3-11
NCHAR and NVARCHAR2 Datatypes.....	3-12
AL16UTF16 and UTF8 Encodings.....	3-12
NCHAR Datatype.....	3-12
NVARCHAR2 Datatype	3-13
LONG and LONG RAW Datatypes	3-13
ROWID and UROWID Datatypes	3-14
Predefined PL/SQL BOOLEAN Datatype.....	3-15
Predefined PL/SQL Datetime and Interval Datatypes	3-15
DATE Datatype	3-16
TIMESTAMP Datatype	3-17
TIMESTAMP WITH TIME ZONE Datatype.....	3-18
TIMESTAMP WITH LOCAL TIME ZONE Datatype.....	3-18
INTERVAL YEAR TO MONTH Datatype	3-19
INTERVAL DAY TO SECOND Datatype	3-19
Datetime and Interval Arithmetic	3-20
Avoiding Truncation Problems Using Date and Time Subtypes	3-20
Predefined PL/SQL Large Object (LOB) Datatypes	3-21
BFILE Datatype.....	3-22

BLOB Datatype.....	3-22
CLOB Datatype.....	3-22
NCLOB Datatype	3-22
User-Defined PL/SQL Subtypes	3-22
Defining Subtypes	3-23
Using Subtypes.....	3-23
Type Compatibility with Subtypes	3-24
Constraints and Default Values with Subtypes.....	3-25
PL/SQL Datatype Conversion	3-26
Explicit Conversion.....	3-26
Implicit Conversion	3-27

4 Using PL/SQL Control Structures

Overview of PL/SQL Control Structures	4-1
Testing Conditions (IF and CASE Statements)	4-2
Using the IF-THEN Statement	4-2
Using the IF-THEN-ELSE Statement.....	4-2
Using the IF-THEN-ELSIF Statement.....	4-3
Using the Simple CASE Statement	4-4
Using the Searched CASE Statement	4-6
Guidelines for IF and CASE Statements	4-6
Controlling Loop Iterations (LOOP, EXIT, and CONTINUE Statements)	4-7
Using the Basic LOOP Statement.....	4-8
Using the EXIT Statement	4-8
Using the EXIT-WHEN Statement.....	4-9
Using the CONTINUE Statement.....	4-9
Using the CONTINUE-WHEN Statement	4-10
Labeling a PL/SQL Loop	4-10
Using the WHILE-LOOP Statement.....	4-11
Using the FOR-LOOP Statement	4-12
How PL/SQL Loops Repeat.....	4-13
Dynamic Ranges for Loop Bounds.....	4-13
Scope of the Loop Counter Variable	4-14
Using the EXIT Statement in a FOR Loop	4-15
Sequential Control (GOTO and NULL Statements)	4-16
Using the GOTO Statement	4-16
GOTO Statement Restrictions	4-17
Using the NULL Statement.....	4-18

5 Using PL/SQL Collections and Records

What Are PL/SQL Collections and Records?	5-1
Understanding PL/SQL Collections	5-2
Understanding PL/SQL Nested Tables.....	5-2
Understanding PL/SQL Varrays.....	5-3
Understanding PL/SQL Associative Arrays (Index-By Tables).....	5-3
How Globalization Settings Affect VARCHAR2 Keys for Associative Arrays	5-4

Understanding PL/SQL Records.....	5-5
Choosing Which PL/SQL Collection Types to Use.....	5-5
Choosing Between Nested Tables and Associative Arrays	5-6
Choosing Between Nested Tables and Varrays.....	5-6
Defining Collection Types.....	5-7
Declaring Collection Variables.....	5-8
Initializing and Referencing Collections	5-10
Referencing Collection Elements.....	5-12
Assigning Collections.....	5-13
Comparing Collections.....	5-17
Using Multilevel Collections	5-19
Using Collection Methods.....	5-20
Checking If a Collection Element Exists (EXISTS Method)	5-21
Counting the Elements in a Collection (COUNT Method).....	5-21
Checking the Maximum Size of a Collection (LIMIT Method)	5-22
Finding the First or Last Collection Element (FIRST and LAST Methods)	5-22
Looping Through Collection Elements (PRIOR and NEXT Methods).....	5-23
Increasing the Size of a Collection (EXTEND Method).....	5-25
Decreasing the Size of a Collection (TRIM Method).....	5-26
Deleting Collection Elements (DELETE Method)	5-27
Applying Methods to Collection Parameters.....	5-28
Avoiding Collection Exceptions	5-28
Defining and Declaring Records.....	5-31
Using Records as Subprogram Parameters and Function Return Values	5-32
Assigning Values to Records.....	5-34
Comparing Records	5-36
Inserting Records Into the Database.....	5-36
Updating the Database with Record Values	5-36
Restrictions on Record Inserts and Updates	5-38
Querying Data Into Collections of Records.....	5-38

6 Using Static SQL

Description of Static SQL	6-1
Data Manipulation Language (DML) Statements	6-1
Transaction Control Language (TCL) Statements.....	6-3
SQL Functions.....	6-3
SQL Pseudocolumns.....	6-3
CURRVAL and NEXTVAL.....	6-4
LEVEL.....	6-5
ROWID	6-5
ROWNUM	6-5
SQL Operators	6-6
Comparison Operators.....	6-6
Set Operators	6-7
Row Operators	6-7
Managing Cursors in PL/SQL.....	6-7
Implicit Cursors.....	6-7

Attributes of Implicit Cursors	6-7
%FOUND Attribute: Has a DML Statement Changed Rows?	6-8
%ISOPEN Attribute: Always FALSE for Implicit Cursors	6-8
%NOTFOUND Attribute: Has a DML Statement Failed to Change Rows?	6-8
%ROWCOUNT Attribute: How Many Rows Affected So Far?	6-8
Guidelines for Using Attributes of Implicit Cursors	6-9
Explicit Cursors	6-9
Declaring a Cursor	6-10
Opening a Cursor	6-10
Fetching with a Cursor	6-11
Fetching Bulk Data with a Cursor	6-12
Closing a Cursor	6-13
Attributes of Explicit Cursors	6-13
%FOUND Attribute: Has a Row Been Fetched?	6-13
%ISOPEN Attribute: Is the Cursor Open?	6-14
%NOTFOUND Attribute: Has a Fetch Failed?	6-14
%ROWCOUNT Attribute: How Many Rows Fetched So Far?	6-15
Querying Data with PL/SQL	6-16
Selecting At Most One Row (SELECT INTO Statement)	6-16
Selecting Multiple Rows (BULK COLLECT Clause)	6-16
Looping Through Multiple Rows (Cursor FOR Loop)	6-17
Performing Complicated Query Processing (Explicit Cursors)	6-17
Cursor FOR LOOP	6-17
Implicit Cursor FOR LOOP	6-17
Explicit Cursor FOR LOOP	6-18
Defining Aliases for Expression Values in a Cursor FOR Loop	6-18
Using Subqueries	6-19
Using Correlated Subqueries	6-20
Writing Maintainable PL/SQL Subqueries	6-21
Using Cursor Variables (REF CURSORS)	6-22
What Are Cursor Variables (REF CURSORS)?	6-22
Why Use Cursor Variables?	6-22
Declaring REF CURSOR Types and Cursor Variables	6-23
Passing Cursor Variables As Parameters	6-24
Controlling Cursor Variables (OPEN-FOR, FETCH, and CLOSE Statements)	6-25
Opening a Cursor Variable	6-25
Using a Cursor Variable as a Host Variable	6-27
Fetching from a Cursor Variable	6-28
Closing a Cursor Variable	6-29
Reducing Network Traffic When Passing Host Cursor Variables to PL/SQL	6-29
Avoiding Errors with Cursor Variables	6-30
Restrictions on Cursor Variables	6-30
Using Cursor Expressions	6-31
Restrictions on Cursor Expressions	6-31
Example: Cursor Expressions	6-32
Constructing REF CURSORS with Cursor Subqueries	6-32
Overview of Transaction Processing in PL/SQL	6-32

Using COMMIT in PL/SQL	6-33
Using ROLLBACK in PL/SQL.....	6-34
Using SAVEPOINT in PL/SQL	6-35
How Oracle Database Does Implicit Rollbacks	6-36
Ending Transactions	6-36
Setting Transaction Properties (SET TRANSACTION Statement)	6-37
Overriding Default Locking	6-37
Using FOR UPDATE	6-38
Using LOCK TABLE.....	6-39
Fetching Across Commits.....	6-39
Doing Independent Units of Work with Autonomous Transactions	6-40
Advantages of Autonomous Transactions	6-41
Defining Autonomous Transactions	6-41
Comparison of Autonomous Transactions and Nested Transactions	6-43
Transaction Context.....	6-43
Transaction Visibility.....	6-43
Controlling Autonomous Transactions	6-44
Entering and Exiting.....	6-44
Committing and Rolling Back.....	6-44
Using Savepoints.....	6-44
Avoiding Errors with Autonomous Transactions.....	6-45
Using Autonomous Triggers.....	6-45
Invoking Autonomous Functions from SQL.....	6-46

7 Using Dynamic SQL

When You Need Dynamic SQL	7-1
Using Native Dynamic SQL.....	7-2
Using the EXECUTE IMMEDIATE Statement.....	7-2
Using the OPEN-FOR, FETCH, and CLOSE Statements	7-4
Repeating 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.....	7-5
Using DBMS_SQL Package	7-6
DBMS_SQL.TO_REFCURSOR Function	7-7
DBMS_SQL.TO_CURSOR_NUMBER Function.....	7-8
Avoiding SQL Injection in PL/SQL	7-9
Overview of SQL Injection Techniques	7-9
Statement Modification.....	7-9
Statement Injection	7-11
Guarding Against SQL Injection.....	7-12
Using Bind Arguments to Guard Against SQL Injection.....	7-12
Using Validation Checks to Guard Against SQL Injection.....	7-13

8 Using PL/SQL Subprograms

What Are PL/SQL Subprograms?.....	8-1
Subprogram Parts	8-1
Subprogram Types.....	8-2

Subprogram Calls.....	8-3
Subprogram Examples	8-3
Why Use PL/SQL Subprograms?	8-4
Using the RETURN Statement	8-5
Declaring Nested PL/SQL Subprograms	8-5
Passing Parameters to PL/SQL Subprograms	8-6
Formal and Actual Subprogram Parameters	8-6
Using Positional, Named, or Mixed Notation for PL/SQL Subprogram Parameters	8-7
Specifying Subprogram Parameter Modes.....	8-9
Using IN Mode.....	8-9
Using OUT Mode.....	8-9
Using IN OUT Mode	8-10
Summary of Subprogram Parameter Modes	8-10
Using Default Values for Subprogram Parameters.....	8-10
Overloading PL/SQL Subprogram Names	8-12
Guidelines for Overloading with Numeric Types	8-13
Restrictions on Overloading	8-14
When Compiler Catches Overloading Errors	8-14
How PL/SQL Subprogram Calls Are Resolved	8-16
Using Invoker's Rights or Definer's Rights (AUTHID Clause)	8-17
Advantages of Invoker's Rights	8-18
Specifying Subprogram Privileges (AUTHID Clause)	8-18
Who Is the Current User During Subprogram Execution?	8-19
How External References Are Resolved in IR Subprograms.....	8-19
Need for Template Objects in IR Subprograms	8-21
Overriding Default Name Resolution in IR Subprograms	8-21
Granting Privileges to IR Subprograms.....	8-21
Using Views and Database Triggers with IR Subprograms	8-22
Using Database Links with IR Subprograms.....	8-22
Using Object Types with IR Subprograms	8-23
Invoking IR Instance Methods	8-24
Using Recursive PL/SQL Subprograms	8-24
Invoking External Subprograms	8-25
Controlling Side Effects of PL/SQL Subprograms	8-26
Understanding PL/SQL Subprogram Parameter Aliasing	8-27
Using the Cross-Session PL/SQL Function Result Cache	8-29
Enabling Result-Caching for a Function.....	8-30
Developing Applications with Result-Cached Functions	8-31
Restrictions on Result-Cached Functions	8-31
Examples of Result-Cached Functions.....	8-32
Result-Cached Application Configuration Parameters.....	8-32
Result-Cached Recursive Function.....	8-34
Advanced Result-Cached Function Topics	8-34
Rules for a Cache Hit.....	8-34
Bypassing the Result Cache.....	8-35
Making Result-Cached Functions Handle Session-Specific Settings	8-35
Making Result-Cached Functions Handle Session-Specific Application Contexts.....	8-36

Choosing Result-Caching Granularity.....	8-37
Result Caches in Oracle RAC Environment.....	8-38
Managing the Result Cache.....	8-38
Hot-Patching PL/SQL Program Units on Which Result-Cached Functions Depend ...	8-39

9 Using Triggers

Overview of Triggers	9-1
Guidelines for Designing Triggers	9-2
Privileges Required to Use Triggers	9-3
Creating Triggers	9-3
Naming Triggers	9-4
When Does the Trigger Fire?	9-5
Do Import and SQL*Loader Fire Triggers?.....	9-5
How Column Lists Affect UPDATE Triggers	9-5
Controlling When a Trigger Fires (BEFORE and AFTER Options)	9-6
Ordering of Triggers	9-6
Modifying Complex Views (INSTEAD OF Triggers).....	9-7
Views that Require INSTEAD OF Triggers.....	9-8
Triggers on Nested Table View Columns	9-8
Example: INSTEAD OF Trigger.....	9-10
Firing Triggers One or Many Times (FOR EACH ROW Option)	9-11
Firing Triggers Based on Conditions (WHEN Clause)	9-12
Compound Triggers.....	9-12
Why Use Compound Triggers?	9-12
Compound Trigger Sections.....	9-13
Triggering Statements of Compound Triggers.....	9-14
Compound Trigger Restrictions	9-14
Compound Trigger Example	9-15
Using Compound Triggers to Avoid Mutating-Table Error	9-16
Coding the Trigger Body	9-17
Accessing Column Values in Row Triggers	9-19
Example: Modifying LOB Columns with a Trigger.....	9-19
INSTEAD OF Triggers on Nested Table View Columns	9-20
Avoiding Trigger Name Conflicts (REFERENCING Option)	9-20
Detecting the DML Operation that Fired a Trigger	9-21
Error Conditions and Exceptions in the Trigger Body	9-21
Triggers on Object Tables.....	9-21
Triggers and Handling Remote Exceptions	9-22
Restrictions on Creating Triggers	9-23
Maximum Trigger Size.....	9-23
SQL Statements Allowed in Trigger Bodies.....	9-24
Trigger Restrictions on LONG and LONG RAW Datatypes.....	9-24
Trigger Restrictions on Mutating Tables	9-24
Restrictions on Mutating Tables Relaxed	9-25
System Trigger Restrictions.....	9-26
Foreign Function Callouts	9-26
Who Uses the Trigger?	9-26

Compiling Triggers	9-26
Dependencies for Triggers	9-27
Recompiling Triggers	9-27
Modifying Triggers	9-28
Debugging Triggers	9-28
Enabling Triggers	9-28
Disabling Triggers	9-28
Viewing Information About Triggers	9-29
Examples of Trigger Applications	9-30
Auditing with Triggers	9-30
Constraints and Triggers	9-34
Referential Integrity Using Triggers	9-35
Foreign Key Trigger for Child Table	9-35
UPDATE and DELETE RESTRICT Trigger for Parent Table	9-36
UPDATE and DELETE SET NULL Triggers for Parent Table	9-37
DELETE Cascade Trigger for Parent Table	9-37
UPDATE Cascade Trigger for Parent Table	9-38
Trigger for Complex Check Constraints	9-39
Complex Security Authorizations and Triggers	9-40
Transparent Event Logging and Triggers	9-41
Derived Column Values and Triggers	9-41
Building Complex Updatable Views Using Triggers	9-42
Fine-Grained Access Control Using Triggers	9-43
Responding to Database Events Through Triggers	9-44
How Events Are Published Through Triggers	9-45
Publication Context	9-45
Error Handling	9-45
Execution Model	9-45
Event Attribute Functions	9-46
Database Events	9-49
Client Events	9-50

10 Using PL/SQL Packages

What is a PL/SQL Package?	10-1
What Goes in a PL/SQL Package?	10-2
Advantages of PL/SQL Packages	10-3
Understanding the PL/SQL Package Specification	10-4
Referencing PL/SQL Package Contents	10-5
Understanding the PL/SQL Package Body	10-5
Examples of PL/SQL Package Features	10-6
Private and Public Items in PL/SQL Packages	10-9
How STANDARD Package Defines the PL/SQL Environment	10-9
Overview of Product-Specific PL/SQL Packages	10-10
DBMS_ALERT Package	10-10
DBMS_OUTPUT Package	10-10
DBMS_PIPE Package	10-11
DBMS_CONNECTION_POOL Package	10-11

HTF and HTP Packages	10-11
UTL_FILE Package.....	10-11
UTL_HTTP Package	10-11
UTL_SMTP Package	10-12
Guidelines for Writing PL/SQL Packages	10-12
Separating Cursor Specifications and Bodies with PL/SQL Packages	10-12

11 Handling PL/SQL Errors

Overview of PL/SQL Run-Time Error Handling	11-1
Guidelines for Avoiding and Handling PL/SQL Errors and Exceptions.....	11-2
Advantages of PL/SQL Exceptions	11-3
Summary of Predefined PL/SQL Exceptions	11-4
Defining Your Own PL/SQL Exceptions	11-6
Declaring PL/SQL Exceptions	11-6
Scope Rules for PL/SQL Exceptions	11-6
Associating a PL/SQL Exception with a Number (Pragma EXCEPTION_INIT)	11-7
Defining Your Own Error Messages (RAISE_APPLICATION_ERROR Procedure)	11-8
Redeclaring Predefined Exceptions.....	11-9
How PL/SQL Exceptions Are Raised	11-9
How PL/SQL Exceptions Propagate.....	11-10
Reraising a PL/SQL Exception	11-12
Handling Raised PL/SQL Exceptions.....	11-13
Exceptions Raised in Declarations.....	11-14
Handling Exceptions Raised in Exception Handlers	11-14
Branching To or from an Exception Handler.....	11-14
Retrieving the Error Code and Error Message (SQLCODE and SQLERRM Functions)	11-14
Catching Unhandled Exceptions	11-16
Guidelines for Handling PL/SQL Errors	11-16
Continuing Execution After an Exception Is Raised.....	11-16
Retrying a Transaction	11-17
Using Locator Variables to Identify Exception Locations.....	11-18
Overview of PL/SQL Compile-Time Warnings.....	11-18
PL/SQL Warning Categories	11-19
Controlling PL/SQL Warning Messages.....	11-19
Using DBMS_WARNING Package	11-20

12 Tuning PL/SQL Applications for Performance

Initialization Parameters for PL/SQL Compilation.....	12-1
How PL/SQL Optimizes Your Programs.....	12-3
When to Tune PL/SQL Code.....	12-4
Guidelines for Avoiding PL/SQL Performance Problems	12-4
Avoiding CPU Overhead in PL/SQL Code.....	12-4
Make SQL Statements as Efficient as Possible.....	12-5
Make Function Calls as Efficient as Possible	12-5
Make Loops as Efficient as Possible	12-6
Use Built-In String Functions	12-7
Put Least Expensive Conditional Tests First.....	12-7

Minimize Datatype Conversions	12-7
Use PLS_INTEGER or SIMPLE_INTEGER for Integer Arithmetic	12-7
Use BINARY_FLOAT and BINARY_DOUBLE for Floating-Point Arithmetic	12-8
Avoiding Memory Overhead in PL/SQL Code	12-8
Declare VARCHAR2 Variables of 4000 or More Characters	12-8
Group Related Subprograms into Packages	12-8
Pin Packages in the Shared Memory Pool.....	12-9
Apply Advice of Compiler Warnings.....	12-9
Collecting Data About User-Defined Identifiers.....	12-9
Profiling and Tracing PL/SQL Programs	12-9
Using the Profiler API: Package DBMS_PROFILER	12-10
Using the Trace API: Package DBMS_TRACE.....	12-10
Reducing Loop Overhead for DML Statements and Queries with Bulk SQL.....	12-11
Running One DML Statement Multiple Times (FORALL Statement)	12-12
How FORALL Affects Rollbacks	12-15
Counting Rows Affected by FORALL (%BULK_ROWCOUNT Attribute)	12-16
Handling FORALL Exceptions (%BULK_EXCEPTIONS Attribute).....	12-17
Retrieving Query Results into Collections (BULK COLLECT Clause)	12-19
Examples of Bulk Fetching from a Cursor	12-20
Limiting Rows for a Bulk FETCH Operation (LIMIT Clause).....	12-21
Retrieving DML Results Into a Collection (RETURNING INTO Clause)	12-22
Using FORALL and BULK COLLECT Together.....	12-23
Using Host Arrays with Bulk Binds.....	12-23
Writing Computation-Intensive PL/SQL Programs.....	12-24
Tuning Dynamic SQL with EXECUTE IMMEDIATE Statement and Cursor Variables	12-24
Tuning PL/SQL Subprogram Calls with NOCOPY Hint	12-25
Compiling PL/SQL Program Units for Native Execution.....	12-26
Determining Whether to Use PL/SQL Native Compilation	12-27
How PL/SQL Native Compilation Works.....	12-27
Dependencies, Invalidation, and Revalidation.....	12-28
Setting PLSQL_CODE_TYPE Initialization Parameter	12-28
Setting Up a New Database for PL/SQL Native Compilation.....	12-28
Compiling the Entire Database for PL/SQL Native or Interpreted Compilation	12-28
Performing Multiple Transformations with Pipelined Table Functions.....	12-31
Overview of Pipelined Table Functions.....	12-31
Writing a Pipelined Table Function.....	12-32
Using Pipelined Table Functions for Transformations.....	12-32
Returning Results from Pipelined Table Functions	12-33
Pipelining Data Between PL/SQL Table Functions.....	12-34
Optimizing Multiple Calls to Pipelined Table Functions.....	12-34
Fetching from Results of Pipelined Table Functions	12-34
Passing Data with Cursor Variables.....	12-35
Performing DML Operations Inside Pipelined Table Functions	12-37
Performing DML Operations on Pipelined Table Functions.....	12-38
Handling Exceptions in Pipelined Table Functions.....	12-38

13 PL/SQL Language Elements

Assignment Statement	13-3
AUTONOMOUS_TRANSACTION Pragma	13-6
Block Declaration	13-8
CASE Statement.....	13-16
CLOSE Statement.....	13-18
Collection Definition.....	13-19
Collection Methods.....	13-23
Comments	13-27
COMMIT Statement.....	13-28
Constant and Variable Declaration.....	13-29
CONTINUE Statement.....	13-33
Cursor Attributes.....	13-35
Cursor Variables	13-37
Cursor Declaration	13-41
DELETE Statement.....	13-45
EXCEPTION_INIT Pragma	13-47
Exception Definition.....	13-48
EXECUTE IMMEDIATE Statement	13-51
EXIT Statement	13-55
Expression Definition.....	13-57
FETCH Statement.....	13-67
FORALL Statement.....	13-71
Function Declaration and Definition	13-74
GOTO Statement.....	13-79
IF Statement.....	13-80
INLINE Pragma	13-82
INSERT Statement	13-85
Literal Declaration.....	13-88
LOCK TABLE Statement.....	13-91
LOOP Statements	13-93
MERGE Statement	13-98
NULL Statement	13-99
Object Type Declaration	13-100
OPEN Statement.....	13-102
OPEN-FOR Statement	13-104
Package Declaration.....	13-107
Procedure Declaration and Definition.....	13-110
RAISE Statement	13-114
Record Definition.....	13-115
RESTRICT_REFERENCES Pragma	13-118
RETURN Statement	13-120
RETURNING INTO Clause	13-121
ROLLBACK Statement.....	13-124
%ROWTYPE Attribute	13-125
SAVEPOINT Statement.....	13-127
SELECT INTO Statement	13-128

SERIALLY_REUSABLE Pragma	13-132
SET TRANSACTION Statement.....	13-134
SQL Cursor	13-136
SQLCODE Function	13-139
SQLERRM Function	13-140
%TYPE Attribute	13-142
UPDATE Statement.....	13-144

A Wrapping PL/SQL Source Code

What is Wrapping?	A-1
Guidelines for Wrapping PL/SQL Units.....	A-2
Limitations of Wrapping.....	A-2
Limitations of the PL/SQL wrap Utility	A-2
Limitations of the DBMS_DDL wrap Function.....	A-2
Wrapping PL/SQL Code with PL/SQL wrap Utility.....	A-3
Input and Output Files for the PL/SQL wrap Utility.....	A-3
Running the wrap Utility	A-4
Wrapping PL/QL Code with DBMS_DDL Subprograms.....	A-4
Using DBMS_DDL create_wrapped Procedure	A-5

B How PL/SQL Resolves Identifier Names

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

C PL/SQL Program Limits

D PL/SQL Reserved Words and Keywords

Index

Preface

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

Preface topics:

- [Audience](#)
- [Documentation Accessibility](#)
- [Related Documents](#)
- [Conventions](#)
- [Syntax Descriptions](#)

Audience

Oracle Database PL/SQL Language Reference is intended for anyone developing PL/SQL-based applications for the Oracle 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, with good usability, to the disabled community. 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.

TTY Access to Oracle Support Services

Oracle provides dedicated Text Telephone (TTY) access to Oracle Support Services within the United States of America 24 hours a day, seven days a week. For TTY support, call 800.446.2398.

Related Documents

For more information, see the following documents in the Oracle Database 11g Release 1 (11.1) 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*

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
<i>italic</i>	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.

**_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 dictionary view, or about static dictionary views in general, see *Oracle Database Reference*.

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 Reference* for information about how to interpret these descriptions.

What's New in PL/SQL?

This section briefly describes the new PL/SQL features for 11g Release 1 (11.1) and provides links to additional information.

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

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

- [Enhancements to Regular Expression Built-in SQL Functions](#)
- [SIMPLE_INTEGER Datatype](#)
- [CONTINUE Statement](#)
- [Sequences in PL/SQL Expressions](#)
- [Dynamic SQL Enhancements](#)
- [Named and Mixed Notation in PL/SQL Subprogram Invocations](#)
- [Cross-Session PL/SQL Function Result Cache](#)
- [Compound Triggers](#)
- [More Control Over Triggers](#)
- [Database Resident Connection Pool](#)
- [Automatic Subprogram Inlining](#)
- [PL/Scope](#)
- [PL/SQL Hierarchical Profiler](#)
- [PL/SQL Native Compiler Generates Native Code Directly](#)

Enhancements to Regular Expression Built-in SQL Functions

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

See Also:

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

SIMPLE_INTEGER Datatype

The `SIMPLE_INTEGER` datatype is a predefined subtype of the `PLS_INTEGER` datatype that has the same range as `PLS_INTEGER` (-2,147,483,648 through 2,147,483,647) and has a `NOT NULL` constraint. It differs significantly from `PLS_INTEGER` in its overflow semantics.

You can use `SIMPLE_INTEGER` when the value will never be `NULL` and overflow checking is unnecessary. Without the overhead of checking for nullness and overflow, `SIMPLE_INTEGER` provides significantly better performance than `PLS_INTEGER` when `PLSQL_CODE_TYPE= 'NATIVE'`, because arithmetic operations on `SIMPLE_INTEGER` values are done directly in the hardware. When `PLSQL_CODE_TYPE= 'INTERPRETED'`, the performance improvement is smaller.

For more information, see:

- ["SIMPLE_INTEGER Subtype of PLS_INTEGER"](#) on page 3-3
- ["Use PLS_INTEGER or SIMPLE_INTEGER for Integer Arithmetic"](#) on page 12-7

CONTINUE Statement

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

For more information, see:

- ["Controlling Loop Iterations \(LOOP, EXIT, and CONTINUE Statements\)"](#) on page 4-7
- ["CONTINUE Statement"](#) on page 13-33

Sequences in PL/SQL Expressions

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

For more information, see ["CURRVAL and NEXTVAL"](#) on page 6-4.

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 13-51 and ["OPEN-FOR Statement"](#) on page 13-104.

In the `DBMS_SQL` package:

- All datatypes that native dynamic SQL supports are supported.

- The `DBMS_SQL.PARSE` function accepts a `CLOB` argument, allowing dynamic SQL statements larger than 32 KB.
- The new "[DBMS_SQL.TO_REFCURSOR Function](#)" on page 7-7 enables you to switch from the `DBMS_SQL` package to native dynamic SQL.
- The new "[DBMS_SQL.TO_CURSOR_NUMBER Function](#)" on page 7-8 enables you to switch from native dynamic SQL to the `DBMS_SQL` package.

Named and Mixed Notation in PL/SQL Subprogram Invocations

Before Release 11.1, a SQL statement that invoked a PL/SQL subprogram had to specify the actual parameters in positional notation. As of Release 11.1, named and mixed notation are also allowed. This improves usability when a SQL statement invokes a PL/SQL subprogram that has many defaulted parameters, and few of the actual parameters must differ from their default values.

For an example, see the `SELECT` statements following [Example 8-5](#) on page 8-8.

Cross-Session 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 cross-session function result cache. To use it, use the `RESULT_CACHE` clause in each PL/SQL function whose results you want cached. Because the function result cache is stored in a shared global area (SGA), it is available to any session that runs your application.

If you convert your application from single-session caching to cross-session caching, your application will use more SGA, but significantly less total system memory.

For more information, see:

- "[Using the Cross-Session PL/SQL Function Result Cache](#)" on page 8-29
- "[Function Declaration and Definition](#)" on page 13-74

Compound Triggers

A compound trigger is a Database Manipulation Language (DML) trigger that can fire at more than one timing point.

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

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

you want the actions you implement for the various timing points to share common data.

For more information, see ["Compound Triggers"](#) on page 9-12.

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 allows you to control the firing order of triggers that are defined on the same table and have the same timing point.

For more information, see:

- ["Ordering of Triggers"](#) on page 9-6
- ["Enabling Triggers"](#) on page 9-28
- ["Disabling Triggers"](#) on page 9-28

See Also: *Oracle Database SQL Language Reference* for information about the `CREATE TRIGGER` statement

Database Resident Connection Pool

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

For more information, see ["DBMS_CONNECTION_POOL Package"](#) on page 10-11.

Automatic Subprogram Inlining

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

You can use `PRAGMA INLINE` to specify that individual subprogram calls are, or are not, to be inlined. You can also turn on automatic inlining—that is, ask the compiler to search for inlining opportunities—by setting the Oracle 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:

- ["How PL/SQL Optimizes Your Programs"](#) on page 12-3
- ["INLINE Pragma"](#) on page 13-82

See Also: *Oracle Database Reference* for information about the initialization parameter `PLSQL_OPTIMIZE_LEVEL`

PL/Scope

PL/Scope is a compiler-driven tool that collects and organizes data about user-defined identifiers from PL/SQL source code. Because PL/Scope is a compiler-driven tool, you

use it through interactive development environments (such as SQL Developer and JDeveloper), rather than directly.

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

For more information, see ["Collecting Data About User-Defined Identifiers"](#) on page 12-9.

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

PL/SQL Hierarchical Profiler

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

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

For more information, see ["Profiling and Tracing PL/SQL Programs"](#) on page 12-9.

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

PL/SQL Native Compiler Generates Native Code Directly

The PL/SQL native compiler now generates native code directly, instead of translating PL/SQL code to C code and having the C compiler generate the native code. An individual developer can now compile program 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 Program Units for Native Execution"](#) on page 12-26.

Overview of PL/SQL

This chapter introduces the main features of the PL/SQL language. It shows how PL/SQL meets the challenges of database programming, and how you can reuse techniques that you know from other programming languages.

Topics:

- [Advantages of PL/SQL](#)
- [Understanding the Main Features of PL/SQL](#)
- [PL/SQL Architecture](#)

Advantages of PL/SQL

PL/SQL is a completely portable, high-performance transaction processing language that offers the following advantages:

- [Tight Integration with SQL](#)
- [Better Performance](#)
- [Higher Productivity](#)
- [Full Portability](#)
- [Tight Security](#)
- [Access to Predefined Packages](#)
- [Support for Object-Oriented Programming](#)
- [Support for Developing Web Applications and Pages](#)

Tight Integration with SQL

SQL has become the standard database language because it is flexible, powerful, and easy to learn. A few English-like statements such as `SELECT`, `INSERT`, `UPDATE`, and `DELETE` make it easy to manipulate the data stored in a relational database.

PL/SQL lets you use all the SQL data manipulation, cursor control, and transaction control statements, as well as all the SQL functions, operators, and pseudocolumns. This extensive SQL support lets you manipulate Oracle data flexibly and safely. Also, PL/SQL fully supports SQL datatypes, reducing the need to convert data passed between your applications and the database.

The PL/SQL language is tightly integrated with SQL. You do not have to translate between SQL and PL/SQL datatypes; a `NUMBER` or `VARCHAR2` column in the database is stored in a `NUMBER` or `VARCHAR2` variable in PL/SQL. This integration saves you

both learning time and processing time. Special PL/SQL language features let you work with table columns and rows without specifying the datatypes, saving on maintenance work when the table definitions change.

Running a SQL query and processing the result set is as easy in PL/SQL as opening a text file and processing each line in popular scripting languages. Using PL/SQL to access metadata about database objects and handle database error conditions, you can write utility programs for database administration that are reliable and produce readable output about the success of each operation. Many database features, such as triggers and object types, use PL/SQL. You can write the bodies of triggers and methods for object types in PL/SQL.

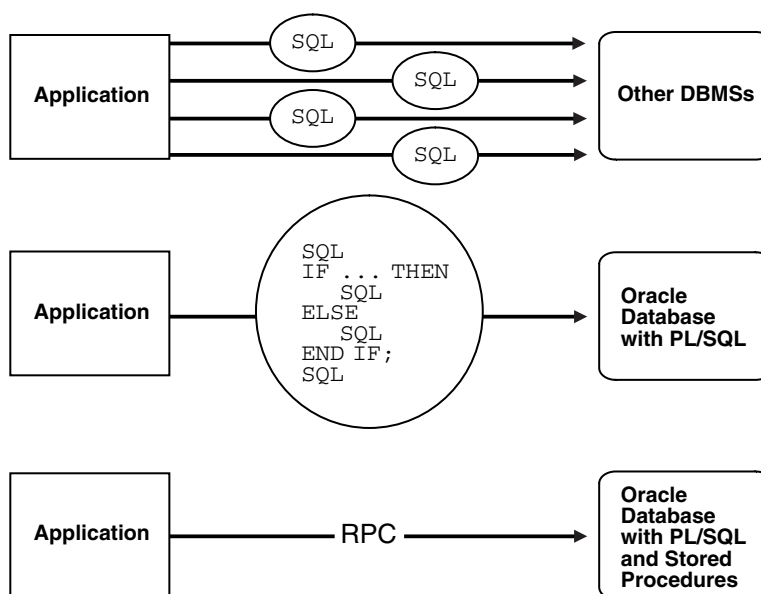
PL/SQL supports both static and dynamic SQL. **Static SQL** is SQL whose full text is known at compilation time. **Dynamic SQL** is SQL whose full text is not known until run time. Dynamic SQL enables you to make your applications more flexible and versatile. For information about using static SQL with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

Better Performance

With PL/SQL, an entire block of statements can be sent to Oracle at one time. This can drastically reduce network traffic between the database and an application. As [Figure 1-1](#) shows, you can use PL/SQL blocks and subprograms (procedures and functions) to group SQL statements before sending them to the database for execution. PL/SQL also has language features to further speed up SQL statements that are issued inside a loop.

PL/SQL stored subprograms are compiled once and stored in executable form, so subprogram calls are efficient. Because stored subprograms execute in the database server, a single call over the network can start a large job. This division of work reduces network traffic and improves response times. Stored subprograms are cached and shared among users, which lowers memory requirements and call overhead.

Figure 1-1 PL/SQL Boosts Performance



Higher Productivity

PL/SQL lets you write very compact code for manipulating data. In the same way that scripting languages such as Perl can read, transform, and write data from files, PL/SQL can query, transform, and update data in a database. PL/SQL saves time on design and debugging by offering a full range of software-engineering features, such as exception handling, encapsulation, data hiding, and object-oriented datatypes.

PL/SQL extends tools such as Oracle Forms. With PL/SQL in these tools, you can use familiar language constructs to build applications. For example, you can use an entire PL/SQL block in an Oracle Forms trigger, instead of multiple trigger steps, macros, or user exits. PL/SQL is the same in all environments. After you learn PL/SQL with one Oracle tool, you can transfer your knowledge to other tools.

Full Portability

Applications written in PL/SQL can run on any operating system and platform where the Oracle database runs. With PL/SQL, you can write portable program libraries and reuse them in different environments.

Tight Security

PL/SQL stored subprograms move application code from the client to the server, where you can protect it from tampering, hide the internal details, and restrict who has access. For example, you can grant users access to a subprogram that updates a table, but not grant them access to the table itself or to the text of the UPDATE statement. Triggers written in PL/SQL can control or record changes to data, making sure that all changes obey your business rules.

For information on wrapping, or hiding, the source of a PL/SQL unit, see [Appendix A, "Wrapping PL/SQL Source Code"](#).

Access to Predefined Packages

Oracle provides product-specific packages that define APIs you can invoke from PL/SQL to perform many useful tasks. These packages include DBMS_ALERT for using database triggers, DBMS_FILE for reading and writing operating system text files, DBMS_HTTP for making hypertext transfer protocol (HTTP) callouts, DBMS_OUTPUT for display output from PL/SQL blocks and subprograms, and DBMS_PIPE for communicating over named pipes. For additional information on these packages, see ["Overview of Product-Specific PL/SQL Packages"](#) on page 10-10.

For complete information on the packages supplied by Oracle, see *Oracle Database PL/SQL Packages and Types Reference*.

Support for Object-Oriented Programming

Object types are an ideal object-oriented modeling tool, which you can use to reduce the cost and time required to build complex applications. Besides enabling you to create software components that are modular, maintainable, and reusable, object types allow different teams of programmers to develop software components concurrently.

By encapsulating operations with data, object types let you move data-maintenance code out of SQL scripts and PL/SQL blocks into methods. Also, object types hide implementation details, so that you can change the details without affecting client programs.

In addition, object types allow for realistic data modeling. Complex real-world entities and relationships map directly into object types. This direct mapping helps your programs better reflect the world they are trying to simulate. For information on object types, see *Oracle Database Object-Relational Developer's Guide*.

Support for Developing Web Applications and Pages

You can use PL/SQL to develop Web applications and Server Pages (PSPs). For an overview of the use of PL/SQL with the Web, see "[Using PL/SQL to Create Web Applications and Server Pages](#)" on page 2-42.

Understanding the Main Features of PL/SQL

PL/SQL combines the data-manipulating power of SQL with the processing power of procedural languages. You can control program flow with statements like `IF` and `LOOP`. As with other procedural programming languages, you can declare variables, define subprograms (procedures and functions), and trap run-time errors.

PL/SQL lets you break complex problems down into easily understandable procedural code, and reuse this code across multiple applications. When a problem can be solved through plain SQL, you can issue SQL statements directly inside your PL/SQL programs, without learning new APIs. PL/SQL's data types correspond with SQL's column types, making it easy to interchange PL/SQL variables with data inside a table.

Topics:

- [Understanding PL/SQL Block Structure](#)
- [Understanding PL/SQL Variables and Constants](#)
- [Processing Queries with PL/SQL](#)
- [Declaring PL/SQL Subprograms](#)
- [Declaring Datatypes for PL/SQL Variables](#)
- [Understanding PL/SQL Control Structures](#)
- [Understanding Conditional Compilation](#)
- [Writing Reusable PL/SQL Code](#)
- [Inputting and Outputting Data with PL/SQL](#)
- [Understanding PL/SQL Data Abstraction](#)
- [Understanding PL/SQL Error Handling](#)

Understanding PL/SQL Block Structure

The basic units of a PL/SQL program are logical blocks (procedures, functions, and anonymous blocks), which can be nested. A block groups related declarations and statements. You can place declarations close to where they are used, such as inside a large subprogram. The declarations are local to the block and cease to exist when the block completes, helping to avoid cluttered namespaces for variables and subprograms. For a syntax description of the block structure, see "[Block Declaration](#)" on page 13-8.

As [Example 1-1](#) shows the three parts of a PL/SQL block. Only the executable part is required. For an example of PL/SQL block structure, see [Example 1-4](#) on page 1-6.

Example 1–1 PL/SQL Block Structure

```
-- Declarative part (optional):
DECLARE
    -- Declarations of types, variables, and subprograms
-- Executable part (required):
BEGIN
    -- Statements (which can use items declared in declarative part)
-- Exception-handling part (optional:
[EXCEPTION
    -- Exception handlers for exceptions raised in executable part]
END;
```

You can nest blocks in the executable and exception-handling parts of a PL/SQL block or subprogram but not in the declarative part. You can define local subprograms in the declarative part of any block. You can invoke local subprograms only from the block in which they are defined.

Understanding PL/SQL Variables and Constants

PL/SQL lets you declare variables and constants, then use them in SQL and procedural statements anywhere an expression can be used. You must declare a constant or variable before referencing it in any other statements. For additional information, see ["Declarations"](#) on page 2-9.

Topics:

- [Declaring Variables](#)
- [Assigning Values to Variables](#)
- [Bind Variables](#)
- [Declaring Constants](#)

Declaring Variables

Variables can have any SQL datatype, such as CHAR, DATE, or NUMBER, or a PL/SQL-only datatype, such as BOOLEAN or PLS_INTEGER. For example, assume that you want to declare variables for part data, such as `part_no` to hold 6-digit numbers and `in_stock` to hold the Boolean value TRUE or FALSE. You declare these and related part variables as shown in [Example 1–2](#). Note that there is a semicolon (;) at the end of each line in the declaration section.

Example 1–2 Declaring Variables in PL/SQL

```
DECLARE
    part_no    NUMBER(6);
    part_name  VARCHAR2(20);
    in_stock   BOOLEAN;
    part_price NUMBER(6,2);
    part_desc  VARCHAR2(50);
```

You can also declare nested tables, variable-size arrays (varrays for short), and records using the TABLE, VARRAY, and RECORD composite datatypes. See [Chapter 5, "Using PL/SQL Collections and Records"](#).

Assigning Values to Variables

You can assign values to a variable in three ways. The first way uses the assignment operator (: =), a colon followed by an equal sign, as shown in [Example 1–3](#). You place

the variable to the left of the operator and an expression, including function calls, to the right. Note that you can assign a value to a variable when it is declared.

Example 1–3 Assigning Values to Variables with the Assignment Operator

```

DECLARE
    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
    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;
/

```

The second way to assign values to a variable is by selecting (or fetching) database values into it. In [Example 1–4](#), 10% of an employee's salary is selected into the bonus variable. Now you can use the bonus variable in another computation or insert its value into a database table.

Example 1–4 Using SELECT INTO to Assign Values to Variables

```

DECLARE
    bonus NUMBER(8,2);
    emp_id NUMBER(6) := 100;
BEGIN
    SELECT salary * 0.10 INTO bonus FROM employees
    WHERE employee_id = emp_id;
END;
/

```

The third way to assign a value to a variable is by passing it as an OUT or IN OUT parameter to a subprogram, and then assigning the value inside the subprogram. [Example 1–5](#) passes the sal variable to a subprogram, and the subprogram updates the variable.

In the example, DBMS_OUTPUT.PUT_LINE is used to display output from the PL/SQL program. For more information, see ["Inputting and Outputting Data with PL/SQL"](#) on page 1-16. For information on the DBMS_OUTPUT package, see ["DBMS_OUTPUT Package"](#) on page 10-10.

Example 1–5 Assigning Values to Variables as Parameters of a Subprogram

```

REM SERVEROUTPUT must be set to ON

```

```

REM to display output with DBMS_OUTPUT
SET SERVEROUTPUT ON FORMAT WRAPPED
DECLARE
  new_sal NUMBER(8,2);
  emp_id  NUMBER(6) := 126;
  PROCEDURE adjust_salary(emp_id NUMBER, sal IN OUT NUMBER) IS
    emp_job VARCHAR2(10);
    avg_sal NUMBER(8,2);
  BEGIN
    SELECT job_id INTO emp_job FROM employees
      WHERE employee_id = emp_id;
    SELECT AVG(salary) INTO avg_sal FROM employees
      WHERE job_id = emp_job;
    DBMS_OUTPUT.PUT_LINE
      ('The average salary for ' || emp_job
      || ' employees: ' || TO_CHAR(avg_sal));
    sal := (sal + avg_sal)/2;
    -- adjust sal value which is returned
  END;
BEGIN
  SELECT AVG(salary) INTO new_sal FROM employees;
  DBMS_OUTPUT.PUT_LINE ('The average salary for all employees: '
    || TO_CHAR(new_sal));
  adjust_salary(emp_id, new_sal);
  -- assigns a new value to new_sal
  DBMS_OUTPUT.PUT_LINE
    ('The adjusted salary for employee ' || TO_CHAR(emp_id)
    || ' is ' || TO_CHAR(new_sal)); -- sal has new value
END;
/

```

Bind Variables

Bind variables improve performance by allowing Oracle to reuse SQL statements.

When you embed a SQL INSERT, UPDATE, DELETE, or SELECT statement directly in your PL/SQL code, PL/SQL turns the variables in the WHERE and VALUES clauses into bind variables automatically. Oracle can reuse these SQL statements each time the same code is executed. To run similar statements with different variable values, you can save parsing overhead by invoking a stored subprogram that accepts parameters and then issues the statements with the parameters substituted in the appropriate places.

PL/SQL does not create bind variables automatically when you use dynamic SQL, but you can use them with dynamic SQL by specifying them explicitly.

Declaring Constants

Declaring a constant is like declaring a variable except that you must add the keyword CONSTANT and immediately assign a value to the constant. No further assignments to the constant are allowed. The following example declares a constant:

```
credit_limit CONSTANT NUMBER := 5000.00;
```

See "[Constants](#)" on page 2-10.

Processing Queries with PL/SQL

Processing a SQL query with PL/SQL is like processing files with other languages. For example, a Perl program opens a file, reads the file contents, processes each line, then

closes the file. In the same way, a PL/SQL program issues a query and processes the rows from the result set as shown in [Example 1-6](#).

Example 1-6 Processing Query Results in a LOOP

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

You can use a simple loop like the one shown here, or you can control the process precisely by using individual statements to perform the query, retrieve data, and finish processing.

Declaring PL/SQL Subprograms

Subprograms are named PL/SQL blocks that can be invoked with a set of parameters. PL/SQL has two types of subprograms: procedures and functions. The following is an example of a declaration of a PL/SQL procedure:

```
DECLARE
  in_string VARCHAR2(100) := 'This is my test string.';
  out_string VARCHAR2(200);
  PROCEDURE double
    ( original IN VARCHAR2, new_string OUT VARCHAR2 ) AS
  BEGIN
    new_string := original || original;
  END;
```

For example of a subprogram declaration in a package, see [Example 1-14](#) on page 1-15. For more information on subprograms, see "[What Are PL/SQL Subprograms?](#)" on page 8-1.

You can create standalone subprograms with SQL statements that are stored in the database. See "[Subprograms \(Procedures and Functions\)](#)" on page 1-13.

Declaring Datatypes for PL/SQL Variables

As part of the declaration for each PL/SQL variable, you declare its datatype. Usually, this datatype is one of the types shared between PL/SQL and SQL, such as NUMBER or VARCHAR2. For easier maintenance of code that interacts with the database, you can also use the special qualifiers %TYPE and %ROWTYPE to declare variables that hold table columns or table rows. For more information on datatypes, see [Chapter 3, "PL/SQL Datatypes"](#).

Topics:

- [%TYPE](#)
- [%ROWTYPE](#)

%TYPE

The %TYPE attribute provides the datatype of a variable or database column. This is particularly useful when declaring variables that will hold database values. For example, assume there is a column named last_name in a table named employees.

To declare a variable named `v_last_name` that has the same datatype as column `last_name`, use dot notation and the `%TYPE` attribute, as follows:

```
v_last_name employees.last_name%TYPE;
```

Declaring `v_last_name` with `%TYPE` has two advantages. First, you need not know the exact datatype of `last_name`. Second, if you change the database definition of `last_name`, perhaps to make it a longer character string, the datatype of `v_last_name` changes accordingly at run time.

For more information on `%TYPE`, see ["Using the %TYPE Attribute"](#) on page 2-10 and ["%TYPE Attribute"](#) on page 13-142.

%ROWTYPE

In PL/SQL, records are used to group data. A record consists of a number of related fields in which data values can be stored. The `%ROWTYPE` attribute provides a record type that represents a row in a table. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable. See ["Cursors"](#) on page 1-17.

Columns in a row and corresponding fields in a record have the same names and datatypes. In the following example, you declare a record named `dept_rec`. Its fields have the same names and datatypes as the columns in the `departments` table.

```
DECLARE
  dept_rec departments%ROWTYPE; -- declare record variable
```

You use dot notation to reference fields, as the following example shows:

```
v_deptid := dept_rec.department_id;
```

If you declare a cursor that retrieves the last name, salary, hire date, and job class of an employee, you can use `%ROWTYPE` to declare a record that stores the same information as shown in [Example 1-7](#). When you execute the `FETCH` statement, the value in the `last_name` column of the `employees` table is assigned to the `last_name` field of `employee_rec`, the value in the `salary` column is assigned to the `salary` field, and so on.

Example 1-7 Using %ROWTYPE with an Explicit Cursor

```
DECLARE
  CURSOR c1 IS
    SELECT last_name, salary, hire_date, job_id FROM employees
       WHERE employee_id = 120;
  -- declare record variable that represents a row
  -- fetched from the employees table
  employee_rec c1%ROWTYPE;
BEGIN
  -- open the explicit cursor
  -- and use it to fetch data into employee_rec
  OPEN c1;
  FETCH c1 INTO employee_rec;
  DBMS_OUTPUT.PUT_LINE('Employee name: ' || employee_rec.last_name);
END;
```

For more information on `%ROWTYPE`, see ["Using the %ROWTYPE Attribute"](#) on page 2-12 and ["%ROWTYPE Attribute"](#) on page 13-125.

Understanding PL/SQL Control Structures

Control structures are the most important PL/SQL extension to SQL. Not only does PL/SQL let you manipulate Oracle data, it lets you process the data using conditional, iterative, and sequential flow-of-control statements such as `IF-THEN-ELSE`, `CASE`, `FOR-LOOP`, `WHILE-LOOP`, `CONTINUE-WHEN`, `EXIT-WHEN`, and `GOTO`. For additional information, see [Chapter 4, "Using PL/SQL Control Structures"](#).

Topics:

- [Conditional Control](#)
- [Iterative Control](#)
- [Sequential Control](#)

Conditional Control

Often, it is necessary to take alternative actions depending on circumstances. The `IF-THEN-ELSE` statement lets you execute a sequence of statements conditionally. The `IF` clause checks a condition, the `THEN` clause defines what to do if the condition is true and the `ELSE` clause defines what to do if the condition is false or null.

[Example 1-8](#) shows the use of `IF-THEN-ELSE` to determine the salary raise an employee receives based on the current salary of the employee.

To choose among several values or courses of action, you can use `CASE` constructs. The `CASE` expression evaluates a condition and returns a value for each case. The case statement evaluates a condition and performs an action, such as an entire PL/SQL block, for each case. See [Example 1-8](#).

Example 1-8 Using the `IF-THEN_ELSE` and `CASE` Statement for Conditional Control

```

DECLARE
    jobid      employees.job_id%TYPE;
    empid      employees.employee_id%TYPE := 115;
    sal        employees.salary%TYPE;
    sal_raise  NUMBER(3,2);
BEGIN
    SELECT job_id, salary INTO jobid, sal from employees
       WHERE employee_id = empid;
    CASE
        WHEN jobid = 'PU_CLERK' THEN
            IF sal < 3000 THEN sal_raise := .12;
            ELSE sal_raise := .09;
            END IF;
        WHEN jobid = 'SH_CLERK' THEN
            IF sal < 4000 THEN sal_raise := .11;
            ELSE sal_raise := .08;
            END IF;
        WHEN jobid = 'ST_CLERK' THEN
            IF sal < 3500 THEN sal_raise := .10;
            ELSE sal_raise := .07;
            END IF;
        ELSE
            BEGIN
                DBMS_OUTPUT.PUT_LINE('No raise for this job: ' || jobid);
            END;
    END CASE;
    UPDATE employees SET salary = salary + salary * sal_raise
       WHERE employee_id = empid;
    COMMIT;
END;
```

/

A sequence of statements that uses query results to select alternative actions is common in database applications. Another common sequence inserts or deletes a row only if an associated entry is found in another table. You can bundle these common sequences into a PL/SQL block using conditional logic.

Iterative Control

LOOP statements let you execute a sequence of statements multiple times. You place the keyword LOOP before the first statement in the sequence and the keywords END LOOP after the last statement in the sequence. The following example shows the simplest kind of loop, which repeats a sequence of statements continually:

```
LOOP
  -- sequence of statements
END LOOP;
```

The FOR-LOOP statement lets you specify a range of integers, then execute a sequence of statements once for each integer in the range. In [Example 1-9](#) the loop inserts 100 numbers, square roots, squares, and the sum of squares into a database table:

Example 1-9 Using the FOR-LOOP

```
CREATE TABLE sqr_root_sum (num NUMBER, sq_root NUMBER(6,2),
                           sqr NUMBER, sum_sqr NUMBER);
DECLARE
  s PLS_INTEGER;
BEGIN
  FOR i in 1..100 LOOP
    s := (i * (i + 1) * (2*i + 1)) / 6; -- sum of squares
    INSERT INTO sqr_root_sum VALUES (i, SQRT(i), i*i, s );
  END LOOP;
END;
```

The WHILE-LOOP statement associates a condition with a sequence of statements. Before each iteration of the loop, the condition is evaluated. If the condition is true, the sequence of statements is executed, then control resumes at the top of the loop. If the condition is false or null, the loop is bypassed and control passes to the next statement.

In [Example 1-10](#), you find the first employee who has a salary over \$15000 and is higher in the chain of command than employee 120:

Example 1-10 Using WHILE-LOOP for Control

```
CREATE TABLE temp (tempid NUMBER(6),
                   tempsal NUMBER(8,2),
                   tempname VARCHAR2(25));
DECLARE
  sal          employees.salary%TYPE := 0;
  mgr_id      employees.manager_id%TYPE;
  lname       employees.last_name%TYPE;
  starting_empid employees.employee_id%TYPE := 120;
BEGIN
  SELECT manager_id INTO mgr_id
  FROM employees
  WHERE employee_id = starting_empid;
  WHILE sal <= 15000 LOOP -- loop until sal > 15000
    SELECT salary, manager_id, last_name INTO sal, mgr_id, lname
```

```

        FROM employees WHERE employee_id = mgr_id;
    END LOOP;
    INSERT INTO temp VALUES (NULL, sal, lname);
    -- insert NULL for tempid
    COMMIT;
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        INSERT INTO temp VALUES (NULL, NULL, 'Not found');
        -- insert NULLs
        COMMIT;
END;
/

```

The `EXIT-WHEN` statement lets you complete a loop if further processing is impossible or undesirable. When the `EXIT` statement is encountered, the condition in the `WHEN` clause is evaluated. If the condition is true, the loop completes and control passes to the next statement. In [Example 1–11](#), the loop completes when the value of `total` exceeds 25,000:

Similarly, the `CONTINUE-WHEN` statement immediately transfers control to the next iteration of the loop when there is no need to continue working on this iteration.

Example 1–11 Using the `EXIT-WHEN` Statement

```

DECLARE
    total    NUMBER(9) := 0;
    counter  NUMBER(6) := 0;
BEGIN
    LOOP
        counter := counter + 1;
        total := total + counter * counter;
        -- exit loop when condition is true
        EXIT WHEN total > 25000;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE
    ('Counter: ' || TO_CHAR(counter) || ' Total: ' || TO_CHAR(total));
END;
/

```

Sequential Control

The `GOTO` statement lets you branch to a label unconditionally. The label, an undeclared identifier enclosed by double angle brackets, must precede an executable statement or a PL/SQL block. When executed, the `GOTO` statement transfers control to the labeled statement or block, as shown in [Example 1–12](#).

Example 1–12 Using the `GOTO` Statement

```

DECLARE
    total    NUMBER(9) := 0;
    counter  NUMBER(6) := 0;
BEGIN
    <<calc_total>>
        counter := counter + 1;
        total := total + counter * counter;
        -- branch to print_total label when condition is true
        IF total > 25000 THEN GOTO print_total;
        ELSE GOTO calc_total;
        END IF;
    <<print_total>>

```



```

DBMS_OUTPUT.PUT_LINE
('Counter: ' || TO_CHAR(counter) || ' Total: ' || TO_CHAR(total));
END;
/

```

Understanding Conditional Compilation

Using conditional compilation, you can customize the functionality in a compiled PL/SQL application by conditionalizing functionality rather than removing any source code. For example, conditional compilation enables you to determine which PL/SQL features in a PL/SQL application are used for specific database releases. The latest PL/SQL features in an application can be run on a new database release while at the same time those features can be conditionalizing so that the same application is compatible with a previous database release. Conditional compilation is also useful when you want to execute debugging subprograms in a development environment, but want to turn off the debugging routines in a production environment. See ["Conditional Compilation"](#) on page 2-34.

Writing Reusable PL/SQL Code

PL/SQL lets you break an application down into manageable, well-defined modules. PL/SQL meets this need with program units, which include blocks, subprograms, and packages. You can reuse program units by loading them into the database as triggers and stored subprograms. For additional information, see [Chapter 8, "Using PL/SQL Subprograms"](#), [Chapter 9, "Using Triggers"](#), and [Chapter 10, "Using PL/SQL Packages"](#).

Topics:

- [Subprograms \(Procedures and Functions\)](#)
- [Packages \(APIs Written in PL/SQL\)](#)

Subprograms (Procedures and Functions)

There are two types of subprograms, procedures and functions, which can accept parameters and be invoked. See ["What Are PL/SQL Subprograms?"](#) on page 8-1.

The SQL CREATE PROCEDURE statement lets you create standalone procedures that are stored in the database. For information, see CREATE PROCEDURE in *Oracle Database SQL Language Reference*. The SQL CREATE FUNCTION statement lets you create standalone functions that are stored in an Oracle database. For information, see CREATE FUNCTION in *Oracle Database SQL Language Reference*. These stored (schema level) subprograms can be accessed from SQL.

As shown in [Example 1-13](#), a subprogram is like a miniature program, beginning with a header followed by an optional declarative part, an executable part, and an optional exception-handling part.

Example 1-13 Creating a Stored Subprogram

```

-- including OR REPLACE is more convenient
-- when updating a subprogram
CREATE OR REPLACE PROCEDURE award_bonus
(emp_id NUMBER, bonus NUMBER) AS
  commission          REAL;
  comm_missing EXCEPTION;
BEGIN -- executable part starts here
  SELECT commission_pct / 100 INTO commission FROM employees
  WHERE employee_id = emp_id;

```

```
IF commission IS NULL THEN
    RAISE comm_missing;
ELSE
    UPDATE employees SET salary = salary + bonus*commission
    WHERE employee_id = emp_id;
END IF;
EXCEPTION -- exception-handling part starts here
WHEN comm_missing THEN
    DBMS_OUTPUT.PUT_LINE
        ('This employee does not receive a commission. ');
    commission := 0;
WHEN OTHERS THEN
    NULL; -- for other exceptions do nothing
END award_bonus;
/
CALL award_bonus(150, 400);
```

When invoked, this procedure accepts an employee Id and a bonus amount. It uses the Id to select the employee's commission percentage from a database table and, at the same time, convert the commission percentage to a decimal amount. Then, it checks the commission amount. If the commission is null, an exception is raised; otherwise, the employee's salary is updated.

Packages (APIs Written in PL/SQL)

PL/SQL lets you bundle logically related types, variables, cursors, and subprograms into a database object called a package. The package defines a simple, clear, interface to a set of related subprograms and types that can be accessed by SQL statements.

Packages usually have two parts: a specification and a body. The specification defines the application programming interface; it declares the types, constants, variables, exceptions, cursors, and subprograms. The body fills in the SQL queries for cursors and the code for subprograms.

To create package specs, use the SQL statement `CREATE PACKAGE`. A `CREATE PACKAGE BODY` statement defines the package body. For information on the `CREATE PACKAGE SQL` statement, see *Oracle Database SQL Language Reference*. For information on the `CREATE PACKAGE BODY SQL` statement, see *Oracle Database SQL Language Reference*.

In [Example 1–14](#), the `emp_actions` package contains two procedures that update the `employees` table and one function that provides information.

Example 1–14 Creating a Package and Package Body

```
-- Package specification:
CREATE OR REPLACE PACKAGE emp_actions AS
    PROCEDURE hire_employee (employee_id    NUMBER,
                            last_name       VARCHAR2,
                            first_name     VARCHAR2,
                            email          VARCHAR2,
                            phone_number   VARCHAR2,
                            hire_date      DATE,
                            job_id         VARCHAR2,
                            salary         NUMBER,
                            commission_pct  NUMBER,
                            manager_id     NUMBER,
                            department_id   NUMBER);
    PROCEDURE fire_employee (emp_id NUMBER);
    FUNCTION num_above_salary (emp_id NUMBER) RETURN NUMBER;
```

```

END emp_actions;
/
-- Package body:
CREATE OR REPLACE PACKAGE BODY emp_actions AS
  -- Code for procedure hire_employee:
  PROCEDURE hire_employee (employee_id    NUMBER,
                           last_name      VARCHAR2,
                           first_name     VARCHAR2,
                           email          VARCHAR2,
                           phone_number   VARCHAR2,
                           hire_date      DATE,
                           job_id          VARCHAR2,
                           salary         NUMBER,
                           commission_pct NUMBER,
                           manager_id     NUMBER,
                           department_id  NUMBER) IS

  BEGIN
    INSERT INTO employees VALUES (employee_id,
                                   last_name,
                                   first_name,
                                   email,
                                   phone_number,
                                   hire_date,
                                   job_id,
                                   salary,
                                   commission_pct,
                                   manager_id,
                                   department_id);

  END hire_employee;

  -- Code for procedure fire_employee:
  PROCEDURE fire_employee (emp_id NUMBER) IS
  BEGIN
    DELETE FROM employees WHERE employee_id = emp_id;
  END fire_employee;

  -- Code for function num_above_salary:
  FUNCTION num_above_salary (emp_id NUMBER) RETURN NUMBER IS
    emp_sal NUMBER(8,2);
    num_count NUMBER;
  BEGIN
    SELECT salary INTO emp_sal FROM employees
      WHERE employee_id = emp_id;
    SELECT COUNT(*) INTO num_count FROM employees
      WHERE salary > emp_sal;
    RETURN num_count;
  END num_above_salary;
END emp_actions;
/

```

Applications that invoke these procedures only need to know the names and parameters from the package specification. You can change the implementation details inside the package body without affecting the invoking applications.

To invoke the procedures of the `emp_actions` package created in [Example 1-14](#), you can execute the statements in [Example 1-15](#). The procedures can be executed in a `BEGIN-END` block or with the `SQL CALL` statement. Note the use of the package name as a prefix to the procedure name.

Example 1–15 Invoking a Procedure in a Package

```

CALL emp_actions.hire_employee(300,
                               'Belden',
                               'Enrique',
                               'EBELDEN',
                               '555.111.2222',
                               '31-AUG-04',
                               'AC_MGR',
                               9000,
                               .1,
                               101,
                               110);

BEGIN
  DBMS_OUTPUT.PUT_LINE
    ('Number of employees with higher salary: ' ||
     TO_CHAR(emp_actions.num_above_salary(120)));
  emp_actions.fire_employee(300);
END;
/

```

Packages are stored in the database, where they can be shared by many applications. Invoking a packaged subprogram for the first time loads the whole package and caches it in memory, saving on disk I/O for subsequent invocations. Thus, packages enhance reuse and improve performance in a multiuser, multi-application environment. For information on packages, see [Chapter 10, "Using PL/SQL Packages"](#).

If a subprogram does not take any parameters, you can include an empty set of parentheses or omit the parentheses, both in PL/SQL and in functions invoked from SQL queries. For calls to a method that takes no parameters, an empty set of parentheses is optional within PL/SQL scopes but required within SQL scopes.

Inputting and Outputting Data with PL/SQL

Most PL/SQL input and output is through SQL statements, to store data in database tables or query those tables. All other PL/SQL I/O is done through APIs that interact with other programs. For example, the DBMS_OUTPUT package has procedures such as PUT_LINE. To see the result outside of PL/SQL requires another program, such as SQL*Plus, to read and display the data passed to DBMS_OUTPUT.

SQL*Plus does not display DBMS_OUTPUT data unless you first issue the SQL*Plus command SET SERVEROUTPUT ON as follows:

```
SET SERVEROUTPUT ON
```

For information on the SERVEROUTPUT setting, see *SQL*Plus User's Guide and Reference*.

Other PL/SQL APIs for processing I/O are:

- HTF and HTP for displaying output on a web page
- DBMS_PIPE for passing information back and forth between PL/SQL and operating-system commands
- UTL_FILE for reading and writing operating-system files
- UTL_HTTP for communicating with web servers
- UTL_SMTP for communicating with mail servers

See "[Overview of Product-Specific PL/SQL Packages](#)" on page 10-10. Although some of these APIs can accept input as well as output, there is no built-in language facility

for accepting data directly from the keyboard. For that, you can use the `PROMPT` and `ACCEPT` statements in SQL*Plus.

Understanding PL/SQL Data Abstraction

Data abstraction lets you work with the essential properties of data without being too involved with details. After you design a data structure, you can focus on designing algorithms that manipulate the data structure.

Topics:

- [Cursors](#)
- [Collections](#)
- [Records](#)
- [Object Types](#)

Cursors

A cursor is a name for a specific private SQL area in which information for processing the specific statement is kept. PL/SQL uses both implicit and explicit cursors. PL/SQL implicitly declares a cursor for all SQL data manipulation statements on a set of rows, including queries that return only one row. For queries that return more than one row, you can explicitly declare a cursor to process the rows individually. For example, [Example 1-7](#) on page 1-9 declares an explicit cursor.

For information on managing cursors with PL/SQL, see "[Managing Cursors in PL/SQL](#)" on page 6-7.

Collections

PL/SQL collection types let you declare high-level datatypes similar to arrays, sets, and hash tables found in other languages. In PL/SQL, array types are known as `varrays` (short for variable-size arrays), set types are known as nested tables, and hash table types are known as associative arrays. Each kind of collection is an ordered group of elements, all of the same type. Each element has a unique subscript that determines its position in the collection. When declaring collections, you use a `TYPE` definition. See "[Defining Collection Types](#)" on page 5-7.

To reference an element, use subscript notation with parentheses, as shown in [Example 1-16](#).

Example 1-16 Using a PL/SQL Collection Type

```
DECLARE
    TYPE staff_list IS TABLE OF employees.employee_id%TYPE;
    staff staff_list;
    lname employees.last_name%TYPE;
    fname employees.first_name%TYPE;
BEGIN
    staff := staff_list(100, 114, 115, 120, 122);
    FOR i IN staff.FIRST..staff.LAST LOOP
        SELECT last_name, first_name INTO lname, fname FROM employees
           WHERE employees.employee_id = staff(i);
        DBMS_OUTPUT.PUT_LINE (TO_CHAR(staff(i)) ||
                               ': ' || lname || ', ' || fname );
    END LOOP;
END;
/
```

Collections can be passed as parameters, so that subprograms can process arbitrary numbers of elements. You can use collections to move data into and out of database tables using high-performance language features known as bulk SQL.

For information on collections, see [Chapter 5, "Using PL/SQL Collections and Records"](#).

Records

Records are composite data structures whose fields can have different datatypes. You can use records to hold related items and pass them to subprograms with a single parameter. When declaring records, you use a `TYPE` definition. See ["Defining and Declaring Records"](#) on page 5-31.

[Example 1-17](#) shows how records are declared.

Example 1-17 Declaring a Record Type

```
DECLARE
    TYPE timerec IS RECORD (hours SMALLINT, minutes SMALLINT);
    TYPE meetin_typ IS RECORD (
        date_held DATE,
        duration timerec, -- nested record
        location VARCHAR2(20),
        purpose VARCHAR2(50));
BEGIN
    -- NULL does nothing but allows unit to be compiled and tested
    NULL;
END;
/
```

You can use the `%ROWTYPE` attribute to declare a record that represents a row in a table or a row from a query result set, without specifying the names and types for the fields.

For information on records, see [Chapter 5, "Using PL/SQL Collections and Records"](#).

Object Types

PL/SQL supports object-oriented programming through object types. An object type encapsulates a data structure along with the subprograms needed to manipulate the data. The variables that form the data structure are known as attributes. The subprograms that manipulate the attributes are known as methods.

Object types reduce complexity by breaking down a large system into logical entities. This lets you create software components that are modular, maintainable, and reusable. Object-type definitions, and the code for the methods, are stored in the database. Instances of these object types can be stored in tables or used as variables inside PL/SQL code. [Example 1-18](#) shows an object type definition for a bank account.

Example 1-18 Defining an Object Type

```
CREATE TYPE bank_account AS OBJECT (
    acct_number NUMBER(5),
    balance NUMBER,
    status VARCHAR2(10),
    MEMBER PROCEDURE open
        (SELF IN OUT NOCOPY bank_account, amount IN NUMBER),
    MEMBER PROCEDURE close (SELF IN OUT NOCOPY bank_account,
        num IN NUMBER, amount OUT NUMBER),
    MEMBER PROCEDURE deposit (SELF IN OUT NOCOPY bank_account,
```

```

                                num IN NUMBER, amount IN NUMBER),
MEMBER PROCEDURE withdraw (SELF IN OUT NOCOPY bank_account,
                                num IN NUMBER, amount IN NUMBER),
MEMBER FUNCTION curr_bal (num IN NUMBER) RETURN NUMBER);
/

```

For information on object types, see *Oracle Database Object-Relational Developer's Guide*.

Understanding PL/SQL Error Handling

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

PL/SQL's exception handling is different from the manual checking you might be used to from C programming, where you insert a check to make sure that every operation succeeded. Instead, the checks and calls to error routines are performed automatically, similar to the exception mechanism in Java programming.

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

You can declare exceptions of your own, for conditions that you decide are errors, or to correspond to database errors that normally result in ORA- error messages. When you detect a user-defined error condition, you execute a `RAISE` statement. See the exception `comm_missing` in [Example 1-13](#) on page 1-14. In the example, if the commission is null, the exception `comm_missing` is raised.

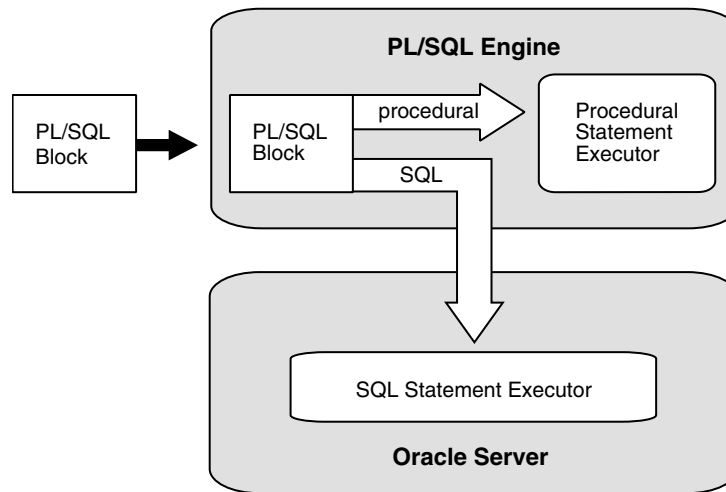
Typically, you put an exception handler at the end of a subprogram to handle exceptions that are raised anywhere inside the subprogram. To continue executing from the spot where an exception happens, enclose the code that might raise an exception inside another `BEGIN-END` block with its own exception handler. For example, you might put separate `BEGIN-END` blocks around groups of SQL statements that might raise `NO_DATA_FOUND`, or around arithmetic operations that might raise `DIVIDE_BY_ZERO`. By putting a `BEGIN-END` block with an exception handler inside a loop, you can continue executing the loop even if some loop iterations raise exceptions. See [Example 5-38](#) on page 5-28.

For information on PL/SQL errors, see "[Overview of PL/SQL Run-Time Error Handling](#)" on page 11-1. For information on PL/SQL warnings, see "[Overview of PL/SQL Compile-Time Warnings](#)" on page 11-18.

PL/SQL Architecture

The PL/SQL compilation and run-time system is an engine that compiles and executes PL/SQL blocks and subprograms. The engine can be installed in an Oracle server or in an application development tool such as Oracle Forms.

In either environment, the PL/SQL engine accepts as input any valid PL/SQL block or subprogram. [Figure 1-2](#) shows the PL/SQL engine processing an anonymous block. The PL/SQL engine executes procedural statements but sends SQL statements to the SQL engine in the Oracle database.

Figure 1–2 PL/SQL Engine

Topics:

- [PL/SQL Architecture in Oracle Database Server](#)
- [PL/SQL Architecture in Oracle Tools](#)

PL/SQL Architecture in Oracle Database Server

Typically, the Oracle database server processes PL/SQL blocks and subprograms.

Topics:

- [Anonymous Blocks](#)
- [Stored Subprograms](#)
- [Database Triggers](#)

Anonymous Blocks

Anonymous PL/SQL blocks can be submitted to interactive tools such as SQL*Plus and Enterprise Manager, or embedded in an Oracle Precompiler or OCI program. At run time, the program sends these blocks to the Oracle database, where they are compiled and executed.

Stored Subprograms

Subprograms can be compiled and stored in an Oracle database, ready to be executed. Once compiled, it is a schema object known as a stored subprogram, which can be referenced by any number of applications connected to that database.

The SQL `CREATE PROCEDURE` statement lets you create standalone procedures that are stored in the database. For information, see `CREATE PROCEDURE` in *Oracle Database SQL Language Reference*. The SQL `CREATE FUNCTION` statement lets you create standalone functions that are stored in an Oracle database. For information, see `CREATE FUNCTION` in *Oracle Database SQL Language Reference*.

Subprograms are stored in a compact compiled form. When invoked, they are loaded and processed immediately. Subprograms take advantage of shared memory, so that only one copy of a subprogram is loaded into memory for execution by multiple users.

Stored subprograms defined within a package are known as packaged subprograms. Those defined independently are called standalone subprograms. Subprograms nested inside other subprograms or within a PL/SQL block are known as local subprograms, which cannot be referenced by other applications and exist only inside the enclosing block.

Stored subprograms are the key to modular, reusable PL/SQL code. Wherever you might use a JAR file in Java, a module in Perl, a shared library in C++, or a DLL in Visual Basic, you can use PL/SQL stored subprograms and packages.

You can invoke stored subprograms from a database trigger, another stored subprogram, an Oracle Precompiler or OCI application, or interactively from SQL*Plus or Enterprise Manager. You can also configure a web server so that the HTML for a web page is generated by a stored subprogram, making it simple to provide a web interface for data entry and report generation.

[Example 1–19](#) shows how you can invoke the stored subprogram in [Example 1–13](#) from SQL*Plus using the CALL statement or using a BEGIN–END block.

Example 1–19 Techniques for Invoking a Standalone Procedure from SQL*Plus

```
CALL award_bonus(179, 1000);

BEGIN
    award_bonus(179, 10000);
END;
/
-- using named notation
BEGIN award_bonus(emp_id=>179, bonus=>10000); END;
/
```

Using the BEGIN–END block is recommended in several situations. For example, using the CALL statement can suppress an ORA-01403 : no data found error that was not handled in the PL/SQL subprogram.

For additional examples of invoking PL/SQL subprograms, see [Example 8–5](#) on page 8-8. For information on the use of the CALL statement, see *Oracle Database SQL Language Reference*

Database Triggers

A database trigger is a stored subprogram associated with a database table, view, or event. The trigger can be invoked once, when some event occurs, or many times, once for each row affected by an INSERT, UPDATE, or DELETE statement. The trigger can be invoked after the event, to record it or take some followup action. Or, the trigger can be invoked before the event to prevent erroneous operations or fix new data so that it conforms to business rules. In [Example 1–20](#) the table-level trigger fires whenever salaries in the employees table are updated, such as the processing in [Example 1–8](#) on page 1-10. For each update, the trigger writes a record to the emp_audit table.

Example 1–20 Creating a Database Trigger

```
CREATE TABLE emp_audit ( emp_audit_id NUMBER(6), up_date DATE,
                        new_sal NUMBER(8,2), old_sal NUMBER(8,2) );

CREATE OR REPLACE TRIGGER audit_sal
    AFTER UPDATE OF salary ON employees FOR EACH ROW
BEGIN
    -- bind variables are used here for values
    INSERT INTO emp_audit VALUES( :old.employee_id, SYSDATE,
```

```
                                :new.salary, :old.salary );  
END;  
/
```

The executable part of a trigger can contain procedural statements as well as SQL data manipulation statements. Besides table-level triggers, there are instead-of triggers for views and system-event triggers for schemas. For more information on triggers, see [Chapter 9, "Using Triggers"](#). For information on the CREATE TRIGGER SQL statement, see *Oracle Database SQL Language Reference*.

PL/SQL Architecture in Oracle Tools

An application development tool that contains the PL/SQL engine can process PL/SQL blocks and subprograms. The tool passes the blocks to its local PL/SQL engine. The engine executes all procedural statements inside the application and sends only SQL statements to the database. Most of the work is done inside the application, not on the database server. If the block contains no SQL statements, the application executes the entire block. This is useful if your application can benefit from conditional and iterative control.

Frequently, Oracle Forms applications use SQL statements to test the value of field entries or to do simple computations. By using PL/SQL instead, you can avoid calls to the database. You can also use PL/SQL functions to manipulate field entries.

PL/SQL Language Fundamentals

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

- [Character Sets and Lexical Units](#)
- [Declarations](#)
- [Naming Conventions](#)
- [Scope and Visibility of PL/SQL Identifiers](#)
- [Assigning Values to Variables](#)
- [PL/SQL Expressions and Comparisons](#)
- [PL/SQL Error-Reporting Functions](#)
- [Using SQL Functions in PL/SQL](#)
- [Conditional Compilation](#)
- [Using PL/SQL to Create Web Applications and Server Pages](#)

Character Sets and Lexical Units

PL/SQL supports two character sets: the **database character set**, which is used for identifiers and source code, and the **national character set**, which is used for national language data. This topic applies only to the database character set. For information about the national character set, see "[NCHAR and NVARCHAR2 Datatypes](#)" on page 3-12.

PL/SQL programs are written as lines of text using the following characters:

- Upper- and lower-case letters A .. Z and a .. z
- Numerals 0 .. 9
- Symbols () + - * / < > = ! ~ ^ ; : . ' @ % , " # \$ & _ | { } ? []
- Tabs, spaces, and carriage returns

PL/SQL keywords are not case-sensitive, so lower-case letters are equivalent to corresponding upper-case letters except within string and character literals.

A line of PL/SQL text contains groups of characters known as lexical units:

- Delimiters (simple and compound symbols)
- Identifiers, which include reserved words
- Literals
- Comments

To improve readability, you can separate lexical units by spaces. In fact, you must separate adjacent identifiers by a space or punctuation. The following line is not allowed because the reserved words `END` and `IF` are joined:

```
IF x > y THEN high := x; ENDIF; -- not allowed, must be END IF
```

You cannot embed spaces inside lexical units except for string literals and comments. For example, the following line is not allowed because the compound symbol for assignment (`:=`) is split:

```
count : = count + 1; -- not allowed, must be :=
```

To show structure, you can split lines using carriage returns, and indent lines using spaces or tabs. This formatting makes the first `IF` statement more readable.

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

The following is easier to read:

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

Topics:

- [Delimiters](#)
- [Identifiers](#)
- [Literals](#)
- [Comments](#)

Delimiters

A delimiter is a simple or compound symbol that has a special meaning to PL/SQL. For example, you use delimiters to represent arithmetic operations such as addition and subtraction. [Table 2–1](#) contains a list of PL/SQL delimiters.

Table 2–1 PL/SQL Delimiters

Symbol	Meaning
+	addition operator
%	attribute indicator
'	character string delimiter
.	component selector
/	division operator
(expression or list delimiter
)	expression or list delimiter
:	host variable indicator
,	item separator
*	multiplication operator
"	quoted identifier delimiter
=	relational operator

Table 2–1 (Cont.) PL/SQL Delimiters

Symbol	Meaning
<	relational operator
>	relational operator
@	remote access indicator
;	statement terminator
-	subtraction/negation operator
:=	assignment operator
=>	association operator
	concatenation operator
**	exponentiation operator
<<	label delimiter (begin)
>>	label delimiter (end)
/*	multi-line comment delimiter (begin)
*/	multi-line comment delimiter (end)
..	range operator
<>	relational operator
!=	relational operator
~=	relational operator
^=	relational operator
<=	relational operator
>=	relational operator
--	single-line comment indicator

Identifiers

You use identifiers to name PL/SQL program items and units, which include constants, variables, exceptions, cursors, cursor variables, subprograms, and packages. Some examples of identifiers follow:

```
X
t2
phone#
credit_limit
LastName
oracle$number
```

An identifier consists of a letter optionally followed by more letters, numerals, dollar signs, underscores, and number signs. Other characters such as hyphens, slashes, and spaces are not allowed, as the following examples show:

```
mine&yours -- not allowed because of the ampersand (&)
debit-amount -- not allowed because of the hyphen (-)
on/off -- not allowed because of the slash (/)
user id -- not allowed because of the space
```

Adjoining and trailing dollar signs, underscores, and number signs are allowed:

```
money$$$tree
```

```
SN##  
try_again_
```

You can use upper, lower, or mixed case to write identifiers. PL/SQL is not case sensitive except within string and character literals. If the only difference between identifiers is the case of corresponding letters, PL/SQL considers them the same:

```
lastname  
LastName  
LASTNAME
```

The size of an identifier cannot exceed 30 characters. Every character, including dollar signs, underscores, and number signs, is significant. For example, PL/SQL considers the following identifiers to be different:

```
lastname  
last_name
```

Make identifiers descriptive. Avoid obscure names such as `cpm`. Instead, use meaningful names such as `cost_per_thousand`.

Topics:

- [Reserved Words](#)
- [Predefined Identifiers](#)
- [Quoted Identifiers](#)

Reserved Words

Some identifiers, called reserved words, have a special syntactic meaning to PL/SQL. For example, the words `BEGIN` and `END` are reserved. Often, reserved words are written in upper case for readability.

Trying to redefine a reserved word causes a compilation error. Instead, you can embed reserved words as part of a longer identifier. For example:

```
DECLARE  
-- end BOOLEAN; the use of "end" is not allowed;  
-- causes compilation error  
  end_of_game BOOLEAN; -- allowed
```

In addition to reserved words, there are keywords that have special meaning in PL/SQL. PL/SQL keywords can be used for identifiers, but this is not recommended. For a list of PL/SQL reserved words and keywords, see [Table D-1](#) on page D-1 and [Table D-2](#) on page D-2.

Predefined Identifiers

Identifiers globally declared in package `STANDARD`, such as the exception `INVALID_NUMBER`, can be redeclared. However, redeclaring predefined identifiers is error prone because your local declaration overrides the global declaration.

Quoted Identifiers

For flexibility, PL/SQL lets you enclose identifiers within double quotes. Quoted identifiers are seldom needed, but occasionally they can be useful. They can contain any sequence of printable characters including spaces but excluding double quotes. Thus, the following identifiers are valid:

```
"X+Y"
```

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

The maximum size of a quoted identifier is 30 characters not counting the double quotes. Though allowed, using PL/SQL reserved words as quoted identifiers is a poor programming practice.

Literals

A literal is an explicit numeric, character, string, or `BOOLEAN` value not represented by an identifier. The numeric literal `147` and the `BOOLEAN` literal `FALSE` are examples. For information on the PL/SQL datatypes, see ["Predefined PL/SQL Scalar Datatypes and Subtypes"](#) on page 3-1.

Topics:

- [Numeric Literals](#)
- [Character Literals](#)
- [String Literals](#)
- [BOOLEAN Literals](#)
- [Date and Time Literals](#)

Numeric Literals

Two kinds of numeric literals can be used in arithmetic expressions: integers and reals. An integer literal is an optionally signed whole number without a decimal point. For example:

```
030 6 -14 0 +32767
```

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

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

PL/SQL considers numbers such as `12.0` and `25.` to be reals even though they have integral values.

A numeric literal value that is composed only of digits and falls in the range `-2147483648` to `2147483647` has a `PLS_INTEGER` datatype; otherwise this literal has the `NUMBER` datatype. You can add the `f` or `d` suffix to a literal value that is composed only of digits to specify the `BINARY_FLOAT` or `BINARY_TABLE` respectively. For the properties of the datatypes, see ["Predefined PL/SQL Numeric Datatypes and Subtypes"](#) on page 3-2.

Numeric literals cannot contain dollar signs or commas, but can be written using scientific notation. Simply suffix the number with an `E` (or `e`) followed by an optionally signed integer. For example:

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

`xEy` stands for "x times ten to the power of y." As the next example shows, the number after `E` is the power of ten by which the number before `E` is multiplied (the double asterisk `**`) is the exponentiation operator):

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

The number after E also corresponds to the number of places the decimal point shifts. In the last example, the implicit decimal point shifted three places to the right. In the following example, it shifts three places to the left:

```
5E-3 = 5 * 10**-3 = 5 * 0.001 = 0.005
```

The absolute value of a NUMBER literal can be in the range 1.0E-130 up to (but not including) 1.0E126. The literal can also be 0. For information on results outside the valid range, see "NUMBER Datatype" on page 3-6.

Example 2-1 NUMBER Literals

```
DECLARE
  n NUMBER;
BEGIN
  n := -9.999999E-130; -- valid
  n := 9.999E125;     -- valid
  -- n := 10.0E125;    -- invalid, "numeric overflow or underflow"
END;
/
```

Real literals can also use the trailing letters f and d to specify the types BINARY_FLOAT and BINARY_DOUBLE, as shown in [Example 2-2](#).

Example 2-2 Using BINARY_FLOAT and BINARY_DOUBLE

```
DECLARE
  x BINARY_FLOAT := sqrt(2.0f);
  -- single-precision floating-point number
  y BINARY_DOUBLE := sqrt(2.0d);
  -- double-precision floating-point number
BEGIN
  NULL;
END;
/
```

Character Literals

A character literal is an individual character enclosed by single quotes (apostrophes). Character literals include all the printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols. Some examples follow:

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

PL/SQL is case sensitive within character literals. For example, PL/SQL considers the literals 'Z' and 'z' to be different. Also, the character literals '0'..'9' are not equivalent to integer literals but can be used in arithmetic expressions because they are implicitly convertible to integers.

String Literals

A character value can be represented by an identifier or explicitly written as a string literal, which is a sequence of zero or more characters enclosed by single quotes. All string literals except the null string (') have datatype CHAR.

The following are examples of string literals:

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



```
'$1,000,000'
```

PL/SQL is case sensitive within string literals. For example, PL/SQL considers the following literals to be different:

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

To represent an apostrophe within a string, you can write two single quotes, which is not the same as writing a double quote:

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

You can also use the following notation to define your own delimiter characters for the literal. You choose a character that is not present in the string, and then do not need to escape other single quotation marks inside the literal:

```
-- q'!...!' notation allows use of single quotes inside literal
string_var := q'!I'm a string, you're a string.!';
```

You can use delimiters [, {, <, and (, pair them with], }, >, and), pass a string literal representing a SQL statement to a subprogram, without doubling the quotation marks around 'INVALID' as follows:

```
func_call(q'[SELECT index_name FROM user_indexes
WHERE status = 'INVALID']');
```

For NCHAR and NVARCHAR2 literals, use the prefix `nq` instead of `q`, as in the following example, where `00E0` represents the character `é`:

```
where_clause := nq'#WHERE COL_VALUE LIKE '%\00E9'#';
```

For more information about the NCHAR datatype and unicode strings, see *Oracle Database Globalization Support Guide*.

BOOLEAN Literals

BOOLEAN literals are the predefined values TRUE, FALSE, and NULL. NULL stands for a missing, unknown, or inapplicable value. Remember, BOOLEAN literals are values, not strings. For example, TRUE is no less a value than the number 25.

Date and Time Literals

Datetime literals have various formats depending on the datatype. For example:

Example 2-3 Using DateTime Literals

```
DECLARE
  d1 DATE := DATE '1998-12-25';
  t1 TIMESTAMP := TIMESTAMP '1997-10-22 13:01:01';
  t2 TIMESTAMP WITH TIME ZONE :=
    TIMESTAMP '1997-01-31 09:26:56.66 +02:00';
-- Three years and two months
-- For greater precision, use the day-to-second interval
i1 INTERVAL YEAR TO MONTH := INTERVAL '3-2' YEAR TO MONTH;
-- Five days, four hours, three minutes, two and 1/100 seconds
i2 INTERVAL DAY TO SECOND :=
  INTERVAL '5 04:03:02.01' DAY TO SECOND;
```

You can also specify whether a given interval value is YEAR TO MONTH or DAY TO SECOND. For example, `current_timestamp - current_timestamp` produces a

value of type `INTERVAL DAY TO SECOND` by default. You can specify the type of the interval using the following formats:

- `(interval_expression) DAY TO SECOND`
- `(interval_expression) YEAR TO MONTH`

For details on the syntax for the date and time types, see the *Oracle Database SQL Language Reference*. For examples of performing date and time arithmetic, see *Oracle Database Advanced Application Developer's Guide*.

Comments

The PL/SQL compiler ignores comments. Adding comments to your program promotes readability and aids understanding. Generally, you use comments to describe the purpose and use of each code segment. PL/SQL supports two comment styles: single-line and multi-line.

Topics:

- [Single-Line Comments](#)
- [Multiline Comments](#)
- [Restrictions on Comments](#)

Single-Line Comments

Single-line comments begin with a double hyphen (`--`) anywhere on a line and extend to the end of the line. A few examples follow:

Example 2-4 Using 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 some other value
END;
/
```

Notice that comments can appear within a statement at the end of a line.

While testing or debugging a program, you might want to disable a line of code by making it a comment, as in the following example:

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

Multiline Comments

Multi-line comments begin with a slash-asterisk (`/*`), end with an asterisk-slash (`*/`), and can span multiple lines, as shown in [Example 2-5](#). You can use multi-line comment delimiters to comment-out whole sections of code.

Example 2-5 Using Multi-Line 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;
    /* The following 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;
/

```

Restrictions on Comments

You cannot nest comments. You cannot use single-line comments in a PL/SQL block that will be processed by an Oracle Precompiler program because end-of-line characters are ignored. As a result, single-line comments extend to the end of the block, not just to the end of a line. In this case, use the `/* */` notation instead.

Declarations

Your program stores values in variables and constants. As the program executes, the values of variables can change, but the values of constants cannot.

You can declare variables and constants in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage space for a value, specify its datatype, and name the storage location so that you can reference it.

Some examples follow:

```

DECLARE
    birthday DATE;
    emp_count SMALLINT := 0;

```

The first declaration names a variable of type `DATE`. The second declaration names a variable of type `SMALLINT` and uses the assignment operator to assign an initial value of zero to the variable.

The next examples show that the expression following the assignment operator can be arbitrarily complex and can refer to previously initialized variables:

```

DECLARE
    pi REAL := 3.14159;
    radius REAL := 1;
    area REAL := pi * radius**2;

```

By default, variables are initialized to `NULL`, so it is redundant to include `:= NULL` in a variable declaration.

Topics:

- [Constants](#)
- [Using DEFAULT](#)
- [Using NOT NULL](#)
- [Using the %TYPE Attribute](#)

- [Using the %ROWTYPE Attribute](#)
- [Restrictions on Declarations](#)

Constants

To declare a constant, put the keyword `CONSTANT` before the type specifier. The following declaration names a constant of type `REAL` and assigns an unchangeable value of 5000 to the constant. A constant must be initialized in its declaration. Otherwise, a compilation error occurs.

```
DECLARE
  credit_limit CONSTANT REAL := 5000.00;
  max_days_in_year CONSTANT INTEGER := 366;
  urban_legend CONSTANT BOOLEAN := FALSE;
```

Using DEFAULT

You can use the keyword `DEFAULT` instead of the assignment operator to initialize variables. For example, the following two declarations are equivalent:

```
blood_type CHAR := '0';
blood_type CHAR DEFAULT '0';
```

Use `DEFAULT` for variables that have a typical value. Use the assignment operator for variables (such as counters and accumulators) that have no typical value. For example:

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

You can also use `DEFAULT` to initialize subprogram parameters, cursor parameters, and fields in a user-defined record.

Using NOT NULL

Besides assigning an initial value, declarations can impose the `NOT NULL` constraint:

```
DECLARE
  acct_id INTEGER(4) NOT NULL := 9999;
```

You cannot assign nulls to a variable defined as `NOT NULL`. If you try, PL/SQL raises the predefined exception `VALUE_ERROR`.

The `NOT NULL` constraint must be followed by an initialization clause.

PL/SQL provide subtypes `NATURALN` and `POSITIVEN` that are predefined as `NOT NULL`. You can omit the `NOT NULL` constraint when declaring variables of these types, and you must include an initialization clause.

Using the %TYPE Attribute

The `%TYPE` attribute provides the datatype of a variable or database column. As shown in [Example 2-6](#), variables declared with `%TYPE` inherit the datatype of a variable, plus default values and constraints.

Example 2-6 Using %TYPE with the Datatype of a Variable

```
DECLARE
  credit PLS_INTEGER RANGE 1000..25000;
  debit credit%TYPE;
```

```

v_name VARCHAR2(20);
name   VARCHAR2(20) NOT NULL := 'John Smith';
-- Increasing length of name also lengthens following variables
upper_name name%TYPE := UPPER(name);
lower_name name%TYPE := LOWER(name);
init_name  name%TYPE := INITCAP(name);
BEGIN
-- Display inherited default values
DBMS_OUTPUT.PUT_LINE('name: ' || name || ' upper_name: '
  || upper_name || ' lower_name: ' || lower_name
  || ' init_name: ' || init_name);
-- lower_name := 'jonathan henry smithson';
-- invalid, character string is too long
-- lower_name := NULL; invalid, NOT NULL CONSTRAINT
-- debit := 50000; invalid, value out of range
END;
/

```

Note that variables declared using `%TYPE` are treated like those declared using a datatype specifier. For example, given the previous declarations, PL/SQL treats `debit` like a `PLS_INTEGER` variable. A `%TYPE` declaration can also include an initialization clause.

The `%TYPE` attribute is particularly useful when declaring variables that refer to database columns. You can reference a table and column, or you can reference an owner, table, and column, as in:

```

DECLARE
-- If the length of the column ever changes, this code
-- will use the new length automatically.
  the_trigger user_triggers.trigger_name%TYPE;

```

When you use `table_name.column_name.%TYPE` to declare a variable, you do not need to know the actual datatype, and attributes such as precision, scale, and length. If the database definition of the column changes, the datatype of the variable changes accordingly at run time. However, `%TYPE` variables do not inherit column constraints, such as the `NOT NULL` or check constraint, or default values. For example, even though the database column `empid` is defined as `NOT NULL` in [Example 2-7](#), you can assign a `NULL` to the variable `v_empid`.

Example 2-7 Using %TYPE with Table Columns

```

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

DECLARE
  v_empid   employees_temp.empid%TYPE;
  v_deptid  employees_temp.deptid%TYPE;
  v_deptname employees_temp.deptname%TYPE;
BEGIN
  v_empid := NULL; -- Works, null constraint not inherited
  -- v_empid := 10000002; -- Invalid, number precision too large
  v_deptid := 50; -- Works, check constraint not inherited
  -- Default value not inherited in following:
  DBMS_OUTPUT.PUT_LINE('v_deptname: ' || v_deptname);
END;

```

/

See ["Constraints and Default Values with Subtypes"](#) on page 3-25 for information on column constraints that are inherited by subtypes declared using %TYPE.

Using the %ROWTYPE Attribute

The %ROWTYPE attribute provides a record type that represents a row in a table or view. Columns in a row and corresponding fields in a record have the same names and datatypes. However, fields in a %ROWTYPE record do not inherit constraints, such as the NOT NULL or check constraint, or default values, as shown in [Example 2-8](#).

Example 2-8 Using %ROWTYPE with Table Rows

```
DECLARE
    emprec employees_temp%ROWTYPE;
BEGIN
    emprec.empid := NULL; -- Works, null constraint not inherited
    -- emprec.empid := 10000002; -- Invalid, number precision too large
    emprec.deptid := 50; -- Works, null constraint not inherited
    -- the default value is not inherited in the following
    DBMS_OUTPUT.PUT_LINE('emprec.deptname: ' || emprec.deptname);
END;
/
```

See Also: [Example 3-14](#) on page 3-25

The record can store an entire row of data selected from the table, or fetched from a cursor or strongly typed cursor variable as shown in [Example 2-9](#).

Example 2-9 Using the %ROWTYPE Attribute

```
DECLARE
-- %ROWTYPE can include all the columns in a table ...
    emp_rec employees%ROWTYPE;
-- ... or a subset of the columns, based on a cursor.
    CURSOR c1 IS
        SELECT department_id, department_name FROM departments;
    dept_rec c1%ROWTYPE;
-- Can even make a %ROWTYPE with columns from multiple tables.
    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;
BEGIN
-- We know EMP_REC can hold a row from the EMPLOYEES table.
    SELECT * INTO emp_rec FROM employees WHERE ROWNUM < 2;
-- We can refer to the fields of EMP_REC using column names
-- from the EMPLOYEES table.
    IF emp_rec.department_id = 20
        AND emp_rec.last_name = 'JOHNSON' THEN
        emp_rec.salary := emp_rec.salary * 1.15;
    END IF;
END;
/
```

Topics:

- [Aggregate Assignment](#)
- [Using Aliases](#)

Aggregate Assignment

Although a `%ROWTYPE` declaration cannot include an initialization clause, there are ways to assign values to all fields in a record at once. You can assign one record to another if their declarations refer to the same table or cursor. [Example 2–10](#) shows record assignments that are allowed.

Example 2–10 Assigning Values to a Record with a `%ROWTYPE` Declaration

```
DECLARE
    dept_rec1 departments%ROWTYPE;
    dept_rec2 departments%ROWTYPE;
    CURSOR c1 IS SELECT department_id, location_id FROM departments;
    dept_rec3 c1%ROWTYPE;
BEGIN
    dept_rec1 := dept_rec2; -- allowed
-- dept_rec2 refers to a table, dept_rec3 refers to a cursor
-- dept_rec2 := dept_rec3; -- not allowed
END;
/
```

You can assign a list of column values to a record by using the `SELECT` or `FETCH` statement, as the following example shows. The column names must appear in the order in which they were defined by the `CREATE TABLE` or `CREATE VIEW` statement.

```
DECLARE
    dept_rec departments%ROWTYPE;
BEGIN
    SELECT * INTO dept_rec FROM departments
        WHERE department_id = 30 and ROWNUM < 2;
END;
/
```

However, there is no constructor for a record type, so you cannot assign a list of column values to a record by using an assignment statement.

Using Aliases

Select-list items fetched from a cursor associated with `%ROWTYPE` must have simple names or, if they are expressions, must have aliases. [Example 2–11](#) uses an alias called `complete_name` to represent the concatenation of two columns:

Example 2–11 Using an Alias for Column Names

```
BEGIN
-- We assign an alias (complete_name) to the expression value, because
-- it has no column name.
    FOR item IN
        ( SELECT first_name || ' ' || last_name complete_name
          FROM employees WHERE ROWNUM < 11 )
    LOOP
-- Now we can refer to the field in the record using this alias.
        DBMS_OUTPUT.PUT_LINE('Employee name: ' || item.complete_name);
    END LOOP;
END;
/
```

Restrictions on Declarations

PL/SQL does not allow forward references. You must declare a variable or constant before referencing it in other statements, including other declarative statements.

PL/SQL does allow the forward declaration of subprograms. For more information, see "[Declaring Nested PL/SQL Subprograms](#)" on page 8-5.

Some languages enable you to declare a list of variables that have the same datatype. PL/SQL does not allow this. You must declare each variable separately:

```
DECLARE
-- Multiple declarations not allowed.
-- i, j, k, l SMALLINT;
-- Instead, declare each separately.
  i SMALLINT;
  j SMALLINT;
-- To save space, you can declare more than one on a line.
  k SMALLINT; l SMALLINT;
```

Naming Conventions

The same naming conventions apply to all PL/SQL program items and units including constants, variables, cursors, cursor variables, exceptions, procedures, functions, and packages. Names can be simple, qualified, remote, or both qualified and remote. For example, you might use the procedure name `raise_salary` in any of the following ways:

```
raise_salary(...);           -- simple
emp_actions.raise_salary(...); -- qualified
raise_salary@newyork(...);   -- remote
emp_actions.raise_salary@newyork(...); -- qualified and remote
```

In the first case, you simply use the procedure name. In the second case, you must qualify the name using dot notation because the procedure is stored in a package called `emp_actions`. In the third case, using the remote access indicator (`@`), you reference the database link `newyork` because the procedure is stored in a remote database. In the fourth case, you qualify the procedure name and reference a database link.

Topics:

- [Synonyms](#)
- [Scoping](#)
- [Case Sensitivity](#)
- [Name Resolution](#)

Synonyms

You can create synonyms to provide location transparency for remote schema objects such as tables, sequences, views, standalone subprograms, packages, and object types. However, you cannot create synonyms for items declared within subprograms or packages. That includes constants, variables, cursors, cursor variables, exceptions, and packaged subprograms.

Scoping

Within the same scope, all declared identifiers must be unique; even if their datatypes differ, variables and parameters cannot share the same name. In [Example 2–12](#), the second declaration is not allowed.

Example 2–12 Errors with Duplicate Identifiers in Same Scope

```
DECLARE
    valid_id BOOLEAN;
    valid_id VARCHAR2(5); -- not allowed, duplicate identifier
BEGIN
    -- The error occurs when the identifier is referenced,
    -- not in the declaration part.
    valid_id := FALSE; -- raises an error here
END;
/
```

For the scoping rules that apply to identifiers, see "[Scope and Visibility of PL/SQL Identifiers](#)" on page 2-16.

Case Sensitivity

Like all identifiers, the names of constants, variables, and parameters are not case sensitive. For example, PL/SQL considers the following names to be the same:

Example 2–13 Case Sensitivity of Identifiers

```
DECLARE
    zip_code INTEGER;
    Zip_Code INTEGER;
    -- duplicate identifier, despite Z/z case difference
BEGIN
    zip_code := 90120;
    -- raises error here because of duplicate identifiers
END;
/
```

Name Resolution

In potentially ambiguous SQL statements, the names of database columns take precedence over the names of local variables and formal parameters. For example, if a variable and a column with the same name are both used in a `WHERE` clause, SQL considers that both cases refer to the column.

To avoid ambiguity, add a prefix to the names of local variables and formal parameters, or use a block label to qualify references as shown in [Example 2–14](#).

Example 2–14 Using a Block Label for Name Resolution

```
CREATE TABLE employees2 AS SELECT last_name FROM employees;
<<main>>
DECLARE
    last_name VARCHAR2(10) := 'King';
    v_last_name VARCHAR2(10) := 'King';
BEGIN
    -- deletes everyone, because both LAST_NAMES refer to the column
    DELETE FROM employees2 WHERE last_name = last_name;
    DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows.');
```

```
ROLLBACK;
```

```

-- OK, column and variable have different names
DELETE FROM employees2 WHERE last_name = v_last_name;
DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows. ');
ROLLBACK;
-- OK, block name specifies that 2nd last_name is a variable
DELETE FROM employees2 WHERE last_name = main.last_name;
DBMS_OUTPUT.PUT_LINE('Deleted ' || SQL%ROWCOUNT || ' rows. ');
ROLLBACK;
END;
/

```

[Example 2–15](#) shows that you can use a subprogram name to qualify references to local variables and formal parameters.

Example 2–15 Using a 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
-- DEPT_NAME.department_name specifies the local variable
-- instead of the table column
        SELECT department_name INTO dept_name.department_name
            FROM departments
            WHERE department_id = dept_name.department_id;
        RETURN department_name;
    END;
BEGIN
    FOR item IN (SELECT department_id FROM departments)
    LOOP
        DBMS_OUTPUT.PUT_LINE
            ('Department: ' || dept_name(item.department_id));
    END LOOP;
END;
/

```

For more information about name resolution, see [Appendix B, "How PL/SQL Resolves Identifier Names"](#).

Scope and Visibility of PL/SQL Identifiers

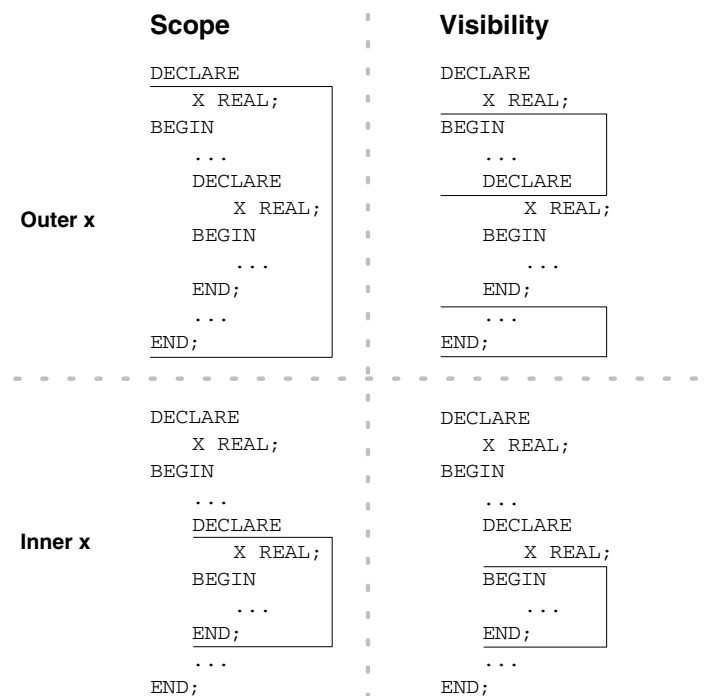
References to an identifier are resolved according to its scope and visibility. The *scope* of an identifier is that region of a program unit (block, subprogram, or package) from which you can reference the identifier. An identifier is visible only in the regions from which you can reference the identifier using an unqualified name. [Figure 2–1](#) shows the scope and visibility of a variable named *x*, which is declared in an enclosing block, then redeclared in a sub-block.

Identifiers declared in a PL/SQL block are considered local to that block and global to all its sub-blocks. If a global identifier is redeclared in a sub-block, both identifiers remain in scope. Within the sub-block, however, only the local identifier is visible because you must use a qualified name to reference the global identifier.

Although you cannot declare an identifier twice in the same block, you can declare the same identifier in two different blocks. The two items represented by the identifier are distinct, and any change in one does not affect the other. However, a block cannot

reference identifiers declared in other blocks at the same level because those identifiers are neither local nor global to the block.

Figure 2–1 Scope and Visibility



[Example 2–16](#) illustrates the scope rules. Notice that the identifiers declared in one sub-block cannot be referenced in the other sub-block. That is because a block cannot reference identifiers declared in other blocks nested at the same level.

Example 2–16 Scope Rules

```

DECLARE
  a CHAR;
  b REAL;
BEGIN
  -- identifiers available here: a (CHAR), b
  DECLARE
    a INTEGER;
    c REAL;
  BEGIN
    NULL; -- identifiers available here: a (INTEGER), b, c
  END;
  DECLARE
    d REAL;
  BEGIN
    NULL; -- identifiers available here: a (CHAR), b, d
  END;
  -- identifiers available here: a (CHAR), b
END;
/
```

Recall that global identifiers can be redeclared in a sub-block, in which case the local declaration prevails and the sub-block cannot reference the global identifier unless you

use a qualified name. The qualifier can be the label of an enclosing block as shown in [Example 2–17](#).

Example 2–17 Using a Label Qualifier with Identifiers

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

As [Example 2–18](#) shows, the qualifier can also be the name of an enclosing subprogram:

Example 2–18 Using Subprogram Qualifier with Identifiers

```
CREATE OR REPLACE PROCEDURE check_credit(limit NUMBER) AS
    rating NUMBER := 3;
    FUNCTION check_rating RETURN BOOLEAN IS
        rating          NUMBER := 1;
        over_limit      BOOLEAN;
    BEGIN
        IF check_credit.rating <= limit THEN
            over_limit := FALSE;
        ELSE
            rating := limit;
            over_limit := TRUE;
        END IF;
        RETURN over_limit;
    END check_rating;
BEGIN
    IF check_rating THEN
        DBMS_OUTPUT.PUT_LINE
            ('Credit rating over limit (' || TO_CHAR(limit)
            || '). ' || ' Rating: ' || TO_CHAR(rating));
    ELSE
        DBMS_OUTPUT.PUT_LINE
            ('Credit rating OK. ' || ' Rating: '
            || TO_CHAR(rating) );
    END IF;
END;
/

CALL check_credit(1);
```

However, within the same scope, a label and a subprogram cannot have the same name. Avoid duplicate labels, like those in [Example 2–19](#).

Example 2-19 PL/SQL Block Using Multiple and Duplicate Labels

```

<<compute_ratio>>
<<another_label>>
DECLARE
    numerator    NUMBER := 22;
    denominator  NUMBER := 7;
    the_ratio    NUMBER;
BEGIN
    <<inner_label>>
    <<another_label>>
DECLARE
    denominator NUMBER := 0;
BEGIN
    -- first use the denominator value = 7 from global DECLARE
    -- to compute a rough value of pi
    the_ratio := numerator/compute_ratio.denominator;
    DBMS_OUTPUT.PUT_LINE('Ratio = ' || the_ratio);
    -- use local denominator value = 0 to raise exception
    -- inner_label is not needed but used for clarification
    the_ratio := numerator/inner_label.denominator;
    DBMS_OUTPUT.PUT_LINE('Ratio = ' || the_ratio);
    -- if you use a duplicate label, you might get errors
    -- or unpredictable results
    the_ratio := numerator/another_label.denominator;
    DBMS_OUTPUT.PUT_LINE('Ratio = ' || the_ratio);
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 inner_label;

END compute_ratio;

/

Assigning Values to Variables

You can use assignment statements to assign values to variables. For example, the following statement assigns a new value to the variable `bonus`, overwriting its old value:

```
bonus := salary * 0.15;
```

Variables and constants are initialized every time a block or subprogram is entered. By default, variables are initialized to `NULL`. Unless you expressly initialize a variable, its value is undefined (`NULL`) as shown in [Example 2-20](#).

Example 2-20 Initialization of Variables and Constants

```

DECLARE
    counter INTEGER;
BEGIN
    -- COUNTER is initially NULL, so 'COUNTER + 1' is also null.
    counter := counter + 1;
    IF counter IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('COUNTER is NULL not 1.');
```

END IF;

END;

/

To avoid unexpected results, never reference a variable before you assign it a value. The expression following the assignment operator can be arbitrarily complex, but it must yield a datatype that is the same as or convertible to the datatype of the variable.

Topics:

- [Assigning BOOLEAN Values](#)
- [Assigning SQL Query Results to PL/SQL Variables](#)

Assigning BOOLEAN Values

Only the values TRUE, FALSE, and NULL can be assigned to a BOOLEAN variable as shown in [Example 2-21](#). You can assign these literal values, or expressions such as comparisons using relational operators.

Example 2-21 *Assigning BOOLEAN Values*

```
DECLARE
  done BOOLEAN; -- DONE is initially NULL
  counter NUMBER := 0;
BEGIN
  done := FALSE; -- Assign a literal value
  WHILE done != TRUE -- Compare to a literal value
  LOOP
    counter := counter + 1;
    done := (counter > 500); -- If counter > 500, DONE = TRUE
  END LOOP;
END;
/
```

Assigning SQL Query Results to PL/SQL Variables

You can use the SELECT statement to have Oracle assign values to a variable. For each item in the select list, there must be a corresponding, type-compatible variable in the INTO list as shown in [Example 2-22](#).

Example 2-22 *Assigning a Query Result to a Variable*

```
DECLARE
  emp_id employees.employee_id%TYPE := 100;
  emp_name employees.last_name%TYPE;
  wages NUMBER(7,2);
BEGIN
  SELECT last_name, salary + (salary * nvl(commission_pct,0))
     INTO emp_name, wages FROM employees
     WHERE employee_id = emp_id;
  DBMS_OUTPUT.PUT_LINE
    ('Employee ' || emp_name || ' might make ' || wages);
END;
/
```

Because SQL does not have a BOOLEAN type, you cannot select column values into a BOOLEAN variable. For additional information on assigning variables with the DML statements, including situations when the value of a variable is undefined, see "[Data Manipulation Language \(DML\) Statements](#)" on page 6-1.

PL/SQL Expressions and Comparisons

PL/SQL expressions are constructed from operands and operators. An operand is a variable, constant, literal, placeholder, or function call. A PL/SQL operator is either unary or binary. A unary operator operates on a single operand. A binary operator operates on two operands. An example of a unary operator is negation (-). An example of a binary operator is addition (+).

An example of a simple arithmetic expression is:

$$-x / 2 + 3$$

The simplest expression consists of a single variable, which yields a value directly. PL/SQL evaluates an expression by combining the values of the operands in ways specified by the operators. An expression always returns a single value. PL/SQL determines the datatype of this value by examining the expression and the context in which it appears.

Topics:

- [Operator Precedence](#)
- [Logical Operators](#)
- [BOOLEAN Expressions](#)
- [CASE Expressions](#)
- [Handling Null Values in Comparisons and Conditional Statements](#)

Operator Precedence

The operations within an expression are done in a particular order depending on their precedence (priority). [Table 2-2](#) shows the default order of operations from first to last (top to bottom).

Table 2-2 Order of Operations

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

Operators with higher precedence are applied first. In the following example, both expressions yield 8 because division has a higher precedence than addition. Operators with the same precedence are applied in no particular order.

$$5 + 12 / 4$$

$$12 / 4 + 5$$

You can use parentheses to control the order of evaluation. For example, the following expression yields 7, not 11, because parentheses override the default operator precedence:

```
(8 + 6) / 2
```

In the next example, the subtraction is done before the division because the most deeply nested subexpression is always evaluated first:

```
100 + (20 / 5 + (7 - 3))
```

The following example shows that you can always use parentheses to improve readability, even when they are not needed:

```
(salary * 0.05) + (commission * 0.25)
```

Logical Operators

The logical operators AND, OR, and NOT follow the tri-state logic shown in [Table 2-3](#). AND and OR are binary operators; NOT is a unary operator.

Table 2-3 Logic Truth Table

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

As the truth table shows, AND returns TRUE only if both its operands are true. On the other hand, OR returns TRUE if either of its operands is true. NOT returns the opposite value (logical negation) of its operand. For example, NOT TRUE returns FALSE.

NOT NULL returns NULL, because nulls are indeterminate. Be careful to avoid unexpected results in expressions involving nulls; see ["Handling Null Values in Comparisons and Conditional Statements"](#) on page 2-30.

Topics:

- [Order of Evaluation](#)
- [Short-Circuit Evaluation](#)
- [Comparison Operators](#)
- [Relational Operators](#)
- [IS NULL Operator](#)
- [LIKE Operator](#)
- [BETWEEN Operator](#)

- [IN Operator](#)
- [Concatenation Operator](#)

Order of Evaluation

When you do not use parentheses to specify the order of evaluation, operator precedence determines the order. Compare the following expressions:

```
NOT (valid AND done) | NOT valid AND done
```

If the `BOOLEAN` variables `valid` and `done` have the value `FALSE`, the first expression yields `TRUE`. However, the second expression yields `FALSE` because `NOT` has a higher precedence than `AND`. Therefore, the second expression is equivalent to:

```
(NOT valid) AND done
```

In the following example, notice that when `valid` has the value `FALSE`, the whole expression yields `FALSE` regardless of the value of `done`:

```
valid AND done
```

Likewise, in the next example, when `valid` has the value `TRUE`, the whole expression yields `TRUE` regardless of the value of `done`:

```
valid OR done
```

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 the result can be determined. This lets you write expressions that might otherwise cause an error. Consider the `OR` expression in [Example 2-23](#).

Example 2-23 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
        DEMS_OUTPUT.PUT_LINE('On hand quantity is zero.');
```

When the value of `on_hand` is zero, the left operand yields `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.

Short-circuit evaluation applies to `IF` statements, `CASE` statements, and `CASE` expressions in PL/SQL.

Comparison Operators

Comparison operators compare one expression to another. The result is always true, false, or null. Typically, you use comparison operators in conditional control statements and in the `WHERE` clause of SQL data manipulation statements.

[Example 2-24](#) provides some examples of comparisons for different types.

Example 2–24 Using Comparison Operators

```

DECLARE
  PROCEDURE assert(assertion VARCHAR2, truth BOOLEAN)
  IS
  BEGIN
    IF truth IS NULL THEN
      DBMS_OUTPUT.PUT_LINE
        ('Assertion ' || assertion || ' is unknown (NULL)');
    ELSIF truth = TRUE THEN
      DBMS_OUTPUT.PUT_LINE
        ('Assertion ' || assertion || ' is TRUE');
    ELSE
      DBMS_OUTPUT.PUT_LINE
        ('Assertion ' || assertion || ' is FALSE');
    END IF;
  END;
BEGIN
  assert('2 + 2 = 4', 2 + 2 = 4);
  assert('10 > 1', 10 > 1);
  assert('10 <= 1', 10 <= 1);
  assert('5 BETWEEN 1 AND 10', 5 BETWEEN 1 AND 10);
  assert('NULL != 0', NULL != 0);
  assert('3 IN (1,3,5)', 3 IN (1,3,5));
  assert('A < Z', 'A' < 'Z');
  assert('baseball' LIKE '%all%', 'baseball' LIKE '%all%');
  assert('suit' || 'case' = 'suitcase',);
  'suit' || 'case' = 'suitcase');
END;
/

```

Relational Operators

The relational operators with their meanings are:

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

IS NULL Operator

The IS NULL operator returns the BOOLEAN value TRUE if its operand is null or FALSE if it is not null. Comparisons involving nulls always yield NULL. Test whether a value is null as follows:

```
IF variable IS NULL THEN ...
```

LIKE Operator

You use the LIKE operator to compare a character, string, or CLOB value to a pattern. Case is significant. LIKE returns the BOOLEAN value TRUE if the patterns match or FALSE if they do not match.

The patterns matched by `LIKE` can include two special-purpose characters called wildcards. An underscore (`_`) matches exactly one character; a percent sign (`%`) matches zero or more characters. For example, if the value of `last_name` is `'JOHNSON'`, the following expression is true:

```
last_name LIKE 'J%S_N'
```

To search for the percent sign and underscore characters, you define an escape character and put that character before the percent sign or underscore. The following example uses the backslash as the escape character, so that the percent sign in the string does not act as a wildcard:

```
IF sale_sign LIKE '50\% off!' ESCAPE '\\' THEN ...
```

BETWEEN Operator

The `BETWEEN` operator tests whether a value lies in a specified range. It means "greater than or equal to *low value* and less than or equal to *high value*." For example, the following expression is false:

```
45 BETWEEN 38 AND 44
```

IN Operator

The `IN` operator tests set membership. It means "equal to any member of." The set can contain nulls, but they are ignored. For example, the following expression tests whether a value is part of a set of values:

```
letter IN ('a','b','c')
```

Be careful when inverting this condition. Expressions of the form:

```
value NOT IN set
```

yield `FALSE` if the set contains a null.

Concatenation Operator

Double vertical bars (`||`) serve as the concatenation operator, which appends one string (`CHAR`, `VARCHAR2`, `CLOB`, or the equivalent Unicode-enabled type) to another. For example, the expression

```
'suit' || 'case'
```

returns the following value:

```
'suitcase'
```

If either operand is a `CLOB` value, the operator returns a temporary `CLOB`. Otherwise, it returns a `VARCHAR2` value.

BOOLEAN Expressions

PL/SQL lets you compare variables and constants in both SQL and procedural statements. These comparisons, called `BOOLEAN` expressions, consist of simple or complex expressions separated by relational operators. Often, `BOOLEAN` expressions are connected by the logical operators `AND`, `OR`, and `NOT`. A `BOOLEAN` expression always yields `TRUE`, `FALSE`, or `NULL`.

In a SQL statement, `BOOLEAN` expressions let you specify the rows in a table that are affected by the statement. In a procedural statement, `BOOLEAN` expressions are the basis for conditional control. There are three kinds of `BOOLEAN` expressions: arithmetic, character, and date.

Topics:

- [BOOLEAN Arithmetic Expressions](#)
- [BOOLEAN Character Expressions](#)
- [BOOLEAN Date Expressions](#)
- [Guidelines for BOOLEAN Expressions](#)

BOOLEAN Arithmetic Expressions

You can use the relational operators to compare numbers for equality or inequality. Comparisons are quantitative; that is, one number is greater than another if it represents a larger quantity. For example, given the assignments:

```
number1 := 75;
number2 := 70;
```

The following expression is true:

```
number1 > number2
```

BOOLEAN Character Expressions

You can compare character values for equality or inequality. By default, comparisons are based on the binary values of each byte in the string. For example, given the assignments:

```
string1 := 'Kathy';
string2 := 'Kathleen';
```

The following expression is true:

```
string1 > string2
```

By setting the initialization parameter `NLS_COMP=ANSI`, you can make 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.

Depending on the value of the `NLS_SORT` parameter, you can perform comparisons that are case-insensitive and even accent-insensitive. A case-insensitive comparison still returns true if the letters of the operands are different in terms of uppercase and lowercase. An accent-insensitive comparison is case-insensitive, and also returns true if the operands differ in accents or punctuation characters. For example, the character values `'True'` and `'TRUE'` are considered identical by a case-insensitive comparison; the character values `'Cooperate'`, `'Co-Operate'`, and `'coöperate'` are all considered the same. To make comparisons case-insensitive, add `_CI` to the end of your usual value for the `NLS_SORT` parameter. To make comparisons accent-insensitive, add `_AI` to the end of the `NLS_SORT` value.

There are semantic differences between the CHAR and VARCHAR2 base types that come into play when you compare character values. For more information, see ["Differences Between CHAR and VARCHAR2 Datatypes"](#) on page 3-9.

Many types can be converted to character types. For example, you can compare, assign, and do other character operations using CLOB variables. For details on the possible conversions, see ["PL/SQL Datatype Conversion"](#) on page 3-26.

BOOLEAN Date Expressions

You can also compare dates. Comparisons are chronological; that is, one date is greater than another if it is more recent. For example, given the assignments:

```
date1 := '01-JAN-91';
date2 := '31-DEC-90';
```

The following expression is true:

```
date1 > date2
```

Guidelines for BOOLEAN Expressions

In general, do not compare real numbers for exact equality or inequality. Real numbers are stored as approximate values. For example, the following IF condition might not yield TRUE:

```
DECLARE
    fraction BINARY_FLOAT := 1/3;
BEGIN
    IF fraction = 11/33 THEN
        DBMS_OUTPUT.PUT_LINE('Fractions are equal (luckily!');
    END IF;
END;
/
```

It is a good idea to use parentheses when doing comparisons. For example, the following expression is not allowed because $100 < \text{tax}$ yields a BOOLEAN value, which cannot be compared with the number 500:

```
100 < tax < 500 -- not allowed
```

The debugged version follows:

```
(100 < tax) AND (tax < 500)
```

A BOOLEAN variable is itself either true or false. You can just use the variable in a conditional test, rather than comparing it to the literal values TRUE and FALSE. In [Example 2-25](#) the loops are all equivalent.

Example 2-25 Using BOOLEAN Variables in Conditional Tests

```
DECLARE
    done BOOLEAN ;
BEGIN
    -- Each WHILE loop is 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;
/
```

Using CLOB values with comparison operators, or functions such as `LIKE` and `BETWEEN`, can create temporary LOBs. You might need to make sure your temporary tablespace is large enough to handle these temporary LOBs.

CASE Expressions

There are two types of expressions used in CASE statements: simple and searched. These expressions correspond to the type of CASE statement in which they are used. See ["Using the Simple CASE Statement"](#) on page 4-4.

Topics:

- [Simple CASE Expression](#)
- [Searched CASE Expression](#)

Simple CASE Expression

A simple CASE expression selects a result from one or more alternatives, and returns the result. Although it contains a block that might stretch over several lines, it really is an expression that forms part of a larger statement, such as an assignment or a subprogram call. The CASE expression uses a selector, an expression whose value determines which alternative to return.

A CASE expression has the form illustrated in [Example 2–26](#). The selector (`grade`) is followed by one or more `WHEN` clauses, which are checked sequentially. The value of the selector determines which clause is evaluated. The first `WHEN` clause that matches the value of the selector determines the result value, and subsequent `WHEN` clauses are not evaluated. If there are no matches, then the optional `ELSE` clause is performed.

Example 2–26 Using the WHEN Clause with a CASE Statement

```
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;
/
```

The optional `ELSE` clause works similarly to the `ELSE` clause in an `IF` statement. If the value of the selector is not one of the choices covered by a `WHEN` clause, the `ELSE` clause is executed. If no `ELSE` clause is provided and none of the `WHEN` clauses are matched, the expression returns `NULL`.

Searched CASE Expression

A searched `CASE` expression lets you test different conditions instead of comparing a single expression to various values. It has the form shown in [Example 2–27](#).

A searched `CASE` expression has no selector. Each `WHEN` clause contains a search condition that yields a `BOOLEAN` value, so you can test different variables or multiple conditions in a single `WHEN` clause.

Example 2–27 Using a Search Condition with a CASE Statement

```
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
                'N/A - Student not enrolled'
-- Test this condition early to detect good students
-- with bad attendance
            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;
/
```

The search conditions are evaluated sequentially. The `BOOLEAN` value of each search condition determines which `WHEN` clause is executed. If a search condition yields `TRUE`, its `WHEN` clause is executed. After any `WHEN` clause is executed, subsequent search conditions are not evaluated. If none of the search conditions yields `TRUE`, the optional `ELSE` clause is executed. If no `WHEN` clause is executed and no `ELSE` clause is supplied, the value of the expression is `NULL`.

Handling Null Values in Comparisons and Conditional Statements

When working with nulls, you can avoid some common mistakes by keeping in mind the following rules:

- Comparisons involving nulls always yield `NULL`.
- Applying the logical operator `NOT` to a null yields `NULL`.

- In conditional control statements, if the condition yields `NULL`, its associated sequence of statements is not executed.
- If the expression in a simple `CASE` statement or `CASE` expression yields `NULL`, it cannot be matched by using `WHEN NULL`. In this case, use the searched case syntax and test `WHEN expression IS NULL`.

In [Example 2–28](#), you might expect the sequence of statements to execute because `x` and `y` seem unequal. But, nulls are indeterminate. Whether or not `x` is equal to `y` is unknown. Therefore, the `IF` condition yields `NULL` and the sequence of statements is bypassed.

Topics:

- [NULL Values and the NOT Operator](#)
- [NULL Values and Zero-Length Strings](#)
- [NULL Values and the Concatenation Operator](#)
- [NULL Values as Arguments to Built-In Functions](#)

Example 2–28 Using NULLs in Comparisons

```
DECLARE
  x NUMBER := 5;
  y NUMBER := NULL;
BEGIN
  IF x != y THEN -- yields NULL, not TRUE
    DBMS_OUTPUT.PUT_LINE('x != y'); -- not executed
  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;

/

In the following example, you might expect the sequence of statements to execute because `a` and `b` seem equal. But, again, that is unknown, so the `IF` condition yields `NULL` and the sequence of statements is bypassed.

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

END IF;

END;

/

NULL Values and the NOT Operator

Recall that applying the logical operator `NOT` to a null yields `NULL`. Thus, the following two `IF` statements are not always equivalent:

```
IF x > y THEN high := x; ELSE high := y; END IF;
```



```
IF NOT x > y THEN high := y; ELSE high := x; END IF;
```

The sequence of statements in the `ELSE` clause is executed when the `IF` condition yields `FALSE` or `NULL`. If neither `x` nor `y` is null, both `IF` statements assign the same value to `high`. However, if either `x` or `y` is null, the first `IF` statement assigns the value of `y` to `high`, but the second `IF` statement assigns the value of `x` to `high`.

NULL Values and Zero-Length Strings

PL/SQL treats any zero-length string like a null. This includes values returned by character functions and `BOOLEAN` expressions. For example, the following statements assign nulls to the target variables:

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

Use the `IS NULL` operator to test for null strings, as follows:

```
IF v_string IS NULL THEN ...
```

NULL Values and the Concatenation Operator

The concatenation operator ignores null operands. For example, the expression:

```
'apple' || NULL || NULL || 'sauce'
```

Returns the following value:

```
'applesauce'
```

NULL Values as Arguments to Built-In Functions

If a null argument is passed to a built-in function, a null is returned except in the following cases.

The function `DECODE` compares its first argument to one or more search expressions, which are paired with result expressions. Any search or result expression can be null. If a search is successful, the corresponding result is returned. In [Example 2–29](#), if the column `manager_id` is null, `DECODE` returns the value `'nobody'`:

Example 2–29 Using the Function DECODE

```
DECLARE
    the_manager VARCHAR2(40);
    name employees.last_name%TYPE;
BEGIN
    -- NULL is a valid argument to DECODE.
    -- In this case, manager_id is null
    -- and the DECODE function returns 'nobody'.
    SELECT DECODE(manager_id, NULL, 'nobody', 'somebody'), last_name
    INTO the_manager, name FROM employees WHERE employee_id = 100;
    DBMS_OUTPUT.PUT_LINE(name || ' is managed by ' || the_manager);
END;
/
```

The function `NVL` returns the value of its second argument if its first argument is null. In [Example 2–30](#), if the column specified in the query is null, the function returns the value `-1` to signify a nonexistent employee in the output:

Example 2–30 Using the Function NVL

```

DECLARE
    the_manager employees.manager_id%TYPE;
    name employees.last_name%TYPE;
BEGIN
    -- NULL is a valid argument to NVL.
    -- In this case, manager_id is null
    -- and the NVL function returns -1.
    SELECT NVL(manager_id, -1), last_name
           INTO the_manager, name FROM employees WHERE employee_id = 100;
    DBMS_OUTPUT.PUT_LINE
        (name || ' is managed by employee Id: ' || the_manager);
END;
/

```

The function REPLACE returns the value of its first argument if its second argument is null, whether the optional third argument is present or not. For example, the call to REPLACE in [Example 2–31](#) does not make any change to the value of OLD_STRING:

Example 2–31 Using the Function REPLACE

```

DECLARE
    string_type VARCHAR2(60);
    old_string string_type%TYPE := 'Apples and oranges';
    v_string string_type%TYPE := 'more apples';
    -- NULL is a valid argument to REPLACE, but does not match
    -- anything so no replacement is done.
    new_string string_type%TYPE := REPLACE(old_string, NULL, v_string);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Old string = ' || old_string);
    DBMS_OUTPUT.PUT_LINE('New string = ' || new_string);
END;
/

```

If its third argument is null, REPLACE returns its first argument with every occurrence of its second argument removed. For example, the following call to REPLACE removes all the dashes from DASHED_STRING, instead of changing them to another character:

```

DECLARE
    string_type VARCHAR2(60);
    dashed string_type%TYPE := 'Gold-i-locks';
    -- When the substitution text for REPLACE is NULL,
    -- the text being replaced is deleted.
    name string_type%TYPE := REPLACE(dashed, '-', NULL);
BEGIN
    DBMS_OUTPUT.PUT_LINE('Dashed name = ' || dashed);
    DBMS_OUTPUT.PUT_LINE('Dashes removed = ' || name);
END;
/

```

If its second and third arguments are null, REPLACE just returns its first argument.

PL/SQL Error-Reporting Functions

PL/SQL has two built-in error-reporting functions, SQLCODE and SQLERRM, for use in PL/SQL exception-handling code. For their descriptions, see ["SQLCODE Function"](#) on page 13-139 and ["SQLERRM Function"](#) on page 13-140.

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

Using SQL Functions in PL/SQL

You can use all SQL functions except the following in PL/SQL expressions:

- Aggregate functions (such as `AVG` and `COUNT`)
- Analytic functions (such as `LAG` and `RATIO_TO_REPORT`)
- Collection functions (such as `CARDINALITY` and `SET`)
- Data mining functions (such as `CLUSTER_ID` and `FEATURE_VALUE`)
- Encoding and decoding functions (such as `DECODE` and `DUMP`)
- Model functions (such as `ITERATION_NUMBER` and `PREVIOUS`)
- Object reference functions (such as `REF` and `VALUE`)
- XML functions (such as `APPENDCHILDXML` and `EXISTSNODE`)
- The following conversion functions:
 - `BIN_TO_NUM`
 - `CAST`
 - `RAWTONHEX`
 - `ROWIDTONCHAR`
- The following miscellaneous functions:
 - `CUBE_TABLE`
 - `DATAOBJ_TO_PARTITION`
 - `LNNVL`
 - `SYS_CONNECT_BY_PATH`
 - `SYS_TYPEID`
 - `WIDTH_BUCKET`

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

When used in a PL/SQL expression, the `RAWTOHEX` function accepts an argument of datatype `RAW` and returns a `VARCHAR2` value with the hexadecimal representation of bytes that make up 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.

See Also: *Oracle Database SQL Language Reference* for information about SQL functions

Conditional Compilation

Using conditional compilation, you can customize the functionality in a PL/SQL application without having to remove any source code. For example, using conditional compilation you can customize a PL/SQL application to:

- Utilize the latest functionality with the latest database release and disable the new features to run the application against an older release of the database

- Activate debugging or tracing functionality in the development environment and hide that functionality in the application while it runs at a production site

See ["Understanding Conditional Compilation"](#) on page 1-13.

Topics:

- [How Does Conditional Compilation Work?](#)
- [Conditional Compilation Examples](#)
- [Conditional Compilation Restrictions](#)

How Does Conditional Compilation Work?

Conditional compilation uses selection directives, inquiry directives, and error directives to specify source text for compilation. Inquiry directives access values set up through name-value pairs in the `PLSQL_CCFLAGS` initialization parameter. Selection directives can test inquiry directives or static package constants.

The `DBMS_DB_VERSION` package provides database version and release constants that can be used for conditional compilation. The `DBMS_PREPROCESSOR` package provides subprograms for accessing the post-processed source text that is selected by conditional compilation directives in a PL/SQL unit.

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

Topics:

- [Conditional Compilation Control Tokens](#)
- [Using Conditional Compilation Selection Directives](#)
- [Using Conditional Compilation Error Directives](#)
- [Using Conditional Compilation Inquiry Directives](#)
- [Using Predefined Inquiry Directives with Conditional Compilation](#)
- [Using Static Expressions with Conditional Compilation](#)
- [Setting the PLSQL_CCFLAGS Initialization Parameter](#)
- [Using DBMS_DB_VERSION Package Constants](#)

Conditional Compilation Control Tokens

The conditional compilation trigger character is `$` and is used to identify code that is processed before the application is compiled. A conditional compilation control token is of the form:

```
preprocessor_control_token ::= $plsql_identifier
```

The `$` must be at the beginning of the identifier name and there cannot be a space between the `$` and the name. The `$` can also be embedded in the identifier name, but it has no special meaning. The reserved preprocessor control tokens are `$IF`, `$THEN`, `$ELSE`, `$ELSIF`, `$END`, and `$ERROR`. For an example of the use of the conditional compilation control tokens, see [Example 2-34](#) on page 2-40.

Using Conditional Compilation Selection Directives

The conditional compilation selection directive evaluates static expressions to determine which text to include in the compilation. The selection directive is of the form:

```
$IF boolean_static_expression $THEN text
  [$ELSIF boolean_static_expression $THEN text]
  [$ELSE text]
$END
```

`boolean_static_expression` must be a `BOOLEAN` static expression. For a description of `BOOLEAN` static expressions, see ["Using Static Expressions with Conditional Compilation"](#) on page 2-36. For information on PL/SQL `IF-THEN` control structures, see ["Testing Conditions \(IF and CASE Statements\)"](#) on page 4-2.

Using Conditional Compilation Error Directives

The error directive `$ERROR` raises a user-defined error and is of the form:

```
$ERROR varchar2_static_expression $END
```

`varchar2_static_expression` must be a `VARCHAR2` static expression. For a description of `VARCHAR2` static expressions, see ["Using Static Expressions with Conditional Compilation"](#) on page 2-36. See [Example 2-33](#).

Using Conditional Compilation Inquiry Directives

The inquiry directive is used to check the compilation environment. The inquiry directive is of the form:

```
inquiry_directive ::= $$id
```

An inquiry directive can be predefined as described in ["Using Predefined Inquiry Directives with Conditional Compilation"](#) on page 2-36 or be user-defined. The following describes the order of the processing flow when conditional compilation attempts to resolve an inquiry directive:

1. The `id` is used as an inquiry directive in the form `$$id` for the search key.
2. The two-pass algorithm proceeds as follows:
 - The string in the `PLSQL_CCFLAGS` initialization parameter is scanned from right to left, searching with `id` for a matching name (case insensitive); done if found.
 - The predefined inquiry directives are searched; done if found.
3. If the `$$id` cannot be resolved to a value, then the `PLW-6003` warning message is reported if the source text is not wrapped. The literal `NULL` is substituted as the value for undefined inquiry directives. Note that if the PL/SQL code is wrapped, then the warning message is disabled so that the undefined inquiry directive is not revealed.

For example, given the following session setting:

```
ALTER SESSION SET
  PLSQL_CCFLAGS = 'plsql_ccflags:true, debug:true, debug:0';
```

The value of `$$debug` is 0 and the value of `$$plsql_ccflags` is `TRUE`. Note that the value of `$$plsql_ccflags` resolves to the user-defined `plsql_ccflags` inside the value of the `PLSQL_CCFLAGS` compiler parameter. This occurs because a user-defined directive overrides the predefined one.

Given this session setting:

```
ALTER SESSION SET PLSQL_CCFLAGS = 'debug:true'
```

Now the value of `$$debug` is `TRUE`, the value of `$$psql_ccflags` is `'debug:true'`, the value of `$$my_id` is the literal `NULL`, and the use of `$$my_id` raises `PLW-6003` if the source text is not wrapped.

For an example of the use of an inquiry directive, see [Example 2-34](#) on page 2-40.

Using Predefined Inquiry Directives with Conditional Compilation

This section describes the inquiry directive names that are predefined and can be used in conditional expressions. These include:

- The Oracle initialization parameters for PL/SQL compilation, such as `PLSQL_CCFLAGS`, `PLSQL_OPTIMIZE_LEVEL`, `PLSQL_CODE_TYPE`, `PLSQL_WARNINGS`, and `NLS_LENGTH_SEMANTICS`. See "[Initialization Parameters for PL/SQL Compilation](#)" on page 12-1. For an example, see [Example 2-34](#).

Note that recompiling a PL/SQL unit with the `REUSE SETTINGS` clause of the `SQL ALTER` statement can protect against changes made to initialization parameter values in the current PL/SQL compilation environment. See [Example 2-35](#).

- `PLSQL_LINE` which is a `PLS_INTEGER` literal value indicating the line number reference to `$$PLSQL_LINE` in the current unit. For example:

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

Note that the value of `PLSQL_LINE` can be defined explicitly with `PLSQL_CCFLAGS`.

- `PLSQL_UNIT` which is a `VARCHAR2` literal value indicating the current source unit. For a named compilation unit, `$$PLSQL_UNIT` contains, but might not be limited to, the unit name. For an anonymous block, `$$PLSQL_UNIT` contains the empty string. For example:

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

Note that the value of `PLSQL_UNIT` can be defined explicitly with `PLSQL_CCFLAGS`. Also note that the previous example shows the use of `PLSQL_UNIT` in regular PL/SQL. Because `$$PLSQL_UNIT = 'AWARD_BONUS'` is a `VARCHAR2` comparison, not a static expression, it is not supported with `$IF`. One valid use of `$IF` with `PLSQL_UNIT` is to determine an anonymous block:

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

Using Static Expressions with Conditional Compilation

Only static expressions which can be fully evaluated by the compiler are allowed during conditional compilation processing. Any expression that contains references to variables or functions that require the execution of the PL/SQL are not available during compilation and cannot be evaluated. For information on PL/SQL datatypes, see "[Predefined PL/SQL Scalar Datatypes and Subtypes](#)" on page 3-1.

A static expression is either a `BOOLEAN`, `PLS_INTEGER`, or `VARCHAR2` static expression. Static constants declared in packages are also static expressions.

Topics:

- [Boolean Static Expressions](#)

- [PLS_INTEGER Static Expressions](#)
- [VARCHAR2 Static Expressions](#)
- [Static Constants](#)

Boolean Static Expressions BOOLEAN static expressions include:

- TRUE, FALSE, and the literal NULL
- Where x and y are PLS_INTEGER static expressions:
 - x > y
 - x < y
 - x >= y
 - x <= y
 - x = y
 - x <> y
- Where x and y are PLS_INTEGER BOOLEAN expressions:
 - NOT x
 - x AND y
 - x OR y
 - x > y
 - x >= y
 - x = y
 - x <= y
 - x <> y
- Where x is a static expression:
 - x IS NULL
 - x IS NOT NULL

PLS_INTEGER Static Expressions PLS_INTEGER static expressions include:

- -2147483648 to 2147483647, and the literal NULL

VARCHAR2 Static Expressions VARCHAR2 static expressions include:

- 'abcdef'
- 'abc' || 'def'
- Literal NULL
- TO_CHAR(x), where x is a PLS_INTEGER static expression
- TO_CHAR(x f,n) where x is a PLS_INTEGER static expression and f and n are VARCHAR2 static expressions
- x || y where x and y are VARCHAR2 or PLS_INTEGER static expressions

Static Constants Static constants are declared in a package specification as follows:

```
static_constant CONSTANT datatype := static_expression;
```

This is a valid declaration of a static constant if:

- The declared *datatype* and the type of *static_expression* are the same
- *static_expression* is a static expression
- *datatype* is either BOOLEAN or PLS_INTEGER

The static constant must be declared in the package specification and referred to as *package_name.constant_name*, even in the body of the *package_name* package.

If a static package constant is used as the BOOLEAN expression in a valid selection directive in a PL/SQL unit, then the conditional compilation mechanism automatically places a dependency on the package referred to. If the package is altered, then the dependent unit becomes invalid and needs to be recompiled to pick up any changes. Note that only valid static expressions can create dependencies.

If you choose to use a package with static constants for controlling conditional compilation in multiple PL/SQL units, then create only the package specification and dedicate it exclusively for controlling conditional compilation because of the multiple dependencies. Note that for control of conditional compilation in an individual unit, you can set a specific flag in PLSQL_CCFLAGS.

In [Example 2-32](#) the `my_debug` package defines constants for controlling debugging and tracing in multiple PL/SQL units. In the example, the constants `debug` and `trace` are used in static expressions in procedures `my_proc1` and `my_proc2`, which places a dependency from the procedures to `my_debug`.

Example 2-32 Using 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;
/
```

Changing the value of one of the constants forces all the dependent units of the package to recompile with the new value. For example, changing the value of `debug` to `FALSE` causes `my_proc1` to be recompiled without the debugging code. `my_proc2` is also recompiled, but `my_proc2` is unchanged because the value of `trace` did not change.

Setting the PLSQL_CCFLAGS Initialization Parameter

You can set the dynamic PLSQL_CCFLAGS initialization parameter to flag names with associated values to control conditional compilation on PL/SQL units. For example, the PLSQL_CCFLAGS initialization parameter can be set dynamically with ALTER

SESSION to turn on debugging and tracing functionality in PL/SQL units as shown in [Example 2-34](#).

You can also set the PLSQL_CCFLAGS initialization parameter to independently control conditional compilation on a specific PL/SQL unit with as shown in [Example 2-35](#) with the SQL ALTER PROCEDURE statement.

The flag names can be set to any unquoted PL/SQL identifier, including reserved words and keywords. If a flag value is explicitly set, it must be set to a TRUE, FALSE, PLS_INTEGER, or NULL. The flag names and values are not case sensitive. For detailed information, including restrictions, on the PLSQL_CCFLAGS initialization parameter, see *Oracle Database Reference*.

Using DBMS_DB_VERSION Package Constants

The DBMS_DB_VERSION package provides constants that are useful when making simple selections for conditional compilation. The PLS_INTEGER constants VERSION and RELEASE identify the current Oracle version and release numbers. The BOOLEAN constants VER_LE_9, VER_LE_9_1, VER_LE_9_2, VER_LE_10, VER_LE_10_1, and VER_LE_10_2 evaluate to TRUE or FALSE as follows:

- VER_LE_v evaluates to TRUE if the database version is less than or equal to v; otherwise, it evaluates to FALSE.
- VER_LE_v_r evaluates to TRUE if the database version is less than or equal to v and release is less than or equal to r; otherwise, it evaluates to FALSE.
- All constants representing Oracle 10g release 1 or earlier are FALSE

[Example 2-33](#) illustrates the use of a DBMS_DB_VERSION constant with conditional compilation. Both the Oracle database version and release are checked. This example also shows the use of \$ERROR.

Example 2-33 Using DBMS_DB_VERSION Constants

```
BEGIN
$IF DBMS_DB_VERSION.VER_LE_10_1 $THEN
    $ERROR 'unsupported database release' $END
$ELSE
    DBMS_OUTPUT.PUT_LINE
        ('Release ' || DBMS_DB_VERSION.VERSION || '.' ||
         DBMS_DB_VERSION.RELEASE || ' is supported. ');
    -- Note that this COMMIT syntax is newly supported in 10.2
    COMMIT WRITE IMMEDIATE NOWAIT;
$END
END;
/
```

For information on the DBMS_DB_VERSION package, see *Oracle Database PL/SQL Packages and Types Reference*.

Conditional Compilation Examples

This section provides examples using conditional compilation.

Topics:

- [Using Conditional Compilation to Specify Code for Database Versions](#)
- [Using DBMS_PREPROCESSOR Procedures to Print or Retrieve Source Text](#)

Using Conditional Compilation to Specify Code for Database Versions

In [Example 2–34](#), conditional compilation is used to determine whether the `BINARY_DOUBLE` datatype can be utilized in the calculations for PL/SQL units in the database. The `BINARY_DOUBLE` datatype can only be used in a database version that is 10g or later. This example also shows the use of the `PLSQL_CCFLAGS` parameter.

Example 2–34 Using Conditional Compilation with Database Versions

```
-- set flags for displaying debugging code and tracing info
ALTER SESSION SET PLSQL_CCFLAGS =
  'my_debug:FALSE, my_tracing:FALSE';

CREATE PACKAGE my_pkg AS
  SUBTYPE my_real IS
    $IF DBMS_DB_VERSION.VERSION < 10 $THEN NUMBER;
    -- check database version
    $ELSE BINARY_DOUBLE;
  $END
  my_pi my_real; my_e my_real;
END my_pkg;
/

CREATE PACKAGE BODY my_pkg AS
BEGIN -- set up values for future calculations based on DB version
  $IF DBMS_DB_VERSION.VERSION < 10 $THEN
    my_pi := 3.14016408289008292431940027343666863227;
    my_e := 2.71828182845904523536028747135266249775;
  $ELSE
    my_pi := 3.14016408289008292431940027343666863227d;
    my_e := 2.71828182845904523536028747135266249775d;
  $END
END my_pkg;
/

CREATE PROCEDURE circle_area(radius my_pkg.my_real) IS
  my_area my_pkg.my_real;
  my_datatype VARCHAR2(30);
BEGIN
  my_area := my_pkg.my_pi * radius;
  DBMS_OUTPUT.PUT_LINE('Radius: ' || TO_CHAR(radius)
    || ' Area: ' || TO_CHAR(my_area) );
  $IF $$my_debug $THEN
    -- if my_debug is TRUE, run some debugging code
    SELECT DATA_TYPE INTO my_datatype FROM USER_ARGUMENTS
      WHERE OBJECT_NAME =
        'CIRCLE_AREA' AND ARGUMENT_NAME = 'RADIUS';
    DBMS_OUTPUT.PUT_LINE('Datatype of the RADIUS argument is: ' || my_datatype);
  $END
END;
/
```

If you want to set `my_debug` to `TRUE`, you can make this change only for procedure `circle_area` with the `REUSE SETTINGS` clause as shown in [Example 2–35](#).

Example 2–35 Using ALTER PROCEDURE to Set PLSQL_CCFLAGS

```
ALTER PROCEDURE circle_area COMPILE PLSQL_CCFLAGS = 'my_debug:TRUE'
  REUSE SETTINGS;
```

Using DBMS_PREPROCESSOR Procedures to Print or Retrieve Source Text

DBMS_PREPROCESSOR subprograms print or retrieve the post-processed source text of a PL/SQL unit after processing the conditional compilation directives. This post-processed text is the actual source used to compile a valid PL/SQL unit.

[Example 2-36](#) shows how to print the post-processed form of `my_pkg` in [Example 2-34](#) with the `PRINT_POST_PROCESSED_SOURCE` procedure.

Example 2-36 Using PRINT_POST_PROCESSED_SOURCE to Display Source Code

```
CALL DBMS_PREPROCESSOR.PRINT_POST_PROCESSED_SOURCE
 ('PACKAGE', 'HR', 'MY_PKG');
```

When `my_pkg` in [Example 2-34](#) is compiled on a 10g release or later database using the HR account, the output of [Example 2-36](#) is similar to the following:

```
PACKAGE my_pkg AS
  SUBTYPE my_real IS

                                     BINARY_DOUBLE
;

  my_pi my_real; my_e my_real;
END my_pkg;
```

`PRINT_POST_PROCESSED_SOURCE` replaces unselected text with whitespace. The lines of code in [Example 2-34](#) that are not included in the post-processed text are represented as blank lines. For information on the `DBMS_PREPROCESSOR` package, see *Oracle Database PL/SQL Packages and Types Reference*.

Conditional Compilation Restrictions

A conditional compilation directive cannot be used in the specification of an object type or in the specification of a schema-level nested table or varray. The attribute structure of dependent types and the column structure of dependent tables is determined by the attribute structure specified in object type specifications. Any changes to the attribute structure of an object type must be done in a controlled manner to propagate the changes to dependent objects. The mechanism for propagating changes is the SQL `ALTER TYPE ATTRIBUTE` statement. Use of a preprocessor directive allows changes to the attribute structure of the object type without the use of an `ALTER TYPE ATTRIBUTE` statement. As a consequence, dependent objects can "go out of sync" or dependent tables can become inaccessible.

The SQL parser imposes restrictions on the placement of directives when performing SQL operations such as the `CREATE OR REPLACE` statement or the execution of an anonymous block. When performing these SQL operations, the SQL parser imposes a restriction on the location of the first conditional compilation directive as follows:

- A conditional compilation directive cannot be used in the specification of an object type or in the specification of a schema-level nested table or varray.
- 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 may occur immediately after the keyword `IS` or `AS`.
- In a schema-level subprogram with at least one formal parameter, the first conditional compilation directive may occur immediately after the opening parenthesis that follows the unit's name. For example:

```
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 may occur immediately after the keyword `BEGIN` or immediately after the keyword `DECLARE` when the trigger block has a `DECLARE` section.
- If an anonymous block uses a placeholder, then this cannot occur within a conditional compilation directive. For example:

```
BEGIN  
  :n := 1; -- valid use of placeholder  
  $IF ... $THEN  
    :n := 1; -- invalid use of placeholder  
$END
```

Using PL/SQL to Create Web Applications and Server Pages

You can use PL/SQL to develop Web applications or server pages. These are briefly described in this section. For detailed information on using PL/SQL to create Web applications, see "Developing Applications with the PL/SQL Web Toolkit" in *Oracle Database Advanced Application Developer's Guide*. For detailed information on using PL/SQL to create Web Server Pages (PSPs), see "Developing PL/SQL Server Pages" in *Oracle Database Advanced Application Developer's Guide*.

Topics:

- [PL/SQL Web Applications](#)
- [PL/SQL Server Pages](#)

PL/SQL Web Applications

With PL/SQL you can create applications that generate Web pages directly from an Oracle 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 Oracle Database. Delivering Web content with PL/SQL stored subprograms provides the power and flexibility of database processing. For example, you can use DML, dynamic SQL, and cursors. You also eliminate the process overhead of forking a new CGI process to handle each HTTP request.

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

- PL/SQL gateway enables a Web browser to invoke a PL/SQL stored subprogram through an HTTP listener. `mod_plsql`, one implementation of the PL/SQL gateway, is a plug-in of Oracle HTTP Server and enables Web browsers to invoke PL/SQL stored subprograms.
- PL/SQL Web Toolkit is a set of PL/SQL packages that provides a generic interface to use stored subprograms invoked by `mod_plsql` at run time.

PL/SQL Server Pages

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

Using special tags, you can embed PL/SQL scripts into HTML source code. The scripts are executed when the pages are requested by Web clients such as browsers. A script can accept parameters, query or update the database, 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.

PL/SQL Datatypes

Every constant, variable, and parameter has a **datatype** (also called a **type**) that determines its storage format, constraints, valid range of values, and operations that can be performed on it. PL/SQL provides many predefined datatypes and subtypes, and lets you define your own PL/SQL subtypes.

A **subtype** is a subset of another datatype, which is called its **base type**. A subtype has the same valid operations as its base type, but only a subset of its valid values. Subtypes can increase reliability, provide compatibility with ANSI/ISO types, and improve readability by indicating the intended use of constants and variables.

This chapter explains the basic, frequently used predefined PL/SQL datatypes and subtypes, how to define and use your own PL/SQL subtypes, and PL/SQL datatype conversion. Later chapters explain specialized predefined datatypes.

[Table 3–1](#) lists the categories of predefined PL/SQL datatypes, describes the data they store, and tells where to find information about the specialized datatypes.

Table 3–1 Categories of Predefined PL/SQL Datatypes

Datatype Category	Data Description
Scalar	Single values with no internal components.
Composite	Data items that have internal components that can be accessed individually. Explained in Chapter 5, "Using PL/SQL Collections and Records" .
Reference	Pointers to other data items. Explained in "Using Cursor Variables (REF CURSORS)" on page 6-22.
Large Object (LOB)	Pointers to large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms.

Topics:

- [Predefined PL/SQL Scalar Datatypes and Subtypes](#)
- [Predefined PL/SQL Large Object \(LOB\) Datatypes](#)
- [User-Defined PL/SQL Subtypes](#)
- [PL/SQL Datatype Conversion](#)

Predefined PL/SQL Scalar Datatypes and Subtypes

Scalar datatypes store single values with no internal components. [Table 3–2](#) lists the predefined PL/SQL scalar datatypes and describes the data they store.

Table 3–2 Categories of Predefined PL/SQL Scalar Datatypes

Category	Data Description
Numeric	Numeric values, on which you can perform arithmetic operations.
Character	Alphanumeric values that represent single characters or strings of characters, which you can manipulate.
BOOLEAN	Logical values, on which you can perform logical operations.
Datetime	Dates and times, which you can manipulate.
Interval	Time intervals, which you can manipulate.

Topics:

- [Predefined PL/SQL Numeric Datatypes and Subtypes](#)
- [Predefined PL/SQL Character Datatypes and Subtypes](#)
- [Predefined PL/SQL BOOLEAN Datatype](#)
- [Predefined PL/SQL Datetime and Interval Datatypes](#)

Predefined PL/SQL Numeric Datatypes and Subtypes

Numeric datatypes let you store numeric data, represent quantities, and perform calculations. [Table 3–3](#) lists the predefined PL/SQL numeric types and describes the data they store.

Table 3–3 Predefined PL/SQL Numeric Datatypes

Datatype	Data Description
PLS_INTEGER or BINARY_INTEGER	Signed integer in range -2,147,483,648 through 2,147,483,647, represented in 32 bits
BINARY_FLOAT	Single-precision IEEE 754-format floating-point number
BINARY_DOUBLE	Double-precision IEEE 754-format floating-point number
NUMBER	Fixed-point or floating-point number with absolute value in range 1E-130 to (but not including) 1.0E126. A NUMBER variable can also represent 0.

Topics:

- [PLS_INTEGER and BINARY_INTEGER Datatypes](#)
- [SIMPLE_INTEGER Subtype of PLS_INTEGER](#)
- [BINARY_FLOAT and BINARY_DOUBLE Datatypes](#)
- [NUMBER Datatype](#)

PLS_INTEGER and BINARY_INTEGER Datatypes

The PLS_INTEGER and BINARY_INTEGER datatypes are identical. For simplicity, this document uses "PLS_INTEGER" to mean both PLS_INTEGER and BINARY_INTEGER.

The PLS_INTEGER datatype stores signed integers in the range -2,147,483,648 through 2,147,483,647, represented in 32 bits.

The PLS_INTEGER datatype has the following advantages over the NUMBER datatype and NUMBER subtypes:

- PLS_INTEGER values require less storage.

- PLS_INTEGER operations use hardware arithmetic, so they are faster than NUMBER operations, which use library arithmetic.

For efficiency, use PLS_INTEGER values for all calculations that fall within its range. For calculations outside the PLS_INTEGER range, use INTEGER, a predefined subtype of the NUMBER datatype.

Note: When a calculation with two PLS_INTEGER datatypes overflows the PLS_INTEGER range, an overflow exception is raised even if the result is assigned to a NUMBER datatype.

Table 3–4 lists the predefined subtypes of the PLS_INTEGER datatype and describes the data they store.

Table 3–4 Predefined Subtypes of PLS_INTEGER Datatype

Datatype	Data Description
NATURAL	Nonnegative PLS_INTEGER value
NATURALN	Nonnegative PLS_INTEGER value with NOT NULL constraint
POSITIVE	Positive PLS_INTEGER value
POSITIVEN	Positive PLS_INTEGER value with NOT NULL constraint
SIGNTYPE	PLS_INTEGER value -1, 0, or 1 (useful for programming tri-state logic)
SIMPLE_INTEGER	PLS_INTEGER value with NOT NULL constraint

SIMPLE_INTEGER Subtype of PLS_INTEGER

SIMPLE_INTEGER is a predefined subtype of the PLS_INTEGER datatype that has the same range as PLS_INTEGER (-2,147,483,648 through 2,147,483,647) and has a NOT NULL constraint. It differs significantly from PLS_INTEGER in its overflow semantics.

You can use SIMPLE_INTEGER when the value will never be NULL and overflow checking is unnecessary. Without the overhead of checking for nullness and overflow, SIMPLE_INTEGER provides significantly better performance than PLS_INTEGER when PLSQL_CODE_TYPE= 'NATIVE', because arithmetic operations on SIMPLE_INTEGER values are done directly in the hardware. When PLSQL_CODE_TYPE= 'INTERPRETED', the performance improvement is smaller.

Topics:

- [Overflow Semantics](#)
- [Overloading Rules](#)
- [Integer Literals](#)
- [Cast Operations](#)
- [Compiler Warnings](#)

Overflow Semantics The overflow semantics of SIMPLE_INTEGER differ significantly from those of PLS_INTEGER. An arithmetic operation that increases a PLS_INTEGER value to greater than 2,147,483,647 or decrease it to less than -2,147,483,648 causes error ORA-01426 ("numeric overflow"). In contrast, when the following PL/SQL block is run from SQL*Plus, it runs without error:

```
DECLARE n SIMPLE_INTEGER := 2147483645;
BEGIN
```

```
FOR j IN 1..4 LOOP
  n := n + 1;
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S999999999'));
END LOOP;
FOR j IN 1..4 LOOP
  n := n - 1;
  DBMS_OUTPUT.PUT_LINE(TO_CHAR(n, 'S999999999'));
END LOOP;
END;
```

The output of the preceding PL/SQL block is:

```
+2147483646
+2147483647
-2147483648
-2147483647
-2147483648
+2147483647
+2147483646
+2147483645
```

Overloading Rules

- In overloaded subprograms, `SIMPLE_INTEGER` and `PLS_INTEGER` actual parameters can be substituted for each other.
- If all of their operands or arguments have the datatype `SIMPLE_INTEGER`, the following produce `SIMPLE_INTEGER` results, using two's complement arithmetic and ignoring overflows:
 - Operators:
 - * Addition (+)
 - * Subtraction (-)
 - * Multiplication (*)
 - Built-in functions:
 - * MAX
 - * MIN
 - * ROUND
 - * SIGN
 - * TRUNC
 - CASE expression

If some but not all operands or arguments have the datatype `SIMPLE_INTEGER`, those of the datatype `SIMPLE_INTEGER` are implicitly cast to `PLS_INTEGER NOT NULL`.

Integer Literals Integer literals in the `SIMPLE_INTEGER` range have the datatype `SIMPLE_INTEGER`. This relieves you from explicitly casting each integer literal to `SIMPLE_INTEGER` in arithmetic expressions computed using two's complement arithmetic.

If and only if all operands and arguments have the datatype `SIMPLE_INTEGER`, PL/SQL uses two's complement arithmetic and ignores overflows. Because overflows are ignored, values can wrap from positive to negative or from negative to positive; for example:

$$2^{30} + 2^{30} = 0x40000000 + 0x40000000 = 0x80000000 = -2^{31}$$

$$-2^{31} + -2^{31} = 0x80000000 + 0x80000000 = 0x00000000 = 0$$

To ensure backward compatibility, when all operands in an arithmetic expression are integer literals, PL/SQL treats the integer literals as if they were cast to `PLS_INTEGER`.

Cast Operations A cast operation that coerces a `PLS_INTEGER` value to the `SIMPLE_INTEGER` datatype makes no conversion if the source value is not `NULL`. If the source value is `NULL`, a run-time exception is raised.

A cast operation that coerces a `SIMPLE_INTEGER` value to the `PLS_INTEGER` datatype makes no conversion. This operation always succeeds (no error is raised).

Compiler Warnings The compiler issues a warning in the following cases:

- An operation mixes `SIMPLE_INTEGER` values with values of other numeric types.
- A `SIMPLE_INTEGER` value is passed as a parameter, a bind, or a define where a `PLS_INTEGER` is expected.

BINARY_FLOAT and BINARY_DOUBLE Datatypes

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

A `BINARY_FLOAT` literal ends with `f` (for example, `2.07f`). A `BINARY_DOUBLE` literal ends with `d` (for example, `3.000094d`).

`BINARY_FLOAT` and `BINARY_DOUBLE` computations do not raise exceptions; therefore, you must check the values that they produce for conditions such as overflow and underflow, using the predefined constants listed and described in [Table 3–5](#). For example:

```
SELECT COUNT(*)
FROM employees
WHERE salary < BINARY_FLOAT_INFINITY;
```

Table 3–5 Predefined PL/SQL `BINARY_FLOAT` and `BINARY_DOUBLE` Constants¹

Constant	Description
<code>BINARY_FLOAT_NAN</code> ¹	<code>BINARY_FLOAT</code> value for which the condition <code>IS NAN</code> (not a number) is true
<code>BINARY_FLOAT_INFINITY</code> ¹	Single-precision positive infinity
<code>BINARY_FLOAT_MAX_NORMAL</code>	<code>BINARY_FLOAT</code> value for which the condition <code>???</code> is true
<code>BINARY_FLOAT_MIN_NORMAL</code>	<code>BINARY_FLOAT</code> value for which the condition <code>???</code> is true
<code>BINARY_FLOAT_MAX_SUBNORMAL</code>	<code>BINARY_FLOAT</code> value for which the condition <code>???</code> is true
<code>BINARY_FLOAT_MIN_SUBNORMAL</code>	<code>BINARY_FLOAT</code> value for which the condition <code>???</code> is true
<code>BINARY_DOUBLE_NAN</code> ¹	<code>BINARY_DOUBLE</code> value for which the condition <code>IS NAN</code> (not a number) is true
<code>BINARY_DOUBLE_INFINITY</code> ¹	Double-precision positive infinity
<code>BINARY_DOUBLE_MAX_NORMAL</code>	<code>BINARY_DOUBLE</code> value for which the condition <code>???</code> is true

Table 3–5 (Cont.) Predefined PL/SQL BINARY_FLOAT and BINARY_DOUBLE Constants¹

Constant	Description
BINARY_DOUBLE_MIN_NORMAL	BINARY_DOUBLE value for which the condition ??? is true
BINARY_DOUBLE_MAX_SUBNORMAL	BINARY_DOUBLE value for which the condition ??? is true
BINARY_DOUBLE_MIN_SUBNORMAL	BINARY_DOUBLE value for which the condition ??? is true

¹ Also predefined by SQL

BINARY_FLOAT and BINARY_DOUBLE datatypes are primarily for high-speed scientific computation, as explained in ["Writing Computation-Intensive PL/SQL Programs"](#) on page 12-24. See also ["Guidelines for Overloading with Numeric Types"](#) on page 8-13, for information about writing libraries that accept different numeric types.

NUMBER Datatype

The NUMBER datatype stores fixed-point or floating-point numbers with absolute values in the range 1E-130 up to (but not including) 1.0E126. A NUMBER variable can also represent 0.

Oracle recommends using only NUMBER literals and results of NUMBER computations that are within the specified range. Otherwise, the following happen:

- Any value that is too small is rounded to zero.
- A literal value that is too large causes a compilation error.
- A computation result that is too large is undefined, causing unreliable results and possibly run-time errors.

A NUMBER value has both **precision** (its total number of digits) and **scale** (the number of digits to the right of the decimal point).

The syntax for specifying a fixed-point NUMBER is:

```
NUMBER(precision, scale)
```

For example:

```
NUMBER(8,2)
```

For an integer, the scale is zero. The syntax for specifying an integer NUMBER is:

```
NUMBER(precision)
```

For example:

```
NUMBER(2)
```

In a floating-point number, the decimal point can float to any position. The syntax for specifying a floating-point NUMBER is:

```
NUMBER
```

Both *precision* and *scale* must be integer literals, not constants or variables.

For *precision*, the maximum value is 38. The default value is 39 or 40, or the maximum for your system, whichever is least.

For *scale*, the minimum and maximum values are -84 and 127, respectively. The default value is zero.

Scale determines where rounding occurs. For example, a value whose scale is 2 is rounded to the nearest hundredth (3.454 becomes 3.45 and 3.456 becomes 3.46). A negative scale causes rounding to the left of the decimal point. For example, a value whose scale is -3 is rounded to the nearest thousand (34462 becomes 34000 and 34562 becomes 35000). A value whose scale is 0 is rounded to the nearest integer (3.4562 becomes 3 and 3.56 becomes 4).

For more information about the NUMBER datatype, see *Oracle Database SQL Language Reference*.

[Table 3–6](#) lists the predefined subtypes of the NUMBER datatype and describes the data they store.

Table 3–6 Predefined Subtypes of NUMBER Datatype

Datatype	Description
DEC, DECIMAL, or NUMERIC	Fixed-point NUMBER with maximum precision of 38 decimal digits
DOUBLE PRECISION or FLOAT	Floating-point NUMBER with maximum precision of 126 binary digits (approximately 38 decimal digits)
INT, INTEGER, or SMALLINT	Integer with maximum precision of 38 decimal digits
REAL	Floating-point NUMBER with maximum precision of 63 binary digits (approximately 18 decimal digits)

Predefined PL/SQL Character Datatypes and Subtypes

Character datatypes let you store alphanumeric values that represent single characters or strings of characters, which you can manipulate. [Table 3–7](#) describes the predefined PL/SQL character types and describes the data they store.

Table 3–7 Predefined PL/SQL Character Datatypes¹

Datatype	Data Description
CHAR	Fixed-length character string with maximum size of 32,767 bytes
VARCHAR2	Variable-length character string with maximum size of 32,767 bytes
RAW	Variable-length binary or byte string with maximum size of 32,767 bytes, not interpreted by PL/SQL
NCHAR	Fixed-length national character string with maximum size of 32,767 bytes
NVARCHAR2	Variable-length national character string with maximum size of 32,767 bytes
LONG ¹	Variable-length character string with maximum size of 32,760 bytes
LONG RAW ¹	Variable-length binary or byte string with maximum size of 32,760 bytes, not interpreted by PL/SQL
ROWID ¹	Physical row identifier, the address of a row in an ordinary table
UROWID	Universal row identifier (physical, logical, or foreign row identifier)

¹ Supported only for backward compatibility with existing applications

Topics:

- [CHAR and VARCHAR2 Datatypes](#)

- [RAW Datatype](#)
- [NCHAR and NVARCHAR2 Datatypes](#)
- [LONG and LONG RAW Datatypes](#)
- [ROWID and UROWID Datatypes](#)

CHAR and VARCHAR2 Datatypes

The CHAR and VARCHAR2 datatypes store fixed-length and variable-length character strings, respectively. All string literals have datatype CHAR.

How CHAR and VARCHAR2 data is represented internally depends on the database character set specified with the CHARACTER SET clause of the CREATE DATABASE statement, which is described in *Oracle Database SQL Language Reference*.

The syntax for specifying a CHAR or VARCHAR2 data item is:

```
[ CHAR | VARCHAR2 ] [( maximum_size [ CHAR | BYTE ] )]
```

For example:

```
CHAR  
VARCHAR2  
CHAR(10 CHAR)  
VARCHAR2(32 BYTE)
```

The *maximum_size* must be an integer literal in the range 1..32767, not a constant or variable. The default value is one.

The default size unit (CHAR or BYTE) is determined by the NLS_LENGTH_SEMANTICS initialization parameter. When a PL/SQL subprogram is compiled, the setting of this parameter is recorded, so that the same setting is used when the subprogram is recompiled after being invalidated. For more information about NLS_LENGTH_SEMANTICS, see *Oracle Database Reference*.

The maximum size of a CHAR or VARCHAR2 data item is 32,767 bytes, whether you specify *maximum_size* in characters or bytes. The maximum number of characters in a CHAR or VARCHAR2 data item depends on how the character set is encoded. For a single-byte character set, the maximum size of a CHAR or VARCHAR2 data item is 32,767 characters. For an *n*-byte character set, the maximum size of a CHAR or VARCHAR2 data item is 32,767/*n* characters, rounded down to the nearest integer. For a multiple-byte character set, specify *maximum_size* in characters to ensure that a CHAR(*n*) or VARCHAR2(*n*) variable can store *n* multiple-byte characters.

If the character value that you assign to a character variable is longer than the maximum size of the variable, PL/SQL does not truncate the value or strip trailing blanks; it stops the assignment and raises the predefined exception VALUE_ERROR.

For example, given the declaration:

```
acronym CHAR(4);
```

the following assignment raises VALUE_ERROR:

```
acronym := 'SPCA '; -- note trailing blank
```

If the character value that you insert into a database column (using either INSERT or UPDATE) is longer than the defined width of the column, PL/SQL does not truncate the value or strip trailing blanks; it stops the insertion and raises error.

To strip trailing blanks from a character value before assigning it to a variable or inserting it into a database column, use the built-in function `RTRIM`. For example, given the preceding declaration, the following assignment does not raise an error:

```
acronym := RTRIM('SPCA '); -- note trailing blank
```

For the syntax of `RTRIM`, see *Oracle Database SQL Language Reference*.

Differences Between CHAR and VARCHAR2 Datatypes

`CHAR` and `VARCHAR2` datatypes differ in the following:

- [Predefined Subtypes of Character Datatypes](#)
- [Memory Allocation for Character Variables](#)
- [Blank-Padding Shorter Character Values](#)
- [Comparing Character Values](#)
- [Maximum Sizes of Values Inserted into Character Database Columns](#)

Predefined Subtypes of Character Datatypes The `CHAR` datatype has one predefined subtype, `CHARACTER`. The `VARCHAR2` datatype has two predefined subtypes, `VARCHAR` and `STRING`. Each of these subtypes has the same range of values as its base type, and can be used instead of its base type for compatibility with ANSI/ISO and IBM types.

Note: In a future PL/SQL release, to accommodate emerging SQL standards, `VARCHAR` might become a separate datatype, no longer synonymous with `VARCHAR2`.

Memory Allocation for Character Variables For a `CHAR` variable, or for a `VARCHAR2` variable whose maximum size is less than 2,000 bytes, PL/SQL allocates enough memory for the maximum size at compile time. For a `VARCHAR2` whose maximum size is 2,000 bytes or more, PL/SQL allocates enough memory to store the actual value at run time. In this way, PL/SQL optimizes smaller `VARCHAR2` variables for performance and larger ones for efficient memory use.

For example, if you assign the same 500-byte value to `VARCHAR2(1999 BYTE)` and `VARCHAR2(2000 BYTE)` variables, PL/SQL allocates 1999 bytes for the former variable at compile time and 500 bytes for the latter variable at run time.

Blank-Padding Shorter Character Values In each of the following situations, whether or not PL/SQL blank-pads the character value depends on the datatype of the receiver:

- The character value that you assign to a PL/SQL character variable is shorter than the maximum size of the variable.
- The character value that you insert into a character database column (using `INSERT`, `UPDATE`, or `MERGE`) is shorter than the defined width of the column.
- The value that you retrieve from a character database column (using `SELECT` or `FETCH`) into a PL/SQL character variable is shorter than the maximum length of the variable.

If the datatype of the receiver is `CHAR`, PL/SQL blank-pads the value to the maximum size. Information about trailing blanks in the original value is lost.

For example, the value assigned to `last_name` in the following statement has six trailing blanks, not only one:

```
last_name CHAR(10) := 'CHEN '; -- note trailing blank
```

If the datatype of the receiver is VARCHAR2, PL/SQL neither blank-pads the value nor strips trailing blanks. Character values are assigned intact, and no information is lost.

Comparing Character Values You can use the ["Relational Operators"](#) on page 2-24 to compare character values. One character value is greater than another if it follows it in the collating sequence used for the database character set. In the following example, the IF condition is TRUE:

```
DECLARE
  last_name1 VARCHAR2(10) := 'COLES';
  last_name2 VARCHAR2(10) := 'COLEMAN';
BEGIN
  IF last_name1 > last_name2 THEN
    DBMS_OUTPUT.PUT_LINE
      (last_name1 || ' is greater than ' || last_name2);
  ELSE
    DBMS_OUTPUT.PUT_LINE
      (last_name2 || ' is greater than ' || last_name1 );
  END IF;
END;
/
```

To be equal, two character values must have the same length.

If both values have datatype CHAR, PL/SQL blank-pads the shorter value to the length of the longer value before comparing them. In [Example 3-1](#), the IF condition is TRUE.

If either value has datatype VARCHAR2, PL/SQL does not adjust their lengths before comparing them. In both [Example 3-2](#) and [Example 3-3](#), the IF condition is FALSE.

Example 3-1 Comparing Two CHAR Values

```
DECLARE
  last_name1 CHAR(5) := 'BELLO';
  last_name2 CHAR(10) := 'BELLO  '; -- note trailing blanks
BEGIN
  IF last_name1 = last_name2 THEN
    DBMS_OUTPUT.PUT_LINE
      (last_name1 || ' is equal to ' || last_name2);
  ELSE
    DBMS_OUTPUT.PUT_LINE
      (last_name2 || ' is not equal to ' || last_name1);
  END IF;
END;
/
```

Example 3-2 Comparing Two VARCHAR2 Values

```
DECLARE
  last_name1 VARCHAR2(10) := 'DOW';
  last_name2 VARCHAR2(10) := 'DOW  '; -- note trailing blanks
BEGIN
  IF last_name1 = last_name2 THEN
    DBMS_OUTPUT.PUT_LINE
      (last_name1 || ' is equal to ' || last_name2 );
  ELSE
    DBMS_OUTPUT.PUT_LINE;
      (last_name2 || ' is not equal to ' || last_name1);
  END IF;
END;
```



```
END;
/
```

Example 3-3 Comparing CHAR Value and VARCHAR2 Value

```
DECLARE
  last_name1 VARCHAR2(10) := 'STAUB';
  last_name2 CHAR(10)     := 'STAUB'; -- PL/SQL blank-pads value
BEGIN
  IF last_name1 = last_name2 THEN
    DBMS_OUTPUT.PUT_LINE
      (last_name1 || ' is equal to ' || last_name2);
  ELSE
    DBMS_OUTPUT.PUT_LINE
      (last_name2 || ' is not equal to ' || last_name1 );
  END IF;
END;
/
```

Maximum Sizes of Values Inserted into Character Database Columns The largest CHAR value that you can insert into a CHAR database column (using INSERT, UPDATE, or MERGE) is 2,000 bytes.

The largest VARCHAR2 value that you can insert into a VARCHAR2 database column is 4,000 bytes.

You can insert any CHAR or VARCHAR2 value into a LONG database column, because the maximum width of a LONG column is 2,147,483,648 bytes (2 GB). However, you cannot retrieve a value longer than 32,767 bytes from a LONG column into a CHAR or VARCHAR2 variable. (The LONG datatype is supported only for backward compatibility with existing applications. For more information, see ["LONG and LONG RAW Datatypes"](#) on page 3-13.)

RAW Datatype

The RAW datatype stores binary or byte strings, such as sequences of graphics characters or digitized pictures. Raw data is like VARCHAR2 data, except that PL/SQL does not interpret raw data. Oracle Net does no character set conversions when you transmit raw data from one system to another.

The syntax for specifying a RAW data item is:

```
RAW (maximum_size)
```

For example:

```
RAW(256)
```

The *maximum_size*, in bytes, must be an integer literal in the range 1..32767, not a constant or variable. The default value is one.

The largest RAW value that you can insert into a RAW database column is 2,000 bytes.

You can insert any RAW value into a LONG RAW database column, because the maximum width of a LONG RAW column is 2,147,483,648 bytes (2 GB). However, you cannot retrieve a value longer than 32,767 bytes from a LONG RAW column into a RAW variable. (The LONG RAW datatype is supported only for backward compatibility with existing applications. For more information, see ["LONG and LONG RAW Datatypes"](#) on page 3-13.)

NCHAR and NVARCHAR2 Datatypes

The NCHAR and NVARCHAR2 datatypes store fixed-length and variable-length national character strings, respectively.

National character strings are composed of characters from the national character set, which is used to represent languages that have thousands of characters, each of which requires two or three bytes (Japanese, for example).

How NCHAR and NVARCHAR2 data is represented internally depends on the national character set specified with the NATIONAL CHARACTER SET clause of the CREATE DATABASE statement, which is described in *Oracle Database SQL Language Reference*.

Topics:

- [AL16UTF16 and UTF8 Encodings](#)
- [NCHAR Datatype](#)
- [NVARCHAR2 Datatype](#)

AL16UTF16 and UTF8 Encodings The national character set represents data as Unicode, using either the AL16UTF16 or UTF8 encoding. [Table 3–8](#) compares AL16UTF16 and UTF8 encodings.

Table 3–8 Comparison of AL16UTF16 and UTF8 Encodings

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

For maximum reliability, Oracle recommends using the default AL16UTF16 encoding wherever practical. To use UTF encoding, specify it in the NATIONAL CHARACTER SET clause of the CREATE DATABASE statement.

To determine how many bytes a Unicode string needs, use the built-in function LENGTHB.

For more information about the NATIONAL CHARACTER SET clause of the CREATE DATABASE statement and the LENGTHB function, see *Oracle Database SQL Language Reference*.

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

NCHAR Datatype The NCHAR datatype stores fixed-length national character strings. Because this type can always accommodate multiple-byte characters, you can use it to store any Unicode character data.

The syntax for specifying an NCHAR data item is:

```
NCHAR [(maximum_size)]
```

For example:

NCHAR
NCHAR(100)

The *maximum_size* must be an integer literal, not a constant or variable. It represents the maximum number of characters, not the maximum number of bytes, which is 32,767. The largest *maximum_size* you can specify is 32767/2 with AL16UTF16 encoding and 32767/3 with UTF8 encoding. The default value is one.

The largest NCHAR value that you can insert into an NCHAR database column (using INSERT, UPDATE, or MERGE) is 2,000 bytes.

If the NCHAR value is shorter than the defined width of the NCHAR column, PL/SQL blank-pads the value to the defined width.

You can interchange CHAR and NCHAR values in statements and expressions. It is always safe to convert a CHAR value to an NCHAR value, but converting an NCHAR value to a CHAR value might cause data loss if the character set for the CHAR value cannot represent all the characters in the NCHAR value. Such data loss usually results in characters that look like question marks (?).

NVARCHAR2 Datatype The NVARCHAR2 datatype stores variable-length national character strings. Because this type can always accommodate multiple-byte characters, you can use it to store any Unicode character data.

The syntax for specifying an NVARCHAR2 data item is:

NVARCHAR2 (*maximum_size*)

For example:

NVARCHAR2(300)

The *maximum_size* must be an integer literal, not a constant or variable. It represents the maximum number of characters, not the maximum number of bytes, which is 32,767. The largest *maximum_size* you can specify is 32767/2 with AL16UTF16 encoding and 32767/3 with UTF8 encoding. The default value is one.

The largest NVARCHAR2 value that you can insert into an NVARCHAR2 database column (using INSERT, UPDATE, or MERGE) is 4,000 bytes.

You can interchange VARCHAR2 and NVARCHAR2 values in statements and expressions. It is always safe to convert a VARCHAR2 value to an NVARCHAR2 value, but converting an NVARCHAR2 value to a VARCHAR2 value might cause data loss if the character set for the VARCHAR2 value cannot represent all the characters in the NVARCHAR2 value. Such data loss usually results in characters that look like question marks (?).

LONG and LONG RAW Datatypes

Note: The LONG and LONG RAW datatypes are supported only for backward compatibility with existing applications. For new applications, use CLOB or NCLOB instead of LONG, and BLOB or BFILE instead of LONG RAW. Oracle recommends that you also replace existing LONG and LONG RAW datatypes with LOB datatypes. See ["Predefined PL/SQL Large Object \(LOB\) Datatypes"](#) on page 3-21.

The LONG datatype stores variable-length character strings. The LONG datatype is like the VARCHAR2 datatype, except that the maximum size of a LONG value is 32,760 bytes (as opposed to 32,767 bytes).

The `LONG RAW` datatype stores binary or byte strings. `LONG RAW` data is like `LONG` data, except that `LONG RAW` data is not interpreted by PL/SQL. The maximum size of a `LONG RAW` value is 32,760 bytes.

Because the maximum width of a `LONG` or `LONG RAW` database column is 2,147,483,648 bytes (2 GB), you can insert any `LONG` value into a `LONG` column and any `LONG RAW` value into a `LONG RAW` column. However, you cannot retrieve a value longer than 32,760 bytes from a `LONG` column into a `LONG` variable, or from a `LONG RAW` column into a `LONG RAW` variable.

`LONG` database columns can store text, arrays of characters, and even short documents.

You can reference `LONG` columns in the following:

- `INSERT` statements
- `UPDATE` statements
- Most `SELECT` statements

You cannot reference `LONG` columns in the following:

- Expressions
- SQL function calls
- Some SQL clauses (for example, `WHERE`, `GROUP BY`, and `CONNECT BY`)

For more information about referencing `LONG` columns in SQL statements, see *Oracle Database SQL Language Reference*.

ROWID and UROWID Datatypes

Internally, every database table has a `ROWID` pseudocolumn, which stores binary values called rowids. Each **rowid** represents the storage address of a row. A **physical rowid** identifies a row in an ordinary table. A **logical rowid** identifies a row in an index-organized table. The `ROWID` datatype can store only physical rowids, while the `UROWID` (**universal rowid**) datatype can store physical, logical, or foreign (non-Oracle) rowids.

Note: The `ROWID` datatype is supported only for backward compatibility with existing applications. For new applications, use the `UROWID` datatype.

Physical rowids are useful for fetching across commits, as in [Example 6-42](#) on page 6-39.

When you retrieve a rowid into a `ROWID` variable, you can use the built-in function `ROWIDTOCHAR`, which converts the binary value into an 18-byte character string. Conversely, the function `CHARTOROWID` converts a `ROWID` character string into a rowid. If the conversion fails because the character string does not represent a valid rowid, PL/SQL raises the predefined exception `SYS_INVALID_ROWID`. This also applies to implicit conversions.

To convert between `UROWID` variables and character strings, use regular assignment statements without any function call. The values are implicitly converted between `UROWID` and character types.

See Also:

- *Oracle Database Concepts* for general information about rowids
- *Oracle Database PL/SQL Packages and Types Reference* for information about the package `DBMS_ROWID`, whose subprograms enable you to manipulate rowids

Predefined PL/SQL BOOLEAN Datatype

The `BOOLEAN` datatype stores logical values, which you can use in logical operations. The logical values are the Boolean values `TRUE` and `FALSE` and the value `NULL` (which means a missing, unknown, or inapplicable value).

The syntax for specifying an `BOOLEAN` data item is:

```
BOOLEAN
```

SQL has no datatype equivalent to `BOOLEAN`; therefore you cannot do the following:

- Use `BOOLEAN` variables in SQL statements
- Use `BOOLEAN` parameters in PL/SQL functions invoked from SQL statements
- Use `BOOLEAN` parameters in built-in SQL functions (such as `TO_CHAR`)

You cannot insert the value `TRUE` or `FALSE` into a database column. You cannot retrieve the value of a database column into a `BOOLEAN` variable.

To represent `BOOLEAN` values in output, use `IF-THEN` or `CASE` constructs to translate `BOOLEAN` values into another type (for example, 0 or 1, 'Y' or 'N', 'true' or 'false').

Predefined PL/SQL Datetime and Interval Datatypes

The datatypes in this section let you store and manipulate dates, times, and intervals (periods of time). A variable that has a date and time datatype stores values called datetimes. A variable that has an interval datatype stores values called intervals. A datetime or interval consists of fields, which determine its value. The following list shows the valid values for each field:

Field Name	Valid Datetime Values	Valid Interval Values
<code>YEAR</code>	-4712 to 9999 (excluding year 0)	Any nonzero integer
<code>MONTH</code>	01 to 12	0 to 11
<code>DAY</code>	01 to 31 (limited by the values of <code>MONTH</code> and <code>YEAR</code> , according to the rules of the calendar for the locale)	Any nonzero integer
<code>HOUR</code>	00 to 23	0 to 23
<code>MINUTE</code>	00 to 59	0 to 59
<code>SECOND</code>	00 to 59.9(n), where 9(n) is the precision of time fractional seconds	0 to 59.9(n), where 9(n) is the precision of interval fractional seconds
<code>TIMEZONE_HOUR</code>	-12 to 14 (range accommodates daylight savings time changes)	Not applicable
<code>TIMEZONE_MINUTE</code>	00 to 59	Not applicable

Field Name	Valid Datetime Values	Valid Interval Values
TIMEZONE_REGION	Found in the dynamic performance view V\$TIMEZONE_NAMES	Not applicable
TIMEZONE_ABBR	Found in the dynamic performance view V\$TIMEZONE_NAMES	Not applicable

Except for `TIMESTAMP WITH LOCAL TIMEZONE`, these types are all part of the SQL92 standard. For information about datetime and interval format models, literals, time-zone names, and SQL functions, see *Oracle Database SQL Language Reference*.

Topics:

- [DATE Datatype](#)
- [TIMESTAMP Datatype](#)
- [TIMESTAMP WITH TIME ZONE Datatype](#)
- [TIMESTAMP WITH LOCAL TIME ZONE Datatype](#)
- [INTERVAL YEAR TO MONTH Datatype](#)
- [INTERVAL DAY TO SECOND Datatype](#)
- [Datetime and Interval Arithmetic](#)
- [Avoiding Truncation Problems Using Date and Time Subtypes](#)

DATE Datatype

You use the `DATE` datatype to store fixed-length datetimes, which include the time of day in seconds since midnight. The date portion defaults to the first day of the current month; the time portion defaults to midnight. The date function `SYSDATE` returns the current date and time.

- To compare dates for equality, regardless of the time portion of each date, use the function result `TRUNC(date_variable)` in comparisons, `GROUP BY` operations, and so on.
- To find just the time portion of a `DATE` variable, subtract the date portion: `date_variable - TRUNC(date_variable)`.

Valid dates range from January 1, 4712 BC to December 31, 9999 AD. A Julian date is the number of days since January 1, 4712 BC. Julian dates allow continuous dating from a common reference. You can use the date format model 'J' with the date functions `TO_DATE` and `TO_CHAR` to convert between `DATE` values and their Julian equivalents.

In date expressions, PL/SQL automatically converts character values in the default date format to `DATE` values. The default date format is set by the Oracle initialization parameter `NLS_DATE_FORMAT`. For example, the default might be 'DD-MON-YY', which includes a two-digit number for the day of the month, an abbreviation of the month name, and the last two digits of the year.

You can add and subtract dates. In arithmetic expressions, PL/SQL interprets integer literals as days. For example, `SYSDATE + 1` signifies the same time tomorrow.

TIMESTAMP Datatype

The datatype `TIMESTAMP`, which extends the datatype `DATE`, stores the year, month, day, hour, minute, and second. The syntax is:

```
TIMESTAMP[ (precision)
```

where the optional parameter *precision* specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..9. The default is 6.

The default timestamp format is set by the Oracle initialization parameter `NLS_TIMESTAMP_FORMAT`.

In [Example 3-4](#), you declare a variable of type `TIMESTAMP`, then assign a literal value to it. In the example, the fractional part of the seconds field is 0.275.

Example 3-4 Assigning a Literal Value to a TIMESTAMP Variable

```
DECLARE
    checkout TIMESTAMP(3);
BEGIN
    checkout := '22-JUN-2004 07:48:53.275';
    DBMS_OUTPUT.PUT_LINE( TO_CHAR(checkout));
END;
/
```

In [Example 3-5](#), the `SCN_TO_TIMESTAMP` and `TIMESTAMP_TO_SCN` functions are used to manipulate `TIMESTAMPS`.

Example 3-5 Using the SCN_TO_TIMESTAMP and TIMESTAMP_TO_SCN Functions

```
DECLARE
    right_now TIMESTAMP;
    yesterday TIMESTAMP;
    sometime TIMESTAMP;
    scn1 INTEGER;
    scn2 INTEGER;
    scn3 INTEGER;
BEGIN
    right_now := SYSTIMESTAMP; -- Get the current SCN
    scn1 := TIMESTAMP_TO_SCN(right_now);
    DBMS_OUTPUT.PUT_LINE('Current SCN is ' || scn1);
    yesterday := right_now - 1;
    -- Get the SCN from exactly 1 day ago
    scn2 := TIMESTAMP_TO_SCN(yesterday);
    DBMS_OUTPUT.PUT_LINE('SCN from yesterday is ' || scn2);
    -- Find an arbitrary SCN somewhere between yesterday and today
    -- In a real program we would have stored the SCN
    -- at some significant moment
    scn3 := (scn1 + scn2) / 2;
    sometime := SCN_TO_TIMESTAMP(scn3);
    -- What time was that SCN was in effect?
    DBMS_OUTPUT.PUT_LINE
        ('SCN ' || scn3 || ' was in effect at ' ||
         TO_CHAR(sometime));
END;
/
```

TIMESTAMP WITH TIME ZONE Datatype

The datatype `TIMESTAMP WITH TIME ZONE`, which extends the datatype `TIMESTAMP`, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC)—formerly Greenwich Mean Time. The syntax is:

```
TIMESTAMP[(precision)] WITH TIME ZONE
```

where the optional parameter *precision* specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..9. The default is 6.

The default timestamp with time zone format is set by the Oracle initialization parameter `NLS_TIMESTAMP_TZ_FORMAT`.

In [Example 3-6](#), you declare a variable of type `TIMESTAMP WITH TIME ZONE`, then assign a literal value to it:

Example 3-6 Assigning a Literal to a TIMESTAMP WITH TIME ZONE Variable

```
DECLARE
  logoff TIMESTAMP(3) WITH TIME ZONE;
BEGIN
  logoff := '10-OCT-2004 09:42:37.114 AM +02:00';
  DBMS_OUTPUT.PUT_LINE( TO_CHAR(logoff));
END;
/
```

In this example, the time-zone displacement is `+02:00`.

You can also specify the time zone by using a symbolic name. The specification can include a long form such as `'US/Pacific'`, an abbreviation such as `'PDT'`, or a combination. For example, the following literals all represent the same time. The third form is most reliable because it specifies the rules to follow at the point when switching to daylight savings time.

```
TIMESTAMP '15-APR-2004 8:00:00 -8:00'
TIMESTAMP '15-APR-2004 8:00:00 US/Pacific'
TIMESTAMP '31-OCT-2004 01:30:00 US/Pacific PDT'
```

You can find the available names for time zones in the `TIMEZONE_REGION` and `TIMEZONE_ABBR` columns of the static data dictionary view `V$TIMEZONE_NAMES`.

Two `TIMESTAMP WITH TIME ZONE` values are considered identical if they represent the same instant in UTC, regardless of their time-zone displacements. For example, the following two values are considered identical because, in UTC, 8:00 AM Pacific Standard Time is the same as 11:00 AM Eastern Standard Time:

```
'29-AUG-2004 08:00:00 -8:00'
'29-AUG-2004 11:00:00 -5:00'
```

TIMESTAMP WITH LOCAL TIME ZONE Datatype

The datatype `TIMESTAMP WITH LOCAL TIME ZONE`, which extends the datatype `TIMESTAMP`, includes a time-zone displacement. The time-zone displacement is the difference (in hours and minutes) between local time and Coordinated Universal Time (UTC)—formerly Greenwich Mean Time. You can also use named time zones, as with `TIMESTAMP WITH TIME ZONE`.

The syntax is:

```
TIMESTAMP[(precision)] WITH LOCAL TIME ZONE
```


where the optional parameter *precision* specifies the number of digits in the fractional part of the seconds field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..9. The default is 6.

This datatype differs from `TIMESTAMP WITH TIME ZONE` in that when you insert a value into a database column, the value is normalized to the database time zone, and the time-zone displacement is not stored in the column. When you retrieve the value, Oracle returns it in your local session time zone.

In [Example 3-7](#), you declare a variable of type `TIMESTAMP WITH LOCAL TIME ZONE`:

Example 3-7 Assigning a Literal Value to a `TIMESTAMP WITH LOCAL TIME ZONE`

```
DECLARE
    logoff TIMESTAMP(3) WITH LOCAL TIME ZONE;
BEGIN
    -- logoff := '10-OCT-2004 09:42:37.114 AM +02:00';
    -- raises an error
    logoff := '10-OCT-2004 09:42:37.114 AM ';
    -- okay without displacement
    DBMS_OUTPUT.PUT_LINE( TO_CHAR(logoff));
END;
/
```

INTERVAL YEAR TO MONTH Datatype

You use the datatype `INTERVAL YEAR TO MONTH` to store and manipulate intervals of years and months. The syntax is:

```
INTERVAL YEAR[(precision)] TO MONTH
```

where *precision* specifies the number of digits in the years field. You cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..4. The default is 2.

In [Example 3-8](#), you declare a variable of type `INTERVAL YEAR TO MONTH`, then assign a value of 101 years and 3 months to it:

Example 3-8 Assigning Literals to an `INTERVAL YEAR TO MONTH` Variable

```
DECLARE
    lifetime INTERVAL YEAR(3) TO MONTH;
BEGIN
    lifetime := INTERVAL '101-3' YEAR TO MONTH; -- interval literal
    lifetime := '101-3'; -- implicit conversion from character type
    lifetime := INTERVAL '101' YEAR; -- Can specify just the years
    lifetime := INTERVAL '3' MONTH; -- Can specify just the months
END;
/
```

INTERVAL DAY TO SECOND Datatype

You use the datatype `INTERVAL DAY TO SECOND` to store and manipulate intervals of days, hours, minutes, and seconds. The syntax is:

```
INTERVAL DAY[(leading_precision)
             TO SECOND (fractional_seconds_precision)
```

where *leading_precision* and *fractional_seconds_precision* specify the number of digits in the days field and seconds field, respectively. In both cases, you

cannot use a symbolic constant or variable to specify the precision; you must use an integer literal in the range 0..9. The defaults are 2 and 6, respectively.

In [Example 3-9](#), you declare a variable of type `INTERVAL DAY TO SECOND`:

Example 3-9 Assigning Literals to an `INTERVAL DAY TO SECOND` Variable

```
DECLARE
    lag_time INTERVAL DAY(3) TO SECOND(3);
BEGIN
    lag_time := '7 09:24:30';
    IF lag_time > INTERVAL '6' DAY THEN
        DBMS_OUTPUT.PUT_LINE ( 'Greater than 6 days');
    ELSE
        DBMS_OUTPUT.PUT_LINE ( 'Less than 6 days');
    END IF;
END;
/
```

Datetime and Interval Arithmetic

PL/SQL lets you construct datetime and interval expressions. The following list shows the operators that you can use in such expressions:

Operand 1	Operator	Operand 2	Result Type
datetime	+	interval	datetime
datetime	-	interval	datetime
interval	+	datetime	datetime
datetime	-	datetime	interval
interval	+	interval	interval
interval	-	interval	interval
interval	*	numeric	interval
numeric	*	interval	interval
interval	/	numeric	interval

You can use SQL functions to perform arithmetic operations on datetime values. For details, see *Oracle Database SQL Language Reference*.

Avoiding Truncation Problems Using Date and Time Subtypes

The default precisions for some of the date and time types are less than the maximum precision. For example, the default for `DAY TO SECOND` is `DAY(2) TO SECOND(6)`, while the highest precision is `DAY(9) TO SECOND(9)`. To avoid truncation when assigning variables and passing subprogram parameters of these types, you can declare variables and subprogram parameters of the following subtypes, which use the maximum values for precision:

```
TIMESTAMP_UNCONSTRAINED
TIMESTAMP_TZ_UNCONSTRAINED
TIMESTAMP_LTZ_UNCONSTRAINED
YMINTERVAL_UNCONSTRAINED
DSINTERVAL_UNCONSTRAINED
```

Predefined PL/SQL Large Object (LOB) Datatypes

Large object (LOB) datatypes reference large objects that are stored separately from other data items, such as text, graphic images, video clips, and sound waveforms. LOB datatypes allow efficient, random, piecewise access to this data. Predefined PL/SQL LOB datatypes are listed and described in [Table 3–9](#).

Table 3–9 Predefined PL/SQL Large Object (LOB) Datatypes

Datatype	Description	Size
BFILE	Used to store large binary objects in operating system files outside the database.	System-dependent. Cannot exceed 4 gigabytes (GB).
BLOB	Used to store large binary objects in the database.	8 to 128 terabytes (TB)
CLOB	Used to store large blocks of character data in the database.	8 to 128 TB
NCLOB	Used to store large blocks of NCHAR data in the database.	8 to 128 TB

LOB Locators

To reference a large object that is stored in an external file, a LOB datatype uses a **LOB locator**, which is stored in an external file, either inside the row (**in-line**) or outside the row (**out-of-line**). In the external file, LOB locators are in columns of the types BFILE, BLOB, CLOB, and NCLOB.

PL/SQL operates on large objects through their LOB locators. For example, when you select a BLOB column value, PL/SQL returns only its locator. If PL/SQL returned the locator during a transaction, the locator includes a transaction ID, so you cannot use that locator to update that large object in another transaction. Likewise, you cannot save a locator during one session and then use it in another session.

Differences Between LOB Datatypes and LONG and LONG RAW Datatypes

LOB datatypes differ from LONG and LONG RAW datatypes in the following ways:

Difference	LOB Datatypes	LONG and LONG RAW Datatypes
Support	Functionality enhanced in every release.	Functionality static. Supported only for backward compatibility with existing applications.
Maximum size	8 to 128 TB	2 GB
Access	Random	Sequential
Can be object type attribute	BFILE, BLOB, CLOB: Yes NCLOB: No	No

For more information about LONG and LONG RAW datatypes, see "[LONG and LONG RAW Datatypes](#)" on page 3-13.

For more information on LOBs, see *Oracle Database SecureFiles and Large Objects Developer's Guide*.

Topics:

- [BFILE Datatype](#)
- [BLOB Datatype](#)
- [CLOB Datatype](#)

- [NCLOB Datatype](#)

BFILE Datatype

You use the BFILE datatype to store large binary objects in operating system files outside the database. Every BFILE variable stores a file locator, which points to a large binary file on the server. The locator includes a directory alias, which specifies a full path name. Logical path names are not supported.

BFILES are read-only, so you cannot modify them. Your DBA makes sure that a given BFILE exists and that Oracle has read permissions on it. The underlying operating system maintains file integrity.

BFILES do not participate in transactions, are not recoverable, and cannot be replicated. The maximum number of open BFILES is set by the Oracle initialization parameter `SESSION_MAX_OPEN_FILES`, which is system dependent.

BLOB Datatype

You use the BLOB datatype to store large binary objects in the database, in-line or out-of-line. Every BLOB variable stores a locator, which points to a large binary object.

BLOBs participate fully in transactions, are recoverable, and can be replicated. Changes made by package `DBMS_LOB` can be committed or rolled back. BLOB locators can span transactions (for reads only), but they cannot span sessions.

CLOB Datatype

You use the CLOB datatype to store large blocks of character data in the database, in-line or out-of-line. Both fixed-width and variable-width character sets are supported. Every CLOB variable stores a locator, which points to a large block of character data.

CLOBs participate fully in transactions, are recoverable, and can be replicated. Changes made by package `DBMS_LOB` can be committed or rolled back. CLOB locators can span transactions (for reads only), but they cannot span sessions.

NCLOB Datatype

You use the NCLOB datatype to store large blocks of NCHAR data in the database, in-line or out-of-line. Both fixed-width and variable-width character sets are supported. Every NCLOB variable stores a locator, which points to a large block of NCHAR data.

NCLOBs participate fully in transactions, are recoverable, and can be replicated. Changes made by package `DBMS_LOB` can be committed or rolled back. NCLOB locators can span transactions (for reads only), but they cannot span sessions.

User-Defined PL/SQL Subtypes

A **subtype** is a subset of another datatype, which is called its **base type**. A subtype has the same valid operations as its base type, but only a subset of its valid values. Subtypes can increase reliability, provide compatibility with ANSI/ISO types, and improve readability by indicating the intended use of constants and variables.

PL/SQL predefines several subtypes in package `STANDARD`. For example, PL/SQL predefines the subtypes `CHARACTER` and `INTEGER` as follows:

```
SUBTYPE CHARACTER IS CHAR;
SUBTYPE INTEGER IS NUMBER(38,0); -- allows only whole numbers
```

The subtype `CHARACTER` specifies the same set of values as its base type `CHAR`, so `CHARACTER` is an unconstrained subtype. But, the subtype `INTEGER` specifies only a subset of the values of its base type `NUMBER`, so `INTEGER` is a constrained subtype.

Topics:

- [Defining Subtypes](#)
- [Using Subtypes](#)

Defining Subtypes

You can define your own subtypes in the declarative part of any PL/SQL block, subprogram, or package using the following syntax:

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

where *subtype_name* is a type specifier used in subsequent declarations, *base_type* is any scalar or user-defined PL/SQL datatype, and *constraint* applies only to base types that can specify precision and scale or a maximum size. Note that a default value is not permitted; see [Example 3–13](#) on page 3-25.

Examples:

```
DECLARE
  SUBTYPE BirthDate IS DATE NOT NULL; -- based on DATE type
  SUBTYPE Counter IS NATURAL;         -- based on NATURAL subtype
  TYPE NameList IS TABLE OF VARCHAR2(10);
  SUBTYPE DutyRoster IS NameList;     -- based on TABLE type
  TYPE TimeRec IS RECORD (minutes INTEGER, hours INTEGER);
  SUBTYPE FinishTime IS TimeRec;     -- based on RECORD type
  SUBTYPE ID_Num IS employees.employee_id%TYPE;
  -- based on column type
```

You can use `%TYPE` or `%ROWTYPE` to specify the base type. When `%TYPE` provides the datatype of a database column, the subtype inherits the size constraint (if any) of the column. The subtype does not inherit other kinds of column constraints, such as `NOT NULL` or check constraint, or the default value, as shown in [Example 3–14](#) on page 3-25. For more information, see "[Using the %TYPE Attribute](#)" on page 2-10 and "[Using the %ROWTYPE Attribute](#)" on page 2-12.

Using Subtypes

After you define a subtype, you can declare items of that type. In the following example, you declare a variable of type `Counter`. Notice how the subtype name indicates the intended use of the variable.

```
DECLARE
  SUBTYPE Counter IS NATURAL;
  rows Counter;
```

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

```
DECLARE
  SUBTYPE Accumulator IS NUMBER;
  total Accumulator(7,2);
```

Subtypes can increase reliability by detecting out-of-range values. In [Example 3–10](#), you restrict the subtype `pinteger` to storing integers in the range -9..9. If your

program tries to store a number outside that range in a `pinteger` variable, PL/SQL raises an exception.

Example 3–10 Using Ranges with Subtypes

```
DECLARE
  v_sqlerrm VARCHAR2(64);
  SUBTYPE pinteger IS PLS_INTEGER RANGE -9..9;
  y_axis pinteger;
  PROCEDURE p (x IN pinteger) IS
    BEGIN DBMS_OUTPUT.PUT_LINE (x); END p;
BEGIN
  y_axis := 9; -- valid, in range
  p(10); -- invalid for procedure p
  EXCEPTION
    WHEN OTHERS THEN
      v_sqlerrm := SUBSTR(SQLERRM, 1, 64);
      DBMS_OUTPUT.PUT_LINE('Error: ' || v_sqlerrm);
END;
/
```

Topics:

- [Type Compatibility with Subtypes](#)
- [Constraints and Default Values with Subtypes](#)

Type Compatibility with Subtypes

An unconstrained subtype is interchangeable with its base type. For example, given the following declarations, the value of `amount` can be assigned to `total` without conversion:

Example 3–11 Type Compatibility with the NUMBER Datatype

```
DECLARE
  SUBTYPE Accumulator IS NUMBER;
  amount NUMBER(7,2);
  total Accumulator;
BEGIN
  amount := 10000.50;
  total := amount;
END;
/
```

Different subtypes are interchangeable if they have the same base type:

```
DECLARE
  SUBTYPE b1 IS BOOLEAN;
  SUBTYPE b2 IS BOOLEAN;
  finished b1; -- Different subtypes,
  debugging b2; -- both based on BOOLEAN.
BEGIN
  finished :=FALSE;
  debugging := finished; -- They can be assigned to each other.
END;
/
```

Different subtypes are also interchangeable if their base types are in the same datatype family. For example, given the following declarations, the value of `verb` can be assigned to `sentence`:

```
DECLARE
  SUBTYPE Word IS CHAR(15);
  SUBTYPE Text IS VARCHAR2(1500);
  verb      Word;          -- Different subtypes
  sentence Text(150);     -- of types from the same family
BEGIN
  verb := 'program';
  sentence := verb;      -- can be assigned, if not too long.
END;
/
```

Constraints and Default Values with Subtypes

The examples in this section illustrate the use of constraints and default values with subtypes. In [Example 3–12](#), the procedure enforces the NOT NULL constraint, but does not enforce the size.

Example 3–12 Constraints Inherited by Subprograms

```
DECLARE
  SUBTYPE v_word IS VARCHAR2(10) NOT NULL;
  verb      v_word := 'run';
  noun      VARCHAR2(10) := NULL;
  PROCEDURE word_to_upper (w IN v_word) IS
  BEGIN
    DBMS_OUTPUT.PUT_LINE (UPPER(w));
  END word_to_upper;
BEGIN
  word_to_upper('run_over_ten_characters');
  -- size constraint is not enforced
-- word_to_upper(noun); invalid, NOT NULL constraint is enforced
END;
/
```

[Example 3–13](#) shows to assign a default value to a subtype variable.

Example 3–13 Default Value with Subtypes

```
DECLARE
  SUBTYPE v_word IS VARCHAR2(10) NOT NULL;
  -- invalid to put default here
  verb v_word := 'verb';
  noun v_word := 'noun';
BEGIN
  DBMS_OUTPUT.PUT_LINE (UPPER(verb));
  DBMS_OUTPUT.PUT_LINE (UPPER(noun));
END;
/
```

[Example 3–14](#) shows how column constraints are inherited by subtypes.

Example 3–14 Using SUBTYPE with %TYPE and %ROWTYPE

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

```
DECLARE
  SUBTYPE v_empid_subtype IS employees_temp.empid%TYPE;
  SUBTYPE v_deptid_subtype IS employees_temp.deptid%TYPE;
  SUBTYPE v_deptname_subtype IS employees_temp.deptname%TYPE;
  SUBTYPE v_emprec_subtype IS employees_temp%ROWTYPE;
  v_empid      v_empid_subtype;
  v_deptid     v_deptid_subtype;
  v_deptname   v_deptname_subtype;
  v_emprec     v_emprec_subtype;
BEGIN
  v_empid := NULL;
  -- this works, null constraint is not inherited
  -- v_empid := 10000002; -- invalid, number precision too large
  v_deptid := 50; -- this works, check constraint is not inherited
  -- the default value is not inherited in the following
  DBMS_OUTPUT.PUT_LINE('v_deptname: ' || v_deptname);
  v_emprec.empid := NULL;
  -- this works, null constraint is not inherited
  -- v_emprec.empid := 10000002;
  -- invalid, number precision too large
  v_emprec.deptid := 50;
  -- this works, check constraint is not inherited
  -- the default value is not inherited in the following
  DBMS_OUTPUT.PUT_LINE('v_emprec.deptname: ' || v_emprec.deptname);
END;
/
```

PL/SQL Datatype Conversion

Sometimes it is necessary to convert a value from one datatype to another. For example, to use a `DATE` value in a report, you must convert it to a character string. PL/SQL supports both explicit and implicit (automatic) datatype conversion.

For best reliability and maintainability, use explicit conversion. Implicit conversion is context-sensitive and not always predictable, and its rules might change in later software releases. Implicit conversion can also be slower than explicit conversion.

Topics:

- [Explicit Conversion](#)
- [Implicit Conversion](#)

Explicit Conversion

To explicitly convert values from one datatype to another, you use built-in functions, which are described in *Oracle Database SQL Language Reference*. For example, to convert a `CHAR` value to a `DATE` or `NUMBER` value, you use the function `TO_DATE` or `TO_NUMBER`, respectively. Conversely, to convert a `DATE` or `NUMBER` value to a `CHAR` value, you use the function `TO_CHAR`.

Explicit conversion can prevent errors or unexpected results. For example:

- Using the concatenation operator (`||`) to concatenate a string and an arithmetic expression can produce an error, which you can prevent by using the `TO_CHAR` function to convert the arithmetic expression to a string before concatenation.
- Relying on language settings in the database for the format of a `DATE` value can produce unexpected results, which you can prevent by using the `TO_CHAR` function and specifying the format that you want.

Implicit Conversion

Sometimes PL/SQL can convert a value from one datatype to another automatically. This is called implicit conversion, and the datatypes are called **compatible**. When two datatypes are compatible, you can use a value of one type where a value of the other type is expected. For example, you can pass a numeric literal to a subprogram that expects a string value, and the subprogram receives the string representation of the number.

In [Example 3–15](#), the CHAR variables `start_time` and `finish_time` store string values representing the number of seconds past midnight. The difference between those values can be assigned to the NUMBER variable `elapsed_time`, because PL/SQL converts the CHAR values to NUMBER values automatically.

Example 3–15 Implicit Conversion

```
DECLARE
  start_time  CHAR(5);
  finish_time CHAR(5);
  elapsed_time NUMBER(5);
BEGIN
  -- Get system time as seconds past midnight:
  SELECT TO_CHAR(SYSDATE, 'SSSS') INTO start_time FROM sys.DUAL;
  -- Processing done here

  -- Get system time again:
  SELECT TO_CHAR(SYSDATE, 'SSSS') INTO finish_time FROM sys.DUAL;

  -- Compute and report elapsed time in seconds:
  elapsed_time := finish_time - start_time;
  DBMS_OUTPUT.PUT_LINE ('Elapsed time: ' || TO_CHAR(elapsed_time));
END;
/
```

If you select a value from a column of one datatype, and assign that value to a variable of another datatype, PL/SQL converts the value to the datatype of the variable. This happens, for example, when you select a DATE column value into a VARCHAR2 variable.

If you assign the value of a variable of one database type to a column of another database type, PL/SQL converts the value of the variable to the datatype of the column.

If PL/SQL cannot determine which implicit conversion is needed, you get a compilation error. In such cases, you must use explicit conversion.

[Table 3–10](#) shows which implicit conversions PL/SQL can do. Note that:

- [Table 3–10](#) lists only datatypes that have different representations. Types that have the same representation, such as PLS_INTEGER and BINARY_INTEGER, CLOB and NCLOB, CHAR and NCHAR, and VARCHAR and NVARCHAR2, can be substituted for each other.
- It is your responsibility to ensure that specific values are convertible. For example, PL/SQL can convert the CHAR value '02-JUN-92' to a DATE value but cannot convert the CHAR value 'YESTERDAY' to a DATE value. Similarly, PL/SQL cannot convert a VARCHAR2 value containing alphabetic characters to a NUMBER value.
- Regarding date, time, and interval datatypes:

- Conversion rules for the DATE datatype also apply to the datatypes TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL DAY TO SECOND, and INTERVAL YEAR TO MONTH. However, because of their different internal representations, these types cannot always be converted to each other. For details of implicit conversions between different date and time types, see *Oracle Database SQL Language Reference*.
- To implicitly convert a DATE value to a CHAR or VARCHAR2 value, PL/SQL invokes the function TO_CHAR, which returns a character string in the default date format. To get other information, such as the time or Julian date, invoke TO_CHAR explicitly with a format mask.
- When you insert a CHAR or VARCHAR2 value into a DATE column, PL/SQL implicitly converts the CHAR or VARCHAR2 value to a DATE value by invoking the function TO_DATE, which expects its parameter to be in the default date format. To insert dates in other formats, invoke TO_DATE explicitly with a format mask.
- Regarding LOB datatypes:
 - Converting between CLOB and NCLOB values can be expensive. To make clear that you intend this conversion, use the explicit conversion functions TO_CLOB and TO_NCLOB.
 - Implicit conversion between CLOB values and CHAR and VARCHAR2 values, and between BLOB values and RAW values, lets you use LOB datatypes in most SQL and PL/SQL statements and functions. However, to read, write, and do piecewise operations on LOB values, you must use DBMS_LOB package subprograms, which are described in *Oracle Database PL/SQL Packages and Types Reference*.
- Regarding RAW and LONG RAW datatypes:
 - LONG RAW is supported only for backward compatibility with existing applications. For more information, see "[LONG and LONG RAW Datatypes](#)" on page 3-13.
 - When you select a RAW or LONG RAW column value into a CHAR or VARCHAR2 variable, PL/SQL must convert the internal binary value to a character value. PL/SQL does this by returning each binary byte of RAW or LONG RAW data as a pair of characters. Each character represents the hexadecimal equivalent of a **nibble** (half a byte). For example, PL/SQL returns the binary byte 11111111 as the pair of characters 'FF'. The function RAWTOHEX does the same conversion.
 - Conversion is also necessary when you insert a CHAR or VARCHAR2 value into a RAW or LONG RAW column. Each pair of characters in the variable must represent the hexadecimal equivalent of a binary byte; otherwise, PL/SQL raises an exception.
 - When a LONG value appears in a SQL statement, PL/SQL binds the LONG value as a VARCHAR2 value. However, if the length of the bound VARCHAR2 value exceeds the maximum width of a VARCHAR2 column (4,000 bytes), Oracle converts the bind type to LONG automatically, and then issues an error message because you cannot pass LONG values to a SQL function.

Table 3–10 Possible Implicit PL/SQL Datatype Conversions

From:	To:									
	BLOB	CHAR	CLOB	DATE	LONG	NUMBER	PLS_INTEGER	RAW	UROWID	VARCHAR2
BLOB								Yes		
CHAR			Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
CLOB		Yes								Yes
DATE		Yes			Yes					Yes
LONG		Yes						Yes		Yes
NUMBER		Yes			Yes		Yes			Yes
PLS_INTEGER		Yes			Yes	Yes				Yes
RAW	Yes	Yes			Yes					Yes
UROWID		Yes								Yes
VARCHAR2		Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	

Using PL/SQL Control Structures

This chapter shows you how to structure the flow of control through a PL/SQL program. PL/SQL provides conditional tests, loops, and branches that let you produce well-structured programs.

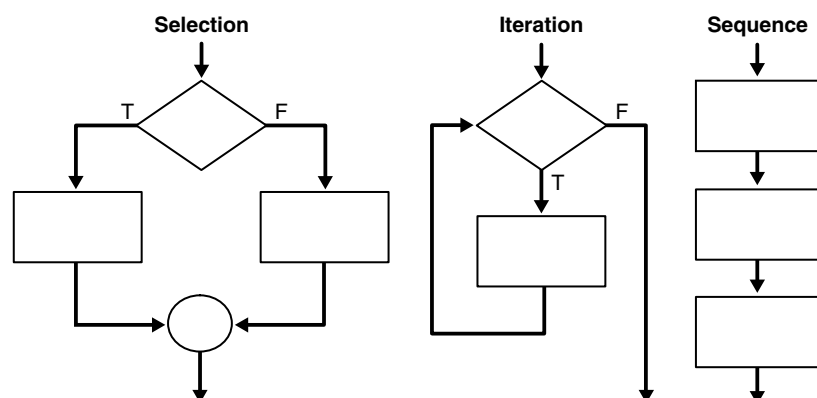
Topics:

- [Overview of PL/SQL Control Structures](#)
- [Testing Conditions \(IF and CASE Statements\)](#)
- [Controlling Loop Iterations \(LOOP, EXIT, and CONTINUE Statements\)](#)
- [Sequential Control \(GOTO and NULL Statements\)](#)

Overview of PL/SQL Control Structures

Procedural computer programs use the basic control structures shown in [Figure 4-1](#).

Figure 4-1 Control Structures



The selection structure tests a condition, then executes one sequence of statements instead of another, depending on whether the condition is true or false. A condition is any variable or expression that returns a `BOOLEAN` value (`TRUE` or `FALSE`). The iteration structure executes a sequence of statements repeatedly as long as a condition holds true. The sequence structure simply executes a sequence of statements in the order in which they occur.

Testing Conditions (IF and CASE Statements)

The IF statement executes a sequence of statements depending on the value of a condition. There are three forms of IF statements: IF-THEN, IF-THEN-ELSE, and IF-THEN-ELSIF. For a description of the syntax of the IF statement, see "[IF Statement](#)" on page 13-80.

The CASE statement is a compact way to evaluate a single condition and choose between many alternative actions. It makes sense to use CASE when there are three or more alternatives to choose from. For a description of the syntax of the CASE statement, see "[CASE Statement](#)" on page 13-16.

Topics:

- [Using the IF-THEN Statement](#)
- [Using the IF-THEN-ELSE Statement](#)
- [Using the IF-THEN-ELSIF Statement](#)
- [Using the Simple CASE Statement](#)
- [Using the Searched CASE Statement](#)
- [Guidelines for IF and CASE Statements](#)

Using the IF-THEN Statement

The simplest form of IF statement associates a condition with a sequence of statements enclosed by the keywords THEN and END IF (not ENDIF) as illustrated in [Example 4-1](#).

The sequence of statements is executed only if the condition is TRUE. If the condition is FALSE or NULL, the IF statement does nothing. In either case, control passes to the next statement.

Example 4-1 Using a Simple IF-THEN Statement

```
DECLARE
  sales  NUMBER(8,2) := 10100;
  quota  NUMBER(8,2) := 10000;
  bonus  NUMBER(6,2);
  emp_id NUMBER(6) := 120;
BEGIN
  IF sales > (quota + 200) THEN
    bonus := (sales - quota)/4;
    UPDATE employees SET salary =
      salary + bonus WHERE employee_id = emp_id;
  END IF;
END;
/
```

Using the IF-THEN-ELSE Statement

The second form of IF statement adds the keyword ELSE followed by an alternative sequence of statements, as shown in [Example 4-2](#).

The statements in the ELSE clause are executed only if the condition is FALSE or NULL. The IF-THEN-ELSE statement ensures that one or the other sequence of statements is executed. In the [Example 4-2](#), the first UPDATE statement is executed when the

condition is TRUE, and the second UPDATE statement is executed when the condition is FALSE or NULL.

Example 4–2 Using a Simple IF-THEN-ELSE Statement

```
DECLARE
  sales  NUMBER(8,2) := 12100;
  quota  NUMBER(8,2) := 10000;
  bonus  NUMBER(6,2);
  emp_id NUMBER(6) := 120;
BEGIN
  IF sales > (quota + 200) THEN
    bonus := (sales - quota)/4;
  ELSE
    bonus := 50;
  END IF;
  UPDATE employees SET salary =
    salary + bonus WHERE employee_id = emp_id;
END;
/
```

IF statements can be nested as shown in [Example 4–3](#).

Example 4–3 Nested IF Statements

```
DECLARE
  sales  NUMBER(8,2) := 12100;
  quota  NUMBER(8,2) := 10000;
  bonus  NUMBER(6,2);
  emp_id NUMBER(6) := 120;
BEGIN
  IF sales > (quota + 200) THEN
    bonus := (sales - quota)/4;
  ELSE
    IF sales > quota THEN
      bonus := 50;
    ELSE
      bonus := 0;
    END IF;
  END IF;
  UPDATE employees SET salary =
    salary + bonus WHERE employee_id = emp_id;
END;
/
```

Using the IF-THEN-ELSIF Statement

Sometimes you want to choose between several alternatives. You can use the keyword ELSIF (not ELSEIF or ELSE IF) to introduce additional conditions, as shown in [Example 4–4](#).

If the first condition is FALSE or NULL, the ELSIF clause tests another condition. An IF statement can have any number of ELSIF clauses; the final ELSE clause is optional. Conditions are evaluated one by one from top to bottom. If any condition is TRUE, its associated sequence of statements is executed and control passes to the next statement. If all conditions are false or NULL, the sequence in the ELSE clause is executed, as shown in [Example 4–4](#).

Example 4-4 Using the IF-THEN-ELSEIF Statement

```

DECLARE
  sales  NUMBER(8,2) := 20000;
  bonus  NUMBER(6,2);
  emp_id NUMBER(6) := 120;
BEGIN
  IF sales > 50000 THEN
    bonus := 1500;
  ELSIF sales > 35000 THEN
    bonus := 500;
  ELSE
    bonus := 100;
  END IF;
  UPDATE employees SET salary =
    salary + bonus WHERE employee_id = emp_id;
END;
/

```

If the value of `sales` is larger than 50000, the first and second conditions are TRUE. Nevertheless, `bonus` is assigned the proper value of 1500 because the second condition is never tested. When the first condition is TRUE, its associated statement is executed and control passes to the INSERT statement.

Another example of an IF-THEN-ELSE statement is [Example 4-5](#).

Example 4-5 Extended IF-THEN 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;
/

```

Using the Simple CASE Statement

Like the IF statement, the CASE statement selects one sequence of statements to execute. However, to select the sequence, the CASE statement uses a selector rather than multiple Boolean expressions. A selector is an expression whose value is used to select one of several alternatives.

To compare the IF and CASE statements, consider the code in [Example 4-5](#) that outputs descriptions of school grades. Note the five Boolean expressions. In each instance, we test whether the same variable, `grade`, is equal to one of five values: 'A', 'B', 'C', 'D', or 'F'. You can rewrite the code in [Example 4-5](#) using the CASE statement, as shown in [Example 4-6](#).

Example 4-6 Using the CASE-WHEN 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;
/

```

The CASE statement is more readable and more efficient. When possible, rewrite lengthy IF-THEN-ELSIF statements as CASE statements.

The CASE statement begins with the keyword CASE. The keyword is followed by a selector, which is the variable `grade` in the last example. The selector expression can be arbitrarily complex. For example, it can contain function calls. Usually, however, it consists of a single variable. The selector expression is evaluated only once. The value it yields can have any PL/SQL datatype other than BLOB, BFILE, an object type, a PL/SQL record, an index-by-table, a varray, or a nested table.

The selector is followed by one or more WHEN clauses, which are checked sequentially. The value of the selector determines which clause is executed. If the value of the selector equals the value of a WHEN-clause expression, that WHEN clause is executed. For example, in the last example, if `grade` equals 'C', the program outputs 'Good'. Execution never falls through; if any WHEN clause is executed, control passes to the next statement.

The ELSE clause works similarly to the ELSE clause in an IF statement. In the last example, if the grade is not one of the choices covered by a WHEN clause, the ELSE clause is selected, and the phrase 'No such grade' is output. The ELSE clause is optional. However, if you omit the ELSE clause, PL/SQL adds the following implicit ELSE clause:

```
ELSE RAISE CASE_NOT_FOUND;
```

There is always a default action, even when you omit the ELSE clause. If the CASE statement does not match any of the WHEN clauses and you omit the ELSE clause, PL/SQL raises the predefined exception `CASE_NOT_FOUND`.

The keywords END CASE terminate the CASE statement. These two keywords must be separated by a space. The CASE statement has the following form:

Like PL/SQL blocks, CASE statements can be labeled. The label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the CASE statement. Optionally, the label name can also appear at the end of the CASE statement.

Exceptions raised during the execution of a CASE statement are handled in the usual way. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram.

An alternative to the CASE statement is the CASE expression, where each WHEN clause is an expression. For details, see ["CASE Expressions"](#) on page 2-28.

Using the Searched CASE Statement

PL/SQL also provides a searched CASE statement, similar to the simple CASE statement, which has the form shown in [Example 4-7](#).

The searched CASE statement has no selector. Also, its WHEN clauses contain search conditions that yield a Boolean value, not expressions that can yield a value of any type, as shown in [Example 4-7](#).

Example 4-7 Using the 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;
-- rather than using ELSE in CASE, could use:
-- EXCEPTION
--   WHEN CASE_NOT_FOUND THEN
--     DBMS_OUTPUT.PUT_LINE('No such grade');
/
```

The search conditions are evaluated sequentially. The Boolean value of each search condition determines which WHEN clause is executed. If a search condition yields TRUE, its WHEN clause is executed. If any WHEN clause is executed, control passes to the next statement, so subsequent search conditions are not evaluated.

If none of the search conditions yields TRUE, the ELSE clause is executed. The ELSE clause is optional. However, if you omit the ELSE clause, PL/SQL adds the following implicit ELSE clause:

```
ELSE RAISE CASE_NOT_FOUND;
```

Exceptions raised during the execution of a searched CASE statement are handled in the usual way. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram.

Guidelines for IF and CASE Statements

Avoid clumsy IF statements like those in the following example:

```
IF new_balance < minimum_balance THEN
  overdrawn := TRUE;
ELSE
  overdrawn := FALSE;
END IF;
IF overdrawn = TRUE THEN
  RAISE insufficient_funds;
END IF;
```

The value of a Boolean expression can be assigned directly to a Boolean variable. You can replace the first IF statement with a simple assignment:

```
overdrawn := new_balance < minimum_balance;
```

A Boolean variable is itself either true or false. You can simplify the condition in the second IF statement:

```
IF overdrawn THEN ...
```

When possible, use the ELSIF clause instead of nested IF statements. Your code will be easier to read and understand. Compare the following IF statements:

```
IF condition1 THEN statement1;
  ELSE IF condition2 THEN statement2;
    ELSE IF condition3 THEN statement3; END IF;
  END IF;
END IF;
IF condition1 THEN statement1;
  ELSEIF condition2 THEN statement2;
  ELSEIF condition3 THEN statement3;
END IF;
```

These statements are logically equivalent, but the second statement makes the logic clearer.

To compare a single expression to multiple values, you can simplify the logic by using a single CASE statement instead of an IF with several ELSIF clauses.

Controlling Loop Iterations (LOOP, EXIT, and CONTINUE Statements)

A LOOP statement executes a sequence of statements multiple times. PL/SQL provides the following loop statements:

- Basic LOOP
- WHILE LOOP
- FOR LOOP
- Cursor FOR LOOP

To exit a loop, PL/SQL provides the following statements:

- EXIT
- EXIT-WHEN

To exit the current iteration of a loop, PL/SQL provides the following statements:

- CONTINUE
- CONTINUE-WHEN

You can put EXIT and CONTINUE statements anywhere inside a loop, but not outside a loop. To complete a PL/SQL block before it reaches its normal end, use the RETURN statement (see ["Using the RETURN Statement"](#) on page 8-5).

For the syntax of the LOOP, EXIT, and CONTINUE statements, see [Chapter 13, "PL/SQL Language Elements"](#).

Topics:

- [Using the Basic LOOP Statement](#)
- [Using the EXIT Statement](#)
- [Using the EXIT-WHEN Statement](#)

- [Using the CONTINUE Statement](#)
- [Using the CONTINUE-WHEN Statement](#)
- [Labeling a PL/SQL Loop](#)
- [Using the WHILE-LOOP Statement](#)
- [Using the FOR-LOOP Statement](#)

For information about the cursor FOR-LOOP, see "[Cursor FOR LOOP](#)" on page 6-17.

Using the Basic LOOP Statement

The simplest LOOP statement is the basic loop, which encloses a sequence of statements between the keywords LOOP and END LOOP, as follows:

```
LOOP
    sequence_of_statements
END LOOP;
```

With each iteration of the loop, the sequence of statements is executed, then control resumes at the top of the loop.

You can use CONTINUE and CONTINUE-WHEN statements in a basic loop, but to prevent an infinite loop, you must use an EXIT or EXIT-WHEN statement.

For the syntax of the basic loop, see "[LOOP Statements](#)" on page 13-93.

Using the EXIT Statement

When an EXIT statement is encountered, the loop completes immediately and control passes to the statement immediately after END LOOP, as in [Example 4-8](#).

For the syntax of the EXIT statement, see "[EXIT Statement](#)" on page 13-55.

Example 4-8 Using an 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;
/
Inside loop: x = 0
Inside loop: x = 1
Inside loop: x = 2
Inside loop: x = 3
After loop: x = 4
```

Using the EXIT-WHEN Statement

When an EXIT-WHEN statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the loop completes and control passes to the

statement immediately after `END LOOP`. Until the condition is true, the `EXIT-WHEN` statement acts like a `NULL` statement (except for the evaluation of its condition) and does not terminate the loop. A statement inside the loop must change the value of the condition, as in [Example 4-9](#).

The `EXIT-WHEN` statement replaces a statement of the form `IF ... THEN ... EXIT`. [Example 4-9](#) is logically equivalent to [Example 4-8](#).

For the syntax of the `EXIT-WHEN` statement, see "[EXIT Statement](#)" on page 13-55.

Example 4-9 Using an EXIT-WHEN Statement

```
DECLARE
  x NUMBER := 0;
BEGIN
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Inside loop: x = ' || TO_CHAR(x));
    x := x + 1; -- Change value of condition
    EXIT WHEN x > 3;
  END LOOP;
  -- After EXIT statement, control resumes here
  DBMS_OUTPUT.PUT_LINE ('After loop: x = ' || TO_CHAR(x));
END;
/
```

Using the CONTINUE Statement

When a `CONTINUE` statement is encountered, the current iteration of the loop completes immediately and control passes to the next iteration of the loop, as in [Example 4-10](#).

A `CONTINUE` statement cannot cross a subprogram or method boundary.

For the syntax of the `CONTINUE` statement, see "[CONTINUE Statement](#)" on page 13-33.

Example 4-10 Using a CONTINUE 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;
/
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
```

Note: As of Release 11.1, CONTINUE is a PL/SQL keyword. If your program invokes a subprogram named CONTINUE, you will get a warning.

Using the CONTINUE-WHEN Statement

When a CONTINUE-WHEN statement is encountered, the condition in the WHEN clause is evaluated. If the condition is true, the current iteration of the loop completes and control passes to the next iteration. Until the condition is true, the CONTINUE-WHEN statement acts like a NULL statement (except for the evaluation of its condition) and does not terminate the iteration. However, the value of the condition can vary from iteration to iteration, so that the CONTINUE terminates some iterations and not others.

The CONTINUE-WHEN statement replaces a statement of the form IF ... THEN ... CONTINUE. [Example 4-11](#) is logically equivalent to [Example 4-10](#).

A CONTINUE-WHEN statement cannot cross a subprogram or method boundary.

For the syntax of the CONTINUE-WHEN statement, see "[CONTINUE Statement](#)" on page 13-33.

Example 4-11 Using a CONTINUE-WHEN 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;
/
```

Labeling a PL/SQL Loop

Like PL/SQL blocks, loops can be labeled. The optional label, an undeclared identifier enclosed by double angle brackets, must appear at the beginning of the LOOP statement. The label name can also appear at the end of the LOOP statement. When you nest labeled loops, use ending label names to improve readability.

With either form of EXIT statement, you can exit not only the current loop, but any enclosing loop. Simply label the enclosing loop that you want to exit. Then, use the label in an EXIT statement, as in [Example 4-12](#). Every enclosing loop up to and including the labeled loop is exited.

With either form of CONTINUE statement, you can complete the current iteration of the labelled loop and exit any enclosed loops.

Example 4-12 Using Labeled Loops

```
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;
/

```

Using the WHILE-LOOP Statement

The WHILE-LOOP statement executes the statements in the loop body as long as a condition is true:

```

WHILE condition LOOP
  sequence_of_statements
END LOOP;

```

Before each iteration of the loop, the condition is evaluated. If it is TRUE, the sequence of statements is executed, then control resumes at the top of the loop. If it is FALSE or NULL, the loop is skipped and control passes to the next statement. See [Example 1-10](#) on page 1-11 for an example using the WHILE-LOOP statement.

The number of iterations depends on the condition and is unknown until the loop completes. The condition is tested at the top of the loop, so the sequence might execute zero times.

Some languages have a LOOP UNTIL or REPEAT UNTIL structure, which tests the condition at the bottom of the loop instead of at the top, so that the sequence of statements is executed at least once. The equivalent in PL/SQL is:

```

LOOP
  sequence_of_statements
  EXIT WHEN boolean_expression
END LOOP;

```

To ensure that a WHILE loop executes at least once, use an initialized Boolean variable in the condition, as follows:

```

done := FALSE;
WHILE NOT done LOOP
  sequence_of_statements
  done := boolean_expression
END LOOP;

```

A statement inside the loop must assign a new value to the Boolean variable to avoid an infinite loop.

Using the FOR-LOOP Statement

Simple FOR loops iterate over a specified range of integers. The number of iterations is known before the loop is entered. A double dot (..) serves as the range operator. The

range is evaluated when the FOR loop is first entered and is never re-evaluated. If the lower bound equals the higher bound, the loop body is executed once.

As [Example 4–13](#) shows, the sequence of statements is executed once for each integer in the range 1 to 500. After each iteration, the loop counter is incremented.

Example 4–13 Simple FOR-LOOP Statement

```
DECLARE
  p    NUMBER := 0;
BEGIN
  FOR k IN 1..500 LOOP -- calculate pi with 500 terms
    p := p + ( (-1) ** (k + 1) ) / ((2 * k) - 1 );
  END LOOP;
  p := 4 * p;
  DBMS_OUTPUT.PUT_LINE
    ( 'pi is approximately : ' || p ); -- print result
END;
/
```

By default, iteration proceeds upward from the lower bound to the higher bound. If you use the keyword `REVERSE`, iteration proceeds downward from the higher bound to the lower bound. After each iteration, the loop counter is decremented. You still write the range bounds in ascending (not descending) order.

Example 4–14 Reverse FOR-LOOP Statement

```
BEGIN
  FOR i IN REVERSE 1..3 LOOP -- assign the values 1,2,3 to i
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;
END;
/
```

Inside a FOR loop, the counter can be read but cannot be changed.

```
BEGIN
  FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
    IF i < 3 THEN
      DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
    ELSE
      i := 2; -- not allowed, raises an error
    END IF;
  END LOOP;
END;
/
```

A useful variation of the FOR loop uses a SQL query instead of a range of integers. This technique lets you run a query and process all the rows of the result set with straightforward syntax. For details, see "[Cursor FOR LOOP](#)" on page 6-17.

Topics:

- [How PL/SQL Loops Repeat](#)
- [Dynamic Ranges for Loop Bounds](#)
- [Scope of the Loop Counter Variable](#)
- [Using the EXIT Statement in a FOR Loop](#)

How PL/SQL Loops Repeat

The bounds of a loop range can be literals, variables, or expressions but must evaluate to numbers. Otherwise, PL/SQL raises the predefined exception `VALUE_ERROR`. The lower bound need not be 1, but the loop counter increment or decrement must be 1.

```
j IN -5..5
k IN REVERSE first..last
step IN 0..TRUNC(high/low) * 2
```

Internally, PL/SQL assigns the values of the bounds to temporary `PLS_INTEGER` variables, and, if necessary, rounds the values to the nearest integer. The magnitude range of a `PLS_INTEGER` is -2147483648 to 2147483647, represented in 32 bits. If a bound evaluates to a number outside that range, you get a numeric overflow error when PL/SQL attempts the assignment. See "[PLS_INTEGER and BINARY_INTEGER Datatypes](#)" on page 3-2.

Some languages provide a `STEP` clause, which lets you specify a different increment (5 instead of 1 for example). PL/SQL has no such structure, but you can easily build one. Inside the `FOR` loop, simply multiply each reference to the loop counter by the new increment. In [Example 4-15](#), you assign today's date to elements 5, 10, and 15 of an index-by table:

Example 4-15 Changing the Increment of the Counter in a FOR-LOOP Statement

```
DECLARE
    TYPE DateList IS TABLE OF DATE INDEX BY PLS_INTEGER;
    dates DateList;
    k CONSTANT INTEGER := 5; -- set new increment
BEGIN
    FOR j IN 1..3 LOOP
        dates(j*k) := SYSDATE; -- multiply loop counter by increment
    END LOOP;
END;
/
```

Dynamic Ranges for Loop Bounds

PL/SQL lets you specify the loop range at run time by using variables for bounds as shown in [Example 4-16](#).

Example 4-16 Specifying a LOOP Range at Run Time

```
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 VALUES(i, 'to be added later');
    END LOOP;
    COMMIT;
END;
/
```

If the lower bound of a loop range evaluates to a larger integer than the upper bound, the loop body is not executed and control passes to the next statement:

```
-- limit becomes 1
FOR i IN 2..limit LOOP
    sequence_of_statements -- executes zero times
```

```
END LOOP;
-- control passes here
```

Scope of the Loop Counter Variable

The loop counter is defined only within the loop. You cannot reference that variable name outside the loop. After the loop exits, the loop counter is undefined:

Example 4–17 Scope of the LOOP Counter Variable

```
BEGIN
  FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;
  DBMS_OUTPUT.PUT_LINE (TO_CHAR(i)); -- raises an error
END;
/
```

You do not need to declare the loop counter because it is implicitly declared as a local variable of type INTEGER. It is safest not to use the name of an existing variable, because the local declaration hides any global declaration.

```
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
    DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  END LOOP;
  DBMS_OUTPUT.PUT_LINE (TO_CHAR(i));
  -- refers to original variable value (5)
END;
/
```

To reference the global variable in this example, you must use a label and dot notation, as shown in [Example 4–18](#).

Example 4–18 Using a Label for Referencing Variables Outside a Loop

```
<<main>>
DECLARE
  i NUMBER := 5;
BEGIN
  FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
    DBMS_OUTPUT.PUT_LINE( 'local: ' || TO_CHAR(i)
                          || ' global: ' || TO_CHAR(main.i));
  END LOOP;
END main;
/
```

The same scope rules apply to nested FOR loops. In [Example 4–19](#) both loop counters have the same name. To reference the outer loop counter from the inner loop, you use a label and dot notation.

Example 4–19 Using Labels on Loops for Referencing

```
BEGIN
<<outer_loop>>
  FOR i IN 1..3 LOOP -- assign the values 1,2,3 to i
    <<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;
/

```

Using the EXIT Statement in a FOR Loop

The EXIT statement lets a FOR loop complete early. In [Example 4-20](#), the loop normally executes ten times, but as soon as the FETCH statement fails to return a row, the loop completes no matter how many times it has executed.

Example 4-20 Using EXIT in a LOOP

```

DECLARE
    v_employees employees%ROWTYPE; -- declare record variable
    CURSOR c1 is SELECT * FROM employees;
BEGIN
    OPEN c1; -- open the cursor before fetching
-- An entire row is fetched into the v_employees record
    FOR i IN 1..10 LOOP
        FETCH c1 INTO v_employees;
        EXIT WHEN c1%NOTFOUND;
        -- process data here
    END LOOP;
    CLOSE c1;
END;
/

```

Suppose you must exit early from a nested FOR loop. To complete not only the current loop, but also any enclosing loop, label the enclosing loop and use the label in an EXIT statement as shown in [Example 4-21](#). To complete the current iteration of the labelled loop and exit any enclosed loops, use a label in a CONTINUE statement.

Example 4-21 Using EXIT with a Label in a LOOP

```

DECLARE
    v_employees employees%ROWTYPE; -- declare record variable
    CURSOR c1 is SELECT * FROM employees;
BEGIN
    OPEN c1; -- open the cursor before fetching
-- An entire row is fetched into the 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 WHEN c1%NOTFOUND;
            -- process data here
        END LOOP;
    END LOOP outer_loop;
    CLOSE c1;
END;
/

```

See Also: [Example 6-10](#) on page 6-11

Sequential Control (GOTO and NULL Statements)

Unlike the `IF` and `LOOP` statements, the `GOTO` and `NULL` statements are not crucial to PL/SQL programming. The `GOTO` statement is seldom needed. Occasionally, it can simplify logic enough to warrant its use. The `NULL` statement can improve readability by making the meaning and action of conditional statements clear.

Overuse of `GOTO` statements can result in code that is hard to understand and maintain. Use `GOTO` statements sparingly. For example, to branch from a deeply nested structure to an error-handling routine, raise an exception rather than use a `GOTO` statement. PL/SQL's exception-handling mechanism is explained in [Chapter 11, "Handling PL/SQL Errors"](#).

This section contains the following topics:

- [Using the GOTO Statement](#)
- [GOTO Statement Restrictions](#)
- [Using the NULL Statement](#)

Using the GOTO Statement

The `GOTO` statement branches to a label unconditionally. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. When executed, the `GOTO` statement transfers control to the labeled statement or block. The labeled statement or block can be down or up in the sequence of statements. In [Example 4-22](#) you go to a PL/SQL block up in the sequence of statements.

Example 4-22 Using a Simple GOTO Statement

```
DECLARE
  p          VARCHAR2(30);
  n          PLS_INTEGER := 37; -- test any integer > 2 for prime
BEGIN
  FOR j in 2..ROUND(SQRT(n)) LOOP
    IF n MOD j = 0 THEN -- test for prime
      p := ' is not a prime number'; -- 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;
/
```

The label `end_loop` in the [Example 4-23](#) is not allowed unless it is preceded by an executable statement. To make the label legal, a `NULL` statement is added.

Example 4-23 Using a NULL Statement to Allow a GOTO to a Label

```
DECLARE
  done BOOLEAN;
BEGIN
  FOR i IN 1..50 LOOP
    IF done THEN
      GOTO end_loop;
    END IF;
  <<end_loop>>
  -- not allowed unless an executable statement follows
  NULL;
```

```

    NULL; -- add NULL statement to avoid error
  END LOOP; -- raises an error without the previous NULL
END;
/

```

[Example 4-24](#) shows a GOTO statement can branch to an enclosing block from the current block.

Example 4-24 Using a GOTO Statement to Branch an Enclosing Block

```

-- example with GOTO statement
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; -- branch to enclosing block
    END IF;
  END;
END;
/

```

The GOTO statement branches to the first enclosing block in which the referenced label appears.

GOTO Statement Restrictions

Some possible destinations of a GOTO statement are not allowed. Specifically, a GOTO statement cannot branch into an IF statement, CASE statement, LOOP statement, or sub-block. For example, the following GOTO statement is not allowed:

```

BEGIN
  GOTO update_row; -- cannot branch into IF statement
  IF valid THEN
    <<update_row>>
    UPDATE emp SET ...
  END IF;
END;

```

A GOTO statement cannot branch from one IF statement clause to another, or from one CASE statement WHEN clause to another.

A GOTO statement cannot branch from an outer block into a sub-block (that is, an inner BEGIN-END block).

A GOTO statement cannot branch out of a subprogram. To end a subprogram early, you can use the RETURN statement or use GOTO to branch to a place right before the end of the subprogram.

A GOTO statement cannot branch from an exception handler back into the current BEGIN-END block. However, a GOTO statement can branch from an exception handler into an enclosing block.

Using the NULL Statement

The NULL statement does nothing, and passes control to the next statement. Some languages refer to such an instruction as a no-op (no operation). See "[NULL Statement](#)" on page 13-99.

In [Example 4-25](#), the NULL statement emphasizes that only salespeople receive commissions.

Example 4-25 Using the NULL Statement to Show 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; -- do nothing if not a sales representative
  END IF;
END;
/
```

The NULL statement is a handy way to create placeholders and stub subprograms. In [Example 4-26](#), the NULL statement lets you compile this subprogram, then fill in the real body later. Note that the use of the NULL statement might raise an unreachable code warning if warnings are enabled. See "[Overview of PL/SQL Compile-Time Warnings](#)" on page 11-18.

Example 4-26 Using NULL as a Placeholder When Creating a Subprogram

```
CREATE OR REPLACE PROCEDURE award_bonus
  (emp_id NUMBER, bonus NUMBER) AS
BEGIN -- executable part starts here
  NULL;
  -- use NULL as placeholder,
  -- raises "unreachable code" if warnings enabled
END award_bonus;
/
```

You can use the NULL statement to indicate that you are aware of a possibility, but no action is necessary. In the following exception block, the NULL statement shows that you have chosen not to take any action for unnamed exceptions:

```
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    ROLLBACK;
  WHEN OTHER THEN
    NULL;
END;
```

See [Example 1-13, "Creating a Stored Subprogram"](#) on page 1-14.

Using PL/SQL Collections and Records

Many programming techniques use collection types such as arrays, bags, lists, nested tables, sets, and trees. You can model these types in database applications using the PL/SQL datatypes `TABLE` and `VARRAY`, which enable you to declare nested tables, associative arrays, and variable-size arrays. This chapter shows how to reference and manipulate collections of data as local variables. You also learn how the `RECORD` datatype lets you manipulate related values of different types as a logical unit.

Topics:

- [What Are PL/SQL Collections and Records?](#)
- [Choosing Which PL/SQL Collection Types to Use](#)
- [Defining Collection Types](#)
- [Declaring Collection Variables](#)
- [Initializing and Referencing Collections](#)
- [Referencing Collection Elements](#)
- [Assigning Collections](#)
- [Comparing Collections](#)
- [Using Multilevel Collections](#)
- [Using Collection Methods](#)
- [Avoiding Collection Exceptions](#)
- [Defining and Declaring Records](#)
- [Using Records as Subprogram Parameters and Function Return Values](#)
- [Assigning Values to Records](#)

What Are PL/SQL Collections and Records?

Collections and records are composite types that have internal components that can be manipulated individually, such as the elements of an array, record, or table.

A collection is an ordered group of elements, all of the same type. It is a general concept that encompasses lists, arrays, and other datatypes used in classic programming algorithms. Each element is addressed by a unique subscript.

A record is a group of related data items stored in fields, each with its own name and datatype. You can think of a record as a variable that can hold a table row, or some columns from a table row. The fields correspond to table columns.

Topics:

- [Understanding PL/SQL Collections](#)
- [Understanding PL/SQL Records](#)

Understanding PL/SQL Collections

PL/SQL offers these collection types:

- Associative arrays, also known as index-by tables, let you look up elements using arbitrary numbers and strings for subscript values. These are similar to hash tables in other programming languages.
- Nested tables hold an arbitrary number of elements. They use sequential numbers as subscripts. You can define equivalent SQL types, allowing nested tables to be stored in database tables and manipulated through SQL.
- Varrays (short for variable-size arrays) hold a fixed number of elements (although you can change the number of elements at run time). They use sequential numbers as subscripts. You can define equivalent SQL types, allowing varrays to be stored in database tables. They can be stored and retrieved through SQL, but with less flexibility than nested tables.

Although collections have only one dimension, you can model multi-dimensional arrays by creating collections whose elements are also collections.

To use collections in an application, you define one or more PL/SQL types, then define variables of those types. You can define collection types in a subprogram or package. You can pass collection variables as parameters to stored subprograms.

To look up data that is more complex than single values, you can store PL/SQL records or SQL object types in collections. Nested tables and varrays can also be attributes of object types.

Topics:

- [Understanding PL/SQL Nested Tables](#)
- [Understanding PL/SQL Varrays](#)
- [Understanding PL/SQL Associative Arrays \(Index-By Tables\)](#)
- [How Globalization Settings Affect VARCHAR2 Keys for Associative Arrays](#)

Understanding PL/SQL Nested Tables

PL/SQL nested tables represent sets of values. You can think of them as one-dimensional arrays with no declared number of elements. You can model multi-dimensional arrays by creating nested tables whose elements are also nested tables.

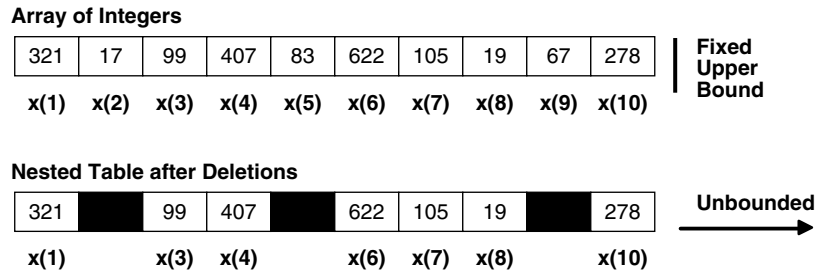
Within the database, nested tables are column types that hold sets of values. Oracle stores the rows of a nested table in no particular order. When you retrieve a nested table from the database into a PL/SQL variable, the rows are given consecutive subscripts starting at 1. That gives you array-like access to individual rows.

Nested tables differ from arrays in two important ways:

1. Nested tables do not have a declared number of elements, while arrays have a predefined number as illustrated in [Figure 5–1](#). The size of a nested table can increase dynamically; however, a maximum limit is imposed. See "[Referencing Collection Elements](#)" on page 5-12.

- Nested tables might not have consecutive subscripts, while arrays are always dense (have consecutive subscripts). Initially, nested tables are dense, but they can become sparse (have nonconsecutive subscripts). You can delete elements from a nested table using the built-in procedure `DELETE`. The built-in function `NEXT` lets you iterate over all the subscripts of a nested table, even if the sequence has gaps.

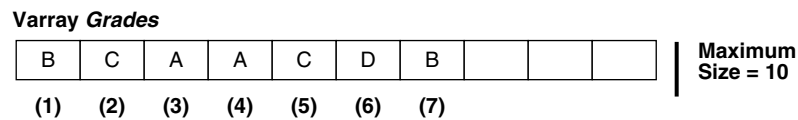
Figure 5–1 Array and Nested Table



Understanding PL/SQL Varrays

Items of type `VARRAY` are called varrays. They let you reference individual elements for array operations, or manipulate the collection as a whole. To reference an element, you use standard subscripting syntax (see [Figure 5–2](#)). For example, `Grade(3)` references the third element in varray `Grades`.

Figure 5–2 Varray of Size 10



A varray has a maximum size, which you specify in its type definition. Its index has a fixed lower bound of 1 and an extensible upper bound. For example, the current upper bound for varray `Grades` is 7, but you can increase its upper bound to maximum of 10. A varray can contain a varying number of elements, from zero (when empty) to the maximum specified in its type definition.

Understanding PL/SQL Associative Arrays (Index-By Tables)

Associative arrays are sets of key-value pairs, where each key is unique and is used to locate a corresponding value in the array. The key can be an integer or a string.

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

For example, here is the declaration of an associative array type, and two arrays of that type, using keys that are strings:

Example 5–1 Declaring Collection Types

```
DECLARE TYPE population_type
  IS TABLE OF NUMBER INDEX BY VARCHAR2(64);
  country_population population_type;
  continent_population population_type;
```

```

    howmany NUMBER;
    which VARCHAR2(64);
BEGIN
    country_population('Greenland') := 100000; -- Creates new entry
    country_population('Iceland') := 750000;   -- Creates new entry
-- Looks up value associated with a string
    howmany := country_population('Greenland');
    continent_population('Australia') := 30000000;
    continent_population('Antarctica') := 1000; -- Creates new entry
    continent_population('Antarctica') := 1001;
        -- Replaces previous value
-- Returns 'Antarctica' as that comes first alphabetically.
    which := continent_population.FIRST;
-- Returns 'Australia' as that comes last alphabetically.
    which := continent_population.LAST;
-- Returns the value corresponding to the last key, in this
-- case the population of Australia.
    howmany := continent_population(continent_population.LAST);
END;
/

```

Associative arrays help you represent data sets of arbitrary size, with fast lookup for an individual element without knowing its position within the array and without having to loop through all the array elements. It is like a simple version of a SQL table where you can retrieve values based on the primary key. For simple temporary storage of lookup data, associative arrays let you avoid using the disk space and network operations required for SQL tables.

Because associative arrays are intended for temporary data rather than storing persistent data, you cannot use them with SQL statements such as `INSERT` and `SELECT INTO`. You can make them persistent for the life of a database session by declaring the type in a package and assigning the values in a package body.

How Globalization Settings Affect VARCHAR2 Keys for Associative Arrays

If settings for national language or globalization change during a session that uses associative arrays with `VARCHAR2` key values, the program might encounter a run-time error. For example, changing the `NLS_COMP` or `NLS_SORT` initialization parameters within a session might cause methods such as `NEXT` and `PRIOR` to raise exceptions. If you need to change these settings during the session, make sure to set them back to their original values before performing further operations with these kinds of associative arrays.

When you declare an associative array using a string as the key, the declaration must use a `VARCHAR2`, `STRING`, or `LONG` type. You can use a different type, such as `NCHAR` or `NVARCHAR2`, as the key value to reference an associative array. You can even use a type such as `DATE`, as long as it can be converted to `VARCHAR2` by the `TO_CHAR` function. Note that the `LONG` datatype is supported only for backward compatibility; see ["LONG and LONG RAW Datatypes"](#) on page 3-13 for more information.

However, you must be careful when using other types that the values used as keys are consistent and unique. For example, the string value of `SYSDATE` might change if the `NLS_DATE_FORMAT` initialization parameter changes, so that `array_element(SYSDATE)` does not produce the same result as before. Two different `NVARCHAR2` values might turn into the same `VARCHAR2` value (containing question marks instead of certain national characters). In that case, `array_element(national_string1)` and `array_element(national_string2)` might refer to the same element. Two different `CHAR` or `VARCHAR2` values that differ in terms of case, accented characters, or punctuation characters might also be considered

the same if the value of the `NLS_SORT` initialization parameter ends in `_CI` (case-insensitive comparisons) or `_AI` (accent- and case-insensitive comparisons).

When you pass an associative array as a parameter to a remote database using a database link, the two databases can have different globalization settings. When the remote database performs operations such as `FIRST` and `NEXT`, it uses its own character order even if that is different from the order where the collection originated. If character set differences mean that two keys that were unique are not unique on the remote database, the program receives a `VALUE_ERROR` exception.

Understanding PL/SQL Records

Records are composed of a group of fields, similar to the columns in a row. The `%ROWTYPE` attribute lets you declare a PL/SQL record that represents a row in a database table, without listing all the columns. Your code keeps working even after columns are added to the table. If you want to represent a subset of columns in a table, or columns from different tables, you can define a view or declare a cursor to select the right columns and do any necessary joins, and then apply `%ROWTYPE` to the view or cursor.

For information on using records in PL/SQL, see the following sections in this chapter:

- [Defining and Declaring Records](#)
- [Assigning Values to Records](#)

Choosing Which PL/SQL Collection Types to Use

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

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

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

Topics:

- [Choosing Between Nested Tables and Associative Arrays](#)
- [Choosing Between Nested Tables and Varrays](#)

Choosing Between Nested Tables and Associative Arrays

Both nested tables and associative arrays (formerly known as index-by tables) use similar subscript notation, but they have different characteristics when it comes to persistence and ease of parameter passing.

Nested tables can be stored in a database column, but associative arrays cannot. Nested tables can simplify SQL operations in which you typically join a single-column table with a larger table.

Associative arrays are appropriate for relatively small lookup tables where the collection can be constructed in memory each time a subprogram is invoked or a

package is initialized. They are good for collecting information whose volume is unknown beforehand, because there is no fixed limit on their size. Their index values are more flexible, because associative array subscripts can be negative, can be nonsequential, and can use string values instead of numbers.

PL/SQL automatically converts between host arrays and associative arrays that use numeric key values. The most efficient way to pass collections to and from the database server is to set up data values in associative arrays, then use those associative arrays with bulk constructs (the `FORALL` statement or `BULK COLLECT` clause).

Choosing Between Nested Tables and Varrays

Varrays are a good choice when:

- The number of elements is known in advance.
- The elements are usually all accessed in sequence.

When stored in the database, varrays keep their ordering and subscripts.

Each varray is stored as a single object, either inside the table of which it is a column (if the varray is less than 4KB) or outside the table but still in the same tablespace (if the varray is greater than 4KB). You must update or retrieve all elements of the varray at the same time, which is most appropriate when performing some operation on all the elements at once. But you might find it impractical to store and retrieve large numbers of elements this way.

Nested tables are a good choice when:

- Index values are not consecutive.
- There is no set number of index values. However, a maximum limit is imposed. See ["Referencing Collection Elements"](#) on page 5-12.
- You must delete or update some elements, but not all the elements at once.
- You would typically create a separate lookup table, with multiple entries for each row of the main table, and access it through join queries.

Nested tables can be sparse: you can delete arbitrary elements, rather than just removing an item from the end.

Nested table data is stored in a separate store table, a system-generated database table associated with the nested table. The database joins the tables for you when you access the nested table. This makes nested tables suitable for queries and updates that only affect some elements of the collection.

You cannot rely on the order and subscripts of a nested table remaining stable as the nested table is stored in and retrieved from the database, because the order and subscripts are not preserved in the database.

Defining Collection Types

To create collections, you define a collection type, then declare variables of that type. Collections follow the same scoping and instantiation rules as other types and variables. Collections are instantiated when you enter a block or subprogram, and cease to exist when you exit. In a package, collections are instantiated when you first reference the package and cease to exist when you end the database session.

Note: For information on the `CREATE TYPE SQL` statement, see *Oracle Database SQL Language Reference*. For information on the `CREATE TYPE BODY SQL` statement, see *Oracle Database SQL Language Reference*.

You can define `TABLE` and `VARRAY` types in the declarative part of any PL/SQL block, subprogram, or package using a `TYPE` definition. For the syntax information, see "[Collection Definition](#)" on page 13-19.

For nested tables and varrays declared within PL/SQL, the element type of the table or varray can be any PL/SQL datatype except `REF CURSOR`.

When defining a `VARRAY` type, you must specify its maximum size with a positive integer. In the following example, you define a type that stores up to 366 dates:

```
DECLARE
    TYPE Calendar IS VARRAY(366) OF DATE;
```

Associative arrays (also known as index-by tables) let you insert elements using arbitrary key values. The keys do not have to be consecutive.

The key datatype can be `PLS_INTEGER`, `VARCHAR2`, or one of `VARCHAR2` subtypes `VARCHAR`, `STRING`, or `LONG`.

You must specify the length of a `VARCHAR2`-based key, except for `LONG` which is equivalent to declaring a key type of `VARCHAR2(32760)`. The types `RAW`, `LONG RAW`, `ROWID`, `CHAR`, and `CHARACTER` are not allowed as keys for an associative array. Note that the `LONG` and `LONG RAW` datatypes are supported only for backward compatibility; see "[LONG and LONG RAW Datatypes](#)" on page 3-13 for more information.

An initialization clause is not allowed. There is no constructor notation for associative arrays. When you reference an element of an associative array that uses a `VARCHAR2`-based key, you can use other types, such as `DATE` or `TIMESTAMP`, as long as they can be converted to `VARCHAR2` with the `TO_CHAR` function.

Associative arrays can store data using a primary key value as the index, where the key values are not sequential. [Example 5-2](#) creates a single element in an associative array, with a subscript of 100 rather than 1.

Example 5-2 Declaring an Associative Array

```
DECLARE
    TYPE EmpTabTyp IS TABLE OF employees%ROWTYPE
        INDEX BY PLS_INTEGER;
    emp_tab EmpTabTyp;
BEGIN
    /* Retrieve employee record. */
    SELECT * INTO emp_tab(100) FROM employees
        WHERE employee_id = 100;
END;
```

Declaring Collection Variables

After defining a collection type, you declare variables of that type. You use the new type name in the declaration, the same as with predefined types such as `NUMBER`.

Example 5-3 Declaring Nested Tables, Varrays, and Associative Arrays

```

DECLARE
    TYPE nested_type IS TABLE OF VARCHAR2(30);
    TYPE varray_type IS VARRAY(5) OF INTEGER;
    TYPE assoc_array_num_type
        IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    TYPE assoc_array_str_type
        IS TABLE OF VARCHAR2(32) INDEX BY PLS_INTEGER;
    TYPE assoc_array_str_type2
        IS TABLE OF VARCHAR2(32) INDEX BY VARCHAR2(64);
    v1 nested_type;
    v2 varray_type;
    v3 assoc_array_num_type;
    v4 assoc_array_str_type;
    v5 assoc_array_str_type2;
BEGIN
    -- an arbitrary number of strings can be inserted v1
    v1 := nested_type('Shipping', 'Sales', 'Finance', 'Payroll');
    v2 := varray_type(1, 2, 3, 4, 5); -- Up to 5 integers
    v3(99) := 10; -- Just start assigning to elements
    v3(7) := 100; -- Subscripts can be any integer values
    v4(42) := 'Smith'; -- Just start assigning to elements
    v4(54) := 'Jones'; -- Subscripts can be any integer values
    v5('Canada') := 'North America';
        -- Just start assigning to elements
    v5('Greece') := 'Europe';
        -- Subscripts can be string values
END;
/

```

As shown in [Example 5-4](#), you can use %TYPE to specify the datatype of a previously declared collection, so that changing the definition of the collection automatically updates other variables that depend on the number of elements or the element type.

Example 5-4 Declaring Collections with %TYPE

```

DECLARE
    TYPE few_depts IS VARRAY(10) OF VARCHAR2(30);
    TYPE many_depts IS VARRAY(100) OF VARCHAR2(64);
    some_depts few_depts;
    -- If we change the type of some_depts from few_depts to many_depts,
    -- local_depts and global_depts will use the same type
    -- when this block is recompiled
    local_depts some_depts%TYPE;
    global_depts some_depts%TYPE;
BEGIN
    NULL;
END;
/

```

You can declare collections as the formal parameters of subprograms. That way, you can pass collections to stored subprograms and from one subprogram to another.

[Example 5-5](#) declares a nested table as a parameter of a packaged subprogram.

Example 5-5 Declaring a Procedure Parameter as a Nested Table

```

CREATE PACKAGE personnel AS
    TYPE staff_list IS TABLE OF employees.employee_id%TYPE;
    PROCEDURE award_bonuses (empleos_buenos IN staff_list);
END personnel;

```

```

/
CREATE PACKAGE BODY personnel AS
  PROCEDURE award_bonuses (empleos_buenos staff_list) IS
  BEGIN
    FOR i IN empleos_buenos.FIRST..empleos_buenos.LAST
    LOOP
      UPDATE employees SET salary = salary + 100
        WHERE employees.employee_id = empleos_buenos(i);
    END LOOP;
  END;
END;
/

```

To invoke `personnel.award_bonuses` from outside the package, you declare a variable of type `personnel.staff_list` and pass that variable as the parameter.

Example 5-6 Invoking a Procedure with a Nested Table Parameter

```

DECLARE
  good_employees personnel.staff_list;
BEGIN
  good_employees := personnel.staff_list(100, 103, 107);
  personnel.award_bonuses (good_employees);
END;
/

```

You can also specify a collection type in the RETURN clause of a function specification.

To specify the element type, you can use `%TYPE`, which provides the datatype of a variable or database column. Also, you can use `%ROWTYPE`, which provides the rowtype of a cursor or database table. See [Example 5-7](#) and [Example 5-8](#).

Example 5-7 Specifying Collection Element Types with %TYPE and %ROWTYPE

```

DECLARE
-- Nested table type that can hold an arbitrary number
--   of employee IDs.
-- The element type is based on a column from the EMPLOYEES table.
-- We do not need to know whether the ID is a number or a string.
  TYPE EmpList IS TABLE OF employees.employee_id%TYPE;
-- Declare a cursor to select a subset of columns.
  CURSOR c1 IS SELECT employee_id FROM employees;
-- Declare an Array type that can hold information
--   about 10 employees.
-- The element type is a record that contains all the same
-- fields as the EMPLOYEES table.
  TYPE Senior_Salespeople IS VARRAY(10) OF employees%ROWTYPE;
-- Declare a cursor to select a subset of columns.
  CURSOR c2 IS SELECT first_name, last_name FROM employees;
-- Array type that can hold a list of names. The element type
-- is a record that contains the same fields as the cursor
-- (that is, first_name and last_name).
  TYPE NameList IS VARRAY(20) OF c2%ROWTYPE;
BEGIN
  NULL;
END;
/

```

[Example 5-8](#) uses a RECORD type to specify the element type. See "Defining and Declaring Records" on page 5-31.

Example 5-8 VARRAY of Records

```
DECLARE TYPE name_rec
  IS RECORD ( first_name VARCHAR2(20), last_name VARCHAR2(25));
  TYPE names IS VARRAY(250) OF name_rec;
BEGIN
  NULL;
END;
/
```

You can also impose a NOT NULL constraint on the element type, as shown in [Example 5-9](#).

Example 5-9 NOT NULL Constraint on Collection Elements

```
DECLARE TYPE EmpList
  IS TABLE OF employees.employee_id%TYPE NOT NULL;
  v_employees EmpList := EmpList(100, 150, 160, 200);
BEGIN
  v_employees(3) := NULL; -- assigning NULL raises an error
END;
/
```

Initializing and Referencing Collections

Until you initialize it, a nested table or varray is atomically null; the collection itself is null, not its elements. To initialize a nested table or varray, you use a constructor, a system-defined function with the same name as the collection type. This function constructs collections from the elements passed to it.

You must explicitly call a constructor for each varray and nested table variable. Associative arrays, the third kind of collection, do not use constructors. Constructor calls are allowed wherever function calls are allowed.

[Example 5-10](#) initializes a nested table using a constructor, which looks like a function with the same name as the collection type:

Example 5-10 Constructor for a Nested Table

```
DECLARE
  TYPE dnames_tab IS TABLE OF VARCHAR2(30);
  dept_names dnames_tab;
BEGIN
  dept_names := dnames_tab('Shipping','Sales','Finance','Payroll');
END;
/
```

Because a nested table does not have a declared size, you can put as many elements in the constructor as necessary.

[Example 5-11](#) initializes a varray using a constructor, which looks like a function with the same name as the collection type:

Example 5-11 Constructor for a Varray

```
DECLARE
  -- In the varray, we put an upper limit on the number of elements
  TYPE dnames_var IS VARRAY(20) OF VARCHAR2(30);
```



```

        dept_names dnames_var;
BEGIN
-- Because dnames is declared as VARRAY(20),
-- we can put up to 10elements in the constructor
    dept_names := dnames_var('Shipping','Sales','Finance','Payroll');
END;
/

```

Unless you impose the NOT NULL constraint in the type declaration, you can pass null elements to a constructor as in [Example 5–12](#).

Example 5–12 Collection Constructor Including Null Elements

```

DECLARE
    TYPE dnames_tab IS TABLE OF VARCHAR2(30);
    dept_names dnames_tab;
    TYPE dnamesNoNulls_type IS TABLE OF VARCHAR2(30) NOT NULL;
BEGIN
    dept_names := dnames_tab('Shipping', NULL,'Finance', NULL);
-- If dept_names were of type dnamesNoNulls_type,
-- we could not include null values in the constructor
END;
/

```

You can initialize a collection in its declaration, which is a good programming practice, as shown in [Example 5–13](#). In this case, you can invoke the collection's EXTEND method to add elements later.

Example 5–13 Combining Collection Declaration and Constructor

```

DECLARE
    TYPE dnames_tab IS TABLE OF VARCHAR2(30);
    dept_names dnames_tab :=
        dnames_tab('Shipping','Sales','Finance','Payroll');
BEGIN
    NULL;
END;
/

```

If you call a constructor without arguments, you get an empty but non-null collection as shown in [Example 5–14](#).

Example 5–14 Empty Varray Constructor

```

DECLARE
    TYPE dnames_var IS VARRAY(20) OF VARCHAR2(30);
    dept_names dnames_var;
BEGIN
    IF dept_names IS NULL THEN
        DBMS_OUTPUT.PUT_LINE
            ('Before initialization, the varray is null.');
```

-- While the varray is null, we cannot check its COUNT attribute.

```

    -- DBMS_OUTPUT.PUT_LINE
    -- ('It has ' || dept_names.COUNT || ' elements.');
```

```

    ELSE
        DBMS_OUTPUT.PUT_LINE
            ('Before initialization, the varray is not null.');
```

```

    END IF;
    dept_names := dnames_var(); -- initialize empty varray
    IF dept_names IS NULL THEN

```

```

        DBMS_OUTPUT.PUT_LINE
            ('After initialization, the varray is null.');
```

ELSE

```

        DBMS_OUTPUT.PUT_LINE
            ('After initialization, the varray is not null.');
```

DBMS_OUTPUT.PUT_LINE

```

            ('It has ' || dept_names.COUNT || ' elements.');
```

END IF;

```

END;
/
```

Referencing Collection Elements

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

```
collection_name (subscript)
```

where *subscript* is an expression that yields an integer in most cases, or a VARCHAR2 for associative arrays declared with strings as keys.

The allowed subscript ranges are:

- For nested tables, 1..2147483647 (the upper limit of PLS_INTEGER).
- For varrays, 1.. *size_limit*, where you specify the limit in the declaration (*size_limit* cannot exceed 2147483647).
- For associative arrays with a numeric key, -2147483648..2147483647.
- For associative arrays with a string key, the length of the key and number of possible values depends on the VARCHAR2 length limit in the type declaration, and the database character set.

[Example 5–15](#) shows how to reference an element in a nested table.

Example 5–15 Referencing a Nested Table Element

```

DECLARE
    TYPE Roster IS TABLE OF VARCHAR2(15);
    names Roster :=
        Roster('D Caruso', 'J Hamil', 'D Piro', 'R Singh');
    PROCEDURE verify_name(the_name VARCHAR2) IS
    BEGIN
        DBMS_OUTPUT.PUT_LINE(the_name);
    END;
BEGIN
    FOR i IN names.FIRST .. names.LAST
    LOOP
        IF names(i) = 'J Hamil' THEN
            DBMS_OUTPUT.PUT_LINE(names(i));
            -- reference to nested table element
        END IF;
    END LOOP;
    verify_name(names(3));
    -- procedure call with reference to element
END;
/
```

[Example 5–16](#) shows how you can reference the elements of an associative array in a function call.

Example 5–16 Referencing an Element of an Associative Array

```

DECLARE
  TYPE sum_multiples IS TABLE OF PLS_INTEGER INDEX BY PLS_INTEGER;
  n PLS_INTEGER := 5; -- number of multiples to sum for display
  sn PLS_INTEGER := 10; -- number of multiples to sum
  m PLS_INTEGER := 3; -- multiple
FUNCTION get_sum_multiples
  (multiple IN PLS_INTEGER, num IN PLS_INTEGER)
RETURN sum_multiples IS
  s sum_multiples;
BEGIN
  FOR i IN 1..num LOOP
    s(i) := multiple * ((i * (i + 1)) / 2);
    -- sum of multiples
  END LOOP;
  RETURN s;
END get_sum_multiples;
BEGIN
-- invok function to retrieve
-- element identified by subscript (key)
DBMS_OUTPUT.PUT_LINE
  ('Sum of the first ' || TO_CHAR(n) || ' multiples of ' ||
   TO_CHAR(m) || ' is ' || TO_CHAR(get_sum_multiples (m, sn)(n)));
END;
/

```

Assigning Collections

One collection can be assigned to another by an INSERT, UPDATE, FETCH, or SELECT statement, an assignment statement, or a subprogram call. You can assign the value of an expression to a specific element in a collection using the syntax:

```
collection_name (subscript) := expression;
```

where *expression* yields a value of the type specified for elements in the collection type definition.

You can use operators such as SET, MULTISSET UNION, MULTISSET INTERSECT, and MULTISSET EXCEPT to transform nested tables as part of an assignment statement.

Assigning a value to a collection element can cause exceptions, such as:

- If the subscript is NULL or is not convertible to the right datatype, PL/SQL raises the predefined exception VALUE_ERROR. Usually, the subscript must be an integer. Associative arrays can also be declared to have VARCHAR2 subscripts.
- If the subscript refers to an uninitialized element, PL/SQL raises SUBSCRIPT_BEYOND_COUNT.
- If the collection is atomically null, PL/SQL raises COLLECTION_IS_NULL.

For more information on collection exceptions, see ["Avoiding Collection Exceptions"](#) on page 5-28, [Example 5–38](#) on page 5-28, and ["Summary of Predefined PL/SQL Exceptions"](#) on page 11-4.

[Example 5–17](#) shows that collections must have the same datatype for an assignment to work. Having the same element type is not enough.

Example 5–17 Datatype Compatibility for Collection Assignment

```
DECLARE
```

```

TYPE last_name_typ IS VARRAY(3) OF VARCHAR2(64);
TYPE surname_typ IS VARRAY(3) OF VARCHAR2(64);
-- These first two variables have the same datatype.
group1 last_name_typ := last_name_typ('Jones', 'Wong', 'Marceau');
group2 last_name_typ := last_name_typ('Klein', 'Patsos', 'Singh');
-- This third variable has a similar declaration,
-- but is not the same type.
group3 surname_typ := surname_typ('Trevisi', 'MacLeod', 'Marquez');
BEGIN
-- Allowed because they have the same datatype
group1 := group2;
-- Not allowed because they have different datatypes
-- group3 := group2; -- raises an error
END;
/

```

If you assign an atomically null nested table or varray to a second nested table or varray, the second collection must be reinitialized, as shown in [Example 5–18](#). In the same way, assigning the value NULL to a collection makes it atomically null.

Example 5–18 Assigning a Null Value to a Nested Table

```

DECLARE
TYPE dnames_tab IS TABLE OF VARCHAR2(30);
-- This nested table has some values
dept_names dnames_tab :=
    dnames_tab('Shipping', 'Sales', 'Finance', 'Payroll');
-- This nested table is not initialized ("atomically null").
empty_set dnames_tab;
BEGIN
-- At first, the initialized variable is not null.
if dept_names IS NOT NULL THEN
    DBMS_OUTPUT.PUT_LINE('OK, at first dept_names is not null.');
```

END IF;

```

-- Then we assign a null nested table to it.
dept_names := empty_set;
-- Now it is null.
if dept_names IS NULL THEN
    DBMS_OUTPUT.PUT_LINE('OK, now dept_names has become null.');
```

END IF;

```

-- We must use another constructor to give it some values.
dept_names := dnames_tab('Shipping', 'Sales', 'Finance', 'Payroll');
```

END;

/

[Example 5–19](#) shows some of the ANSI-standard operators that you can apply to nested tables.

Example 5–19 Assigning Nested Tables with Set Operators

```

DECLARE
TYPE nested_typ IS TABLE OF NUMBER;
nt1 nested_typ := nested_typ(1,2,3);
nt2 nested_typ := nested_typ(3,2,1);
nt3 nested_typ := nested_typ(2,3,1,3);
nt4 nested_typ := nested_typ(1,2,4);
answer nested_typ;
-- The results might be in a different order than you expect.
-- Do not rely on the order of elements in nested tables.
PROCEDURE print_nested_table(the_nt nested_typ) IS

```

```

        output VARCHAR2(128);
BEGIN
    IF the_nt IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('Results: <NULL>');
        RETURN;
    END IF;
    IF the_nt.COUNT = 0 THEN
        DBMS_OUTPUT.PUT_LINE('Results: empty set');
        RETURN;
    END IF;
    FOR i IN the_nt.FIRST .. the_nt.LAST
    LOOP
        output := output || the_nt(i) || ' ';
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Results: ' || output);
END;
BEGIN
    answer := nt1 MULTISSET UNION nt4; -- (1,2,3,1,2,4)
    print_nested_table(answer);
    answer := nt1 MULTISSET UNION nt3; -- (1,2,3,2,3,1,3)
    print_nested_table(answer);
    answer := nt1 MULTISSET UNION DISTINCT nt3; -- (1,2,3)
    print_nested_table(answer);
    answer := nt2 MULTISSET INTERSECT nt3; -- (3,2,1)
    print_nested_table(answer);
    answer := nt2 MULTISSET INTERSECT DISTINCT nt3; -- (3,2,1)
    print_nested_table(answer);
    answer := SET(nt3); -- (2,3,1)
    print_nested_table(answer);
    answer := nt3 MULTISSET EXCEPT nt2; -- (3)
    print_nested_table(answer);
    answer := nt3 MULTISSET EXCEPT DISTINCT nt2; -- ()
    print_nested_table(answer);
END;
/

```

[Example 5–20](#) shows an assignment to a VARRAY of records with an assignment statement.

Example 5–20 Assigning Values to VARRAYs with Complex Datatypes

```

DECLARE
    TYPE emp_name_rec is RECORD (
        firstname    employees.first_name%TYPE,
        lastname     employees.last_name%TYPE,
        hiredate     employees.hire_date%TYPE
    );

    -- Array type that can hold information 10 employees
    TYPE EmpList_arr IS VARRAY(10) OF emp_name_rec;
    SeniorSalespeople EmpList_arr;

    -- Declare a cursor to select a subset of columns.
    CURSOR c1 IS SELECT first_name, last_name, hire_date
        FROM employees;
    Type NameSet IS TABLE OF c1%ROWTYPE;
    SeniorTen NameSet;
    EndCounter NUMBER := 10;

BEGIN
    SeniorSalespeople := EmpList_arr();

```

```

SELECT first_name, last_name, hire_date
  BULK COLLECT INTO SeniorTen
  FROM employees
  WHERE job_id = 'SA_REP'
  ORDER BY hire_date;
IF SeniorTen.LAST > 0 THEN
  IF SeniorTen.LAST < 10 THEN EndCounter := SeniorTen.LAST;
  END IF;
  FOR i in 1..EndCounter LOOP
    SeniorSalespeople.EXTEND(1);
    SeniorSalespeople(i) := SeniorTen(i);
    DBMS_OUTPUT.PUT_LINE(SeniorSalespeople(i).lastname || ', '
      || SeniorSalespeople(i).firstname || ', ' ||
      SeniorSalespeople(i).hiredate);
  END LOOP;
END IF;
END;
/

```

Example 5-21 shows an assignment to a nested table of records with a `FETCH` statement.

Example 5-21 Assigning Values to Tables with Complex Datatypes

```

DECLARE
  TYPE emp_name_rec IS RECORD (
    firstname   employees.first_name%TYPE,
    lastname    employees.last_name%TYPE,
    hiredate    employees.hire_date%TYPE
  );

  -- Table type that can hold information about employees
  TYPE EmpList_tab IS TABLE OF emp_name_rec;
  SeniorSalespeople EmpList_tab;

  -- Declare a cursor to select a subset of columns.
  CURSOR c1 IS SELECT first_name, last_name, hire_date
    FROM employees;
  EndCounter NUMBER := 10;
  TYPE EmpCurTyp IS REF CURSOR;
  emp_cv EmpCurTyp;

BEGIN
  OPEN emp_cv FOR SELECT first_name, last_name, hire_date
    FROM employees
    WHERE job_id = 'SA_REP' ORDER BY hire_date;

  FETCH emp_cv BULK COLLECT INTO SeniorSalespeople;
  CLOSE emp_cv;

  -- for this example, display a maximum of ten employees
  IF SeniorSalespeople.LAST > 0 THEN
    IF SeniorSalespeople.LAST < 10 THEN
      EndCounter := SeniorSalespeople.LAST;
    END IF;
    FOR i in 1..EndCounter LOOP
      DBMS_OUTPUT.PUT_LINE
        (SeniorSalespeople(i).lastname || ', '
        || SeniorSalespeople(i).firstname || ', ' ||
        SeniorSalespeople(i).hiredate);
    END LOOP;
  END IF;
END;

```

```

        END LOOP;
    END IF;
END;
/

```

Comparing Collections

You can check whether a collection is null. Comparisons such as greater than, less than, and so on are not allowed. This restriction also applies to implicit comparisons. For example, collections cannot appear in a `DISTINCT`, `GROUP BY`, or `ORDER BY` list.

If you want to do such comparison operations, you must define your own notion of what it means for collections to be greater than, less than, and so on, and write one or more functions to examine the collections and their elements and return a true or false value.

For nested tables, you can check whether two nested table of the same declared type are equal or not equal, as shown in [Example 5-23](#). You can also apply set operators (`CARDINALITY`, `MEMBER OF`, `IS A SET`, `IS EMPTY`) to check certain conditions within a nested table or between two nested tables, as shown in [Example 5-24](#).

Because nested tables and varrays can be atomically null, they can be tested for nullity, as shown in [Example 5-22](#).

Example 5-22 Checking if a Collection Is Null

```

DECLARE
    TYPE emp_name_rec IS RECORD (
        firstname    employees.first_name%TYPE,
        lastname     employees.last_name%TYPE,
        hiredate     employees.hire_date%TYPE
    );
    TYPE staff IS TABLE OF emp_name_rec;
    members staff;
BEGIN
    -- Condition yields TRUE because we have not used a constructor.
    IF members IS NULL THEN
        DBMS_OUTPUT.PUT_LINE('NULL');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Not NULL');
    END IF;
END;
/

```

[Example 5-23](#) shows that nested tables can be compared for equality or inequality. They cannot be ordered, because there is no greater than or less than comparison.

Example 5-23 Comparing Two Nested Tables

```

DECLARE
    TYPE dnames_tab IS TABLE OF VARCHAR2(30);
    dept_names1 dnames_tab :=
        dnames_tab('Shipping', 'Sales', 'Finance', 'Payroll');
    dept_names2 dnames_tab :=
        dnames_tab('Sales', 'Finance', 'Shipping', 'Payroll');
    dept_names3 dnames_tab :=
        dnames_tab('Sales', 'Finance', 'Payroll');
BEGIN
    -- We can use = or !=, but not < or >.
    -- These 2 are equal even though members are in different order.

```

```

IF dept_names1 = dept_names2 THEN
    DBMS_OUTPUT.PUT_LINE
        ('dept_names1 and dept_names2 have the same members. ');
END IF;
IF dept_names2 != dept_names3 THEN
    DBMS_OUTPUT.PUT_LINE
        ('dept_names2 and dept_names3 have different members. ');
END IF;
END;
/

```

You can test certain properties of a nested table, or compare two nested tables, using ANSI-standard set operations, as shown in [Example 5-24](#).

Example 5-24 Comparing Nested Tables with Set Operators

```

DECLARE
    TYPE nested_typ IS TABLE OF NUMBER;
    nt1 nested_typ := nested_typ(1,2,3);
    nt2 nested_typ := nested_typ(3,2,1);
    nt3 nested_typ := nested_typ(2,3,1,3);
    nt4 nested_typ := nested_typ(1,2,4);
    answer BOOLEAN;
    howmany NUMBER;
    PROCEDURE testify
        (truth BOOLEAN DEFAULT NULL
         quantity NUMBER DEFAULT NULL) IS
    BEGIN
        IF truth IS NOT NULL THEN
            DBMS_OUTPUT.PUT_LINE
                (CASE truth WHEN TRUE THEN 'True' WHEN FALSE THEN 'False' END);
        END IF;
        IF quantity IS NOT NULL THEN
            DBMS_OUTPUT.PUT_LINE(quantity);
        END IF;
    END;
BEGIN
    answer := nt1 IN (nt2,nt3,nt4); -- true, nt1 matches nt2
    testify(truth => answer);
    answer := nt1 SUBMULTISET OF nt3; -- true, all elements match
    testify(truth => answer);
    answer := nt1 NOT SUBMULTISET OF nt4; -- also true
    testify(truth => answer);
    howmany := CARDINALITY(nt3); -- number of elements in nt3
    testify(quantity => howmany);
    howmany := CARDINALITY(SET(nt3)); -- number of distinct elements
    testify(quantity => howmany);
    answer := 4 MEMBER OF nt1; -- false, no element matches
    testify(truth => answer);
    answer := nt3 IS A SET; -- false, nt3 has duplicates
    testify(truth => answer);
    answer := nt3 IS NOT A SET; -- true, nt3 has duplicates
    testify(truth => answer);
    answer := nt1 IS EMPTY; -- false, nt1 has some members
    testify(truth => answer);
END;
/

```


Using Multilevel Collections

In addition to collections of scalar or object types, you can also create collections whose elements are collections. For example, you can create a nested table of varrays, a varray of varrays, a varray of nested tables, and so on.

When creating a nested table of nested tables as a column in SQL, check the syntax of the CREATE TABLE statement to see how to define the storage table.

[Example 5–25](#), [Example 5–26](#), and [Example 5–27](#) are some examples showing the syntax and possibilities for multilevel collections.

Example 5–25 Multilevel VARRAY

```

DECLARE
  TYPE t1 IS VARRAY(10) OF INTEGER;
  TYPE nt1 IS VARRAY(10) OF t1; -- multilevel varray type
  va t1 := t1(2,3,5);
  -- initialize multilevel varray
  nva nt1 := nt1(va, t1(55,6,73), t1(2,4), va);
  i INTEGER;
  val t1;
BEGIN
  -- multilevel access
  i := nva(2)(3); -- i will get value 73
  DBMS_OUTPUT.PUT_LINE('I = ' || i);
  -- add a new varray element to nva
  nva.EXTEND;
  -- replace inner varray elements
  nva(5) := t1(56, 32);
  nva(4) := t1(45,43,67,43345);
  -- replace an inner integer element
  nva(4)(4) := 1; -- replaces 43345 with 1
  -- add a new element to the 4th varray element
  -- and store integer 89 into it.
  nva(4).EXTEND;
  nva(4)(5) := 89;
END;
/

```

Example 5–26 Multilevel Nested Table

```

DECLARE
  TYPE tb1 IS TABLE OF VARCHAR2(20);
  TYPE Ntb1 IS TABLE OF tb1; -- table of table elements
  TYPE Tv1 IS VARRAY(10) OF INTEGER;
  TYPE ntb2 IS TABLE OF tv1; -- table of varray elements
  vtb1 tb1 := tb1('one', 'three');
  vntb1 ntb1 := ntb1(vtb1);
  vntb2 ntb2 := ntb2(tv1(3,5), tv1(5,7,3));
  -- table of varray elements
BEGIN
  vntb1.EXTEND;
  vntb1(2) := vntb1(1);
  -- delete the first element in vntb1
  vntb1.DELETE(1);
  -- delete the first string
  -- from the second table in the nested table
  vntb1(2).DELETE(1);
END;
/

```

Example 5–27 Multilevel Associative Array

```

DECLARE
  TYPE tb1 IS TABLE OF INTEGER INDEX BY PLS_INTEGER;
  -- the following is index-by table of index-by tables
  TYPE ntb1 IS TABLE OF tb1 INDEX BY PLS_INTEGER;
  TYPE va1 IS VARRAY(10) OF VARCHAR2(20);
  -- the following is index-by table of varray elements
  TYPE ntb2 IS TABLE OF va1 INDEX BY PLS_INTEGER;
  v1 va1 := va1('hello', 'world');
  v2 ntb1;
  v3 ntb2;
  v4 tb1;
  v5 tb1; -- empty table
BEGIN
  v4(1) := 34;
  v4(2) := 46456;
  v4(456) := 343;
  v2(23) := v4;
  v3(34) := va1(33, 456, 656, 343);
  -- assign an empty table to v2(35) and try again
  v2(35) := v5;
  v2(35)(2) := 78; -- it works now
END;
/

```

Using Collection Methods

Collection methods make collections easier to use, and make your applications easier to maintain. These methods include `COUNT`, `DELETE`, `EXISTS`, `EXTEND`, `FIRST`, `LAST`, `LIMIT`, `NEXT`, `PRIOR`, and `TRIM`.

A collection method is a built-in subprogram that operates on collections and is invoked using dot notation. The following apply to collection methods:

- Collection methods cannot be invoked from SQL statements.
- `EXTEND` and `TRIM` cannot be used with associative arrays.
- `EXISTS`, `COUNT`, `LIMIT`, `FIRST`, `LAST`, `PRIOR`, and `NEXT` are functions; `EXTEND`, `TRIM`, and `DELETE` are procedures.
- `EXISTS`, `PRIOR`, `NEXT`, `TRIM`, `EXTEND`, and `DELETE` take parameters corresponding to collection subscripts, which are usually integers but can also be strings for associative arrays.
- Only `EXISTS` can be applied to atomically null collections. If you apply another method to such collections, PL/SQL raises `COLLECTION_IS_NULL`.

For more information, see "[Collection Methods](#)" on page 13-23.

Topics:

- [Checking If a Collection Element Exists \(EXISTS Method\)](#)
- [Counting the Elements in a Collection \(COUNT Method\)](#)
- [Checking the Maximum Size of a Collection \(LIMIT Method\)](#)
- [Finding the First or Last Collection Element \(FIRST and LAST Methods\)](#)
- [Looping Through Collection Elements \(PRIOR and NEXT Methods\)](#)
- [Increasing the Size of a Collection \(EXTEND Method\)](#)

- [Decreasing the Size of a Collection \(TRIM Method\)](#)
- [Deleting Collection Elements \(DELETE Method\)](#)
- [Applying Methods to Collection Parameters](#)

Checking If a Collection Element Exists (EXISTS Method)

EXISTS (*n*) returns TRUE if the *n*th element in a collection exists. Otherwise, EXISTS (*n*) returns FALSE. By combining EXISTS with DELETE, you can work with sparse nested tables. You can also use EXISTS to avoid referencing a nonexistent element, which raises an exception. When passed an out-of-range subscript, EXISTS returns FALSE instead of raising SUBSCRIPT_OUTSIDE_LIMIT.

Example 5–28 Checking Whether a Collection Element EXISTS

```

DECLARE
    TYPE NumList IS TABLE OF INTEGER;
    n NumList := NumList(1,3,5,7);
BEGIN
    n.DELETE(2); -- Delete the second element
    IF n.EXISTS(1) THEN
        DBMS_OUTPUT.PUT_LINE('OK, element #1 exists.');
```

```
    END IF;
```

```
    IF n.EXISTS(2) = FALSE THEN
```

```
        DBMS_OUTPUT.PUT_LINE('OK, element #2 was deleted.');
```

```
    END IF;
```

```
    IF n.EXISTS(99) = FALSE THEN
```

```
        DBMS_OUTPUT.PUT_LINE('OK, element #99 does not exist at all.');
```

```
    END IF;
```

```
END;
```

```
/
```

Counting the Elements in a Collection (COUNT Method)

COUNT returns the number of elements that a collection currently contains.

Example 5–29 Counting Collection Elements with COUNT

```

DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    n NumList := NumList(2,4,6,8);
    -- Collection starts with 4 elements.
BEGIN
    DBMS_OUTPUT.PUT_LINE
        ('There are ' || n.COUNT || ' elements in N.');
```

```
n.EXTEND(3); -- Add 3 new elements at the end.
```

```
DBMS_OUTPUT.PUT_LINE
```

```
    ('Now there are ' || n.COUNT || ' elements in N.');
```

```
n := NumList(86,99); -- Assign a completely new value with 2 elements.
```

```
DBMS_OUTPUT.PUT_LINE
```

```
    ('Now there are ' || n.COUNT || ' elements in N.');
```

```
n.TRIM(2); -- Remove the last 2 elements, leaving none.
```

```
DBMS_OUTPUT.PUT_LINE
```

```
    ('Now there are ' || n.COUNT || ' elements in N.');
```

```
END;
```

```
/
```

COUNT is useful because the current size of a collection is not always known. For example, you can fetch a column of Oracle data into a nested table, where the number of elements depends on the size of the result set.

For varrays, COUNT always equals LAST. You can increase or decrease the size of a varray using the EXTEND and TRIM methods, so the value of COUNT can change, up to the value of the LIMIT method.

For nested tables, COUNT normally equals LAST. But, if you delete elements from the middle of a nested table, COUNT becomes smaller than LAST. When tallying elements, COUNT ignores deleted elements. Using DELETE with no parameters sets COUNT to 0.

Checking the Maximum Size of a Collection (LIMIT Method)

For nested tables and associative arrays, which have no declared size, LIMIT returns NULL. For varrays, LIMIT returns the maximum number of elements that a varray can contain. You specify this limit in the type definition, and can change it later with the TRIM and EXTEND methods.

Example 5–30 Checking the Maximum Size of a Collection with LIMIT

```
DECLARE
  TYPE dnames_var IS VARRAY(7) OF VARCHAR2(30);
  dept_names dnames_var :=
    dnames_var('Shipping', 'Sales', 'Finance', 'Payroll');
BEGIN
  DBMS_OUTPUT.PUT_LINE
    ('dept_names has ' || dept_names.COUNT || ' elements now');
  DBMS_OUTPUT.PUT_LINE
    ('dept_names''s type can hold a maximum of '
     || dept_names.LIMIT || ' elements');
  DBMS_OUTPUT.PUT_LINE
    ('The maximum number you can use with '
     || 'dept_names.EXTEND() is '
     || (dept_names.LIMIT - dept_names.COUNT));
END;
/
```

Finding the First or Last Collection Element (FIRST and LAST Methods)

FIRST and LAST return the first and last (smallest and largest) index numbers in a collection that uses integer subscripts.

For an associative array with VARCHAR2 key values, the lowest and highest key values are returned. By default, the order is based on the binary values of the characters in the string. If the NLS_COMP initialization parameter is set to ANSI, the order is based on the locale-specific sort order specified by the NLS_SORT initialization parameter.

If the collection is empty, FIRST and LAST return NULL. If the collection contains only one element, FIRST and LAST return the same index value.

Example 5–31 shows how to use FIRST and LAST to iterate through the elements in a collection that has consecutive subscripts.

Example 5–31 Using FIRST and LAST with a Collection

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

```

BEGIN
  DBMS_OUTPUT.PUT_LINE('N's first subscript is ' || n.FIRST);
  DBMS_OUTPUT.PUT_LINE('N's last subscript is ' || n.LAST);
  -- When the subscripts are consecutive starting at 1,
  -- it's simple to loop through them.
  FOR i IN n.FIRST .. n.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE('Element #' || i || ' = ' || n(i));
  END LOOP;
  n.DELETE(2); -- Delete second element.
  -- When the subscripts have gaps
  -- or the collection might be uninitialized,
  -- the loop logic is more extensive.
  -- Start at the first element
  -- and look for the next element until there are no more.
  IF n IS NOT NULL THEN
    counter := n.FIRST;
    WHILE counter IS NOT NULL
    LOOP
      DBMS_OUTPUT.PUT_LINE
        ('Element #' || counter || ' = ' || n(counter));
      counter := n.NEXT(counter);
    END LOOP;
  ELSE
    DBMS_OUTPUT.PUT_LINE('N is null, nothing to do.');
```

For varrays, `FIRST` always returns 1 and `LAST` always equals `COUNT`.

For nested tables, normally `FIRST` returns 1 and `LAST` equals `COUNT`. But if you delete elements from the beginning of a nested table, `FIRST` returns a number larger than 1. If you delete elements from the middle of a nested table, `LAST` becomes larger than `COUNT`.

When scanning elements, `FIRST` and `LAST` ignore deleted elements.

Looping Through Collection Elements (`PRIOR` and `NEXT` Methods)

`PRIOR(n)` returns the index number that precedes index `n` in a collection. `NEXT(n)` returns the index number that succeeds index `n`. If `n` has no predecessor, `PRIOR(n)` returns `NULL`. If `n` has no successor, `NEXT(n)` returns `NULL`.

For associative arrays with `VARCHAR2` keys, these methods return the appropriate key value; ordering is based on the binary values of the characters in the string, unless the `NLS_COMP` initialization parameter is set to `ANSI`, in which case the ordering is based on the locale-specific sort order specified by the `NLS_SORT` initialization parameter.

These methods are more reliable than looping through a fixed set of subscript values, because elements might be inserted or deleted from the collection during the loop. This is especially true for associative arrays, where the subscripts might not be in consecutive order and so the sequence of subscripts might be (1,2,4,8,16) or ('A','E','I','O','U').

Example 5-32 Using `PRIOR` and `NEXT` to Access Collection Elements

```

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  n NumList := NumList(1966,1971,1984,1989,1999);
```

```

BEGIN
  DBMS_OUTPUT.PUT_LINE('The element after #2 is #' || n.NEXT(2));
  DBMS_OUTPUT.PUT_LINE('The element before #2 is #' || n.PRIOR(2));
  n.DELETE(3);
  -- Delete an element to show how NEXT can handle gaps.
  DBMS_OUTPUT.PUT_LINE
    ('Now the element after #2 is #' || n.NEXT(2));
  IF n.PRIOR(n.FIRST) IS NULL THEN
    DBMS_OUTPUT.PUT_LINE
      ('Can't get PRIOR of the first element or NEXT of the last.');
```

You can use PRIOR or NEXT to traverse collections indexed by any series of subscripts. [Example 5-33](#) uses NEXT to traverse a nested table from which some elements were deleted.

Example 5-33 Using NEXT to Access Elements of a Nested Table

```

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  n NumList := NumList(1,3,5,7);
  counter INTEGER;
BEGIN
  n.DELETE(2); -- Delete second element.
  -- When the subscripts have gaps,
  -- loop logic is more extensive.
  -- Start at first element and look for next element
  -- until there are no more.
  counter := n.FIRST;
  WHILE counter IS NOT NULL
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Counting up: Element #' || counter || ' = ' || n(counter));
    counter := n.NEXT(counter);
  END LOOP;
  -- Run the same loop in reverse order.
  counter := n.LAST;
  WHILE counter IS NOT NULL
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Counting down: Element #' || counter || ' = ' || n(counter));
    counter := n.PRIOR(counter);
  END LOOP;
END;
/
```

When traversing elements, PRIOR and NEXT skip over deleted elements.

Increasing the Size of a Collection (EXTEND Method)

To increase the size of a nested table or varray, use EXTEND.

This procedure has three forms:

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

You cannot use `EXTEND` with index-by tables. You cannot use `EXTEND` to add elements to an uninitialized collection. If you impose the `NOT NULL` constraint on a `TABLE` or `VARRAY` type, you cannot apply the first two forms of `EXTEND` to collections of that type.

`EXTEND` operates on the internal size of a collection, which includes any deleted elements. This refers to deleted elements after using `DELETE (n)`, but not `DELETE` without parameters which completely removes all elements. If `EXTEND` encounters deleted elements, it includes them in its tally. PL/SQL keeps placeholders for deleted elements, so that you can re-create them by assigning new values.

Example 5-34 Using `EXTEND` to Increase the Size of a Collection

```

DECLARE
  TYPE NumList IS TABLE OF INTEGER;
  n NumList := NumList(2,4,6,8);
  x NumList := NumList(1,3);
  PROCEDURE print_numlist(the_list NumList) IS
    output VARCHAR2(128);
  BEGIN
    FOR i IN the_list.FIRST .. the_list.LAST
    LOOP
      output :=
        output || NVL(TO_CHAR(the_list(i)), 'NULL') || ' ';
    END LOOP;
    DBMS_OUTPUT.PUT_LINE(output);
  END;
BEGIN
  DBMS_OUTPUT.PUT_LINE
    ('At first, N has ' || n.COUNT || ' elements. ');
  n.EXTEND(5); -- Add 5 elements at the end.
  DBMS_OUTPUT.PUT_LINE
    ('Now N has ' || n.COUNT || ' elements. ');
-- Elements 5, 6, 7, 8, and 9 are all NULL.
  print_numlist(n);
  DBMS_OUTPUT.PUT_LINE
    ('At first, X has ' || x.COUNT || ' elements. ');
  x.EXTEND(4,2); -- Add 4 elements at the end.
  DBMS_OUTPUT.PUT_LINE
    ('Now X has ' || x.COUNT || ' elements. ');
-- Elements 3, 4, 5, and 6 are copies of element #2.
  print_numlist(x);
END;
/

```

When it includes deleted elements, the internal size of a nested table differs from the values returned by `COUNT` and `LAST`. This refers to deleted elements after using `DELETE (n)`, but not `DELETE` without parameters which completely removes all elements. For example, if you initialize a nested table with five elements, then delete elements 2 and 5, the internal size is 5, `COUNT` returns 3, and `LAST` returns 4. All deleted elements, regardless of position, are treated alike.

Decreasing the Size of a Collection (TRIM Method)

This procedure has two forms:

- `TRIM` removes one element from the end of a collection.
- `TRIM (n)` removes n elements from the end of a collection.

If you want to remove all elements, use DELETE without parameters.

For example, this statement removes the last three elements from nested table courses:

Example 5–35 Using TRIM to Decrease the Size of a Collection

```

DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    n NumList := NumList(1,2,3,5,7,11);
    PROCEDURE print_numlist(the_list NumList) IS
        output VARCHAR2(128);
    BEGIN
        IF n.COUNT = 0 THEN
            DBMS_OUTPUT.PUT_LINE('No elements in collection. ');
        ELSE
            FOR i IN the_list.FIRST .. the_list.LAST
            LOOP
                output :=
                    output || NVL(TO_CHAR(the_list(i), 'NULL') || ' ';
            END LOOP;
            DBMS_OUTPUT.PUT_LINE(output);
        END IF;
    END;
BEGIN
    print_numlist(n);
    n.TRIM(2); -- Remove last 2 elements.
    print_numlist(n);
    n.TRIM; -- Remove last element.
    print_numlist(n);
    n.TRIM(n.COUNT); -- Remove all remaining elements.
    print_numlist(n);
-- If too many elements are specified,
-- TRIM raises the exception SUBSCRIPT_BEYOND_COUNT.
    BEGIN
        n := NumList(1,2,3);
        n.TRIM(100);
        EXCEPTION
            WHEN SUBSCRIPT_BEYOND_COUNT THEN
                DBMS_OUTPUT.PUT_LINE
                    ('There weren't 100 elements to be trimmed. ');
    END;
-- When elements are removed by DELETE,
-- placeholders are left behind.
-- TRIM counts these placeholders
-- as it removes elements from the end.
    n := NumList(1,2,3,4);
    n.DELETE(3); -- delete element 3
-- At this point, n contains elements (1,2,4).
-- TRIMming the last 2 elements
-- removes the 4 and the placeholder, not 4 and 2.
    n.TRIM(2);
    print_numlist(n);
END;
/

```

If *n* is too large, TRIM(*n*) raises SUBSCRIPT_BEYOND_COUNT.

TRIM operates on the internal size of a collection. If TRIM encounters deleted elements, it includes them in its tally. This refers to deleted elements after using DELETE(*n*), but not DELETE without parameters which completely removes all elements.

Example 5–36 Using TRIM on Deleted Elements

```

DECLARE
  TYPE CourseList IS TABLE OF VARCHAR2(10);
  courses CourseList;
BEGIN
  courses := CourseList('Biol 4412', 'Psyc 3112', 'Anth 3001');
  courses.DELETE(courses.LAST); -- delete element 3
  /* At this point, COUNT equals 2, the number of valid
  elements remaining. So, you might expect the next
  statement to empty the nested table by trimming
  elements 1 and 2. Instead, it trims valid element 2
  and deleted element 3 because TRIM includes deleted
  elements in its tally. */
  courses.TRIM(courses.COUNT);
  DBMS_OUTPUT.PUT_LINE(courses(1)); -- prints 'Biol 4412'
END;
/

```

In general, do not depend on the interaction between TRIM and DELETE. It is better to treat nested tables like fixed-size arrays and use only DELETE, or to treat them like stacks and use only TRIM and EXTEND.

Because PL/SQL does not keep placeholders for trimmed elements, you cannot replace a trimmed element simply by assigning it a new value.

Deleting Collection Elements (DELETE Method)

This procedure has various forms:

- DELETE with no parameters removes all elements from a collection, setting COUNT to 0.
- DELETE(*n*) removes the *n*th element from an associative array with a numeric key or a nested table. If the associative array has a string key, the element corresponding to the key value is deleted. If *n* is null, DELETE(*n*) does nothing.
- DELETE(*m,n*) removes all elements in the range *m..n* from an associative array or nested table. If *m* is larger than *n* or if *m* or *n* is NULL, DELETE(*m,n*) does nothing.

For example:

Example 5–37 Using the DELETE Method on a Collection

```

DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  n NumList := NumList(10,20,30,40,50,60,70,80,90,100);
  TYPE NickList IS TABLE OF VARCHAR2(64) INDEX BY VARCHAR2(32);
  nicknames NickList;
BEGIN
  n.DELETE(2); -- deletes element 2
  n.DELETE(3,6); -- deletes elements 3 through 6
  n.DELETE(7,7); -- deletes element 7
  n.DELETE(6,3); -- does nothing since 6 > 3
  n.DELETE; -- deletes all elements
  nicknames('Bob') := 'Robert';
  nicknames('Buffy') := 'Esmerelda';
  nicknames('Chip') := 'Charles';
  nicknames('Dan') := 'Daniel';
  nicknames('Fluffy') := 'Ernestina';
  nicknames('Rob') := 'Robert';
  -- following deletes element denoted by this key

```

```

nicknames.DELETE('Chip');
-- following deletes elements with keys in this alphabetic range
nicknames.DELETE('Buffy', 'Fluffy');
END;
/

```

Varrays always have consecutive subscripts, so you cannot delete individual elements except from the end by using the TRIM method. You can use DELETE without parameters to delete all elements.

If an element to be deleted does not exist, DELETE(*n*) simply skips it; no exception is raised. PL/SQL keeps placeholders for deleted elements, so you can replace a deleted element by assigning it a new value. This refers to deleted elements after using DELETE(*n*), but not DELETE without parameters which completely removes all elements.

DELETE lets you maintain sparse nested tables. You can store sparse nested tables in the database, just like any other nested tables.

The amount of memory allocated to a nested table can increase or decrease dynamically. As you delete elements, memory is freed page by page. If you delete the entire table, all the memory is freed.

Applying Methods to Collection Parameters

Within a subprogram, a collection parameter assumes the properties of the argument bound to it. You can apply the built-in collection methods (FIRST, LAST, COUNT, and so on) to such parameters. You can create general-purpose subprograms that take collection parameters and iterate through their elements, add or delete elements, and so on. For varray parameters, the value of LIMIT is always derived from the parameter type definition, regardless of the parameter mode.

Avoiding Collection Exceptions

[Example 5–38](#) shows various collection exceptions that are predefined in PL/SQL. The example also includes notes on how to avoid the problems.

Example 5–38 Collection Exceptions

```

DECLARE
  TYPE WordList IS TABLE OF VARCHAR2(5);
  words WordList;
  err_msg VARCHAR2(100);
  PROCEDURE display_error IS
  BEGIN
    err_msg := SUBSTR(SQLERRM, 1, 100);
    DBMS_OUTPUT.PUT_LINE('Error message = ' || err_msg);
  END;
BEGIN
  BEGIN
    words(1) := 10; -- Raises COLLECTION_IS_NULL
    -- A constructor has not been used yet.
    -- Note: This exception applies to varrays and nested tables,
    -- but not to associative arrays which do not need a constructor.
  EXCEPTION
    WHEN OTHERS THEN display_error;
  END;
  -- After using a constructor, we can assign values to the elements.
  words := WordList('1st', '2nd', '3rd'); -- 3 elements created

```

```

-- Any expression that returns a VARCHAR2(5) is valid.
words(3) := words(1) || '+2';
BEGIN
  words(3) := 'longer than 5 characters'; -- Raises VALUE_ERROR
-- The assigned value is too long.
  EXCEPTION
    WHEN OTHERS THEN display_error;
END;
BEGIN
  words('B') := 'dunno'; -- Raises VALUE_ERROR
-- The subscript (B) of a nested table must be an integer.
-- Note: Also, NULL is not allowed as a subscript.
  EXCEPTION
    WHEN OTHERS THEN display_error;
END;
BEGIN
  words(0) := 'zero'; -- Raises SUBSCRIPT_OUTSIDE_LIMIT
-- Subscript 0 is outside the allowed subscript range.
  EXCEPTION
    WHEN OTHERS THEN display_error;
END;
BEGIN
  words(4) := 'maybe'; -- Raises SUBSCRIPT_BEYOND_COUNT
-- The subscript (4) exceeds the number of elements in the table.
-- To add new elements, invoke the EXTEND method first.
  EXCEPTION
    WHEN OTHERS THEN display_error;
END;
BEGIN
  words.DELETE(1);
  IF words(1) = 'First' THEN NULL; END IF;
  -- Raises NO_DATA_FOUND
-- The element with subscript (1) was deleted.
  EXCEPTION
    WHEN OTHERS THEN display_error;
END;
END;
/

```

Execution continues in [Example 5-38](#) because the raised exceptions are handled in sub-blocks. See "[Continuing Execution After an Exception Is Raised](#)" on page 11-16. For information about the use of SQLERRM with exception handling, see "[Retrieving the Error Code and Error Message \(SQLCODE and SQLERRM Functions\)](#)" on page 11-14.

The following list summarizes when a given exception is raised.

Collection Exception	Raised when...
COLLECTION_IS_NULL	you try to operate on an atomically null collection.
NO_DATA_FOUND	a subscript designates an element that was deleted, or a nonexistent element of an associative array.
SUBSCRIPT_BEYOND_COUNT	a subscript exceeds the number of elements in a collection.
SUBSCRIPT_OUTSIDE_LIMIT	a subscript is outside the allowed range.
VALUE_ERROR	a subscript is null or not convertible to the key type. This exception might occur if the key is defined as a PLS_INTEGER range, and the subscript is outside this range.

See Also: ["Summary of Predefined PL/SQL Exceptions"](#) on page 11-4

In some cases, you can pass invalid subscripts to a method without raising an exception. For example, when you pass a null subscript to `DELETE (n)`, it does nothing. You can replace deleted elements by assigning values to them, without raising `NO_DATA_FOUND`. This refers to deleted elements after using `DELETE (n)`, but not `DELETE` without parameters which completely removes all elements. For example:

Example 5–39 How Invalid Subscripts are Handled with DELETE(n)

```
DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    nums NumList := NumList(10,20,30); -- initialize table
BEGIN
    nums.DELETE(-1); -- does not raise SUBSCRIPT_OUTSIDE_LIMIT
    nums.DELETE(3); -- delete 3rd element
    DBMS_OUTPUT.PUT_LINE(nums.COUNT); -- prints 2
    nums(3) := 30; -- allowed; does not raise NO_DATA_FOUND
    DBMS_OUTPUT.PUT_LINE(nums.COUNT); -- prints 3
END;
/
```

Packaged collection types and local collection types are never compatible. For example, suppose you want to invoke the following packaged procedure:

Example 5–40 Incompatibility Between Package and Local Collection Types

```
CREATE PACKAGE pkg AS
    TYPE NumList IS TABLE OF NUMBER;
    PROCEDURE print_numlist (nums NumList);
END pkg;
/
CREATE PACKAGE BODY pkg AS
    PROCEDURE print_numlist (nums NumList) IS
    BEGIN
        FOR i IN nums.FIRST..nums.LAST LOOP
            DBMS_OUTPUT.PUT_LINE(nums(i));
        END LOOP;
    END;
END pkg;
/

DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    n1 pkg.NumList := pkg.NumList(2,4); -- type from the package.
    n2 NumList := NumList(6,8); -- local type.
BEGIN
    pkg.print_numlist(n1); -- type from pkg is legal
    -- The packaged procedure cannot accept
    -- a value of the local type (n2)
    -- pkg.print_numlist(n2); -- Causes a compilation error.
END;
/
```

The second procedure call fails, because the packaged and local `VARRAY` types are incompatible despite their identical definitions.

Defining and Declaring Records

To create records, you define a `RECORD` type, then declare records of that type. You can also create or find a table, view, or PL/SQL cursor with the values you want, and use the `%ROWTYPE` attribute to create a matching record.

You can define `RECORD` types in the declarative part of any PL/SQL block, subprogram, or package. When you define your own `RECORD` type, you can specify a `NOT NULL` constraint on fields, or give them default values. See ["Record Definition"](#) on page 13-115.

[Example 5-42](#) and [Example 5-42](#) illustrate record type declarations.

Example 5-41 Declaring and Initializing a Simple Record Type

```
DECLARE
    TYPE DeptRecTyp IS RECORD (
        deptid NUMBER(4) NOT NULL := 99,
        dname departments.department_name%TYPE,
        loc departments.location_id%TYPE,
        region regions%ROWTYPE );
    dept_rec DeptRecTyp;
BEGIN
    dept_rec.dname := 'PURCHASING';
END;
/
```

Example 5-42 Declaring and Initializing Record Types

```
DECLARE
-- Declare a record type with 3 fields.
    TYPE rec1_t IS RECORD
        (field1 VARCHAR2(16), field2 NUMBER, field3 DATE);
-- For any fields declared NOT NULL, we must supply a default value.
    TYPE rec2_t IS RECORD (id INTEGER NOT NULL := -1,
        name VARCHAR2(64) NOT NULL := '[anonymous]');
-- Declare record variables of the types declared
    rec1 rec1_t;
    rec2 rec2_t;
-- Declare a record variable that can hold
-- a row from the EMPLOYEES table.
-- The fields of the record automatically match the names and
-- types of the columns.
-- Don't need a TYPE declaration in this case.
    rec3 employees%ROWTYPE;
-- Or mix fields that are table columns with user-defined fields.
    TYPE rec4_t IS RECORD (first_name employees.first_name%TYPE,
        last_name employees.last_name%TYPE,
        rating NUMBER);

    rec4 rec4_t;
BEGIN
-- Read and write fields using dot notation
    rec1.field1 := 'Yesterday';
    rec1.field2 := 65;
    rec1.field3 := TRUNC(SYSDATE-1);
-- Didn't fill name field, so it takes default value
    DBMS_OUTPUT.PUT_LINE(rec2.name);
END;
/
```

To store a record in the database, you can specify it in an `INSERT` or `UPDATE` statement, if its fields match the columns in the table:

You can use `%TYPE` to specify a field type corresponding to a table column type. Your code keeps working even if the column type is changed (for example, to increase the length of a `VARCHAR2` or the precision of a `NUMBER`). [Example 5-43](#) defines `RECORD` types to hold information about a department:

Example 5-43 Using %ROWTYPE to Declare a Record

```
DECLARE
-- Best: use %ROWTYPE instead of specifying each column.
-- Use <cursor>%ROWTYPE instead of <table>%ROWTYPE because
-- we only want some columns.
-- Declaring cursor doesn't run query or affect performance.
  CURSOR c1 IS
    SELECT department_id, department_name, location_id
    FROM departments;
  rec1 c1%ROWTYPE;
-- Use <column>%TYPE in field declarations to avoid problems if
-- the column types change.
  TYPE DeptRec2 IS RECORD
    (dept_id departments.department_id%TYPE,
     dept_name departments.department_name%TYPE,
     dept_loc departments.location_id%TYPE);
  rec2 DeptRec2;
-- Write each field name, specifying type directly
-- (clumsy and unmaintainable for working with table data
-- use only for all-PL/SQL code).
  TYPE DeptRec3 IS RECORD (dept_id NUMBER,
                           dept_name VARCHAR2(14),
                           dept_loc VARCHAR2(13));

  rec3 DeptRec3;
BEGIN
  NULL;
END;
/
```

PL/SQL lets you define records that contain objects, collections, and other records (called nested records). However, records cannot be attributes of object types.

Using Records as Subprogram Parameters and Function Return Values

Records are easy to process using stored subprograms because you can pass just one parameter, instead of a separate parameter for each field. For example, you can fetch a table row from the `EMPLOYEES` table into a record, and then pass that row as a parameter to a function that computes that employee's vacation allowance. The function can access all the information about that employee by referring to the fields in the record.

The next example shows how to return a record from a function. To make the record type visible across multiple stored subprograms, declare the record type in a package specification.

Example 5-44 Returning a Record from a Function

```
DECLARE
  TYPE EmpRecTyp IS RECORD (
    emp_id NUMBER(6),
```

```

        salary          NUMBER(8,2));
CURSOR desc_salary RETURN EmpRecTyp IS
    SELECT employee_id, salary
    FROM employees
    ORDER BY salary DESC;
emp_rec      EmpRecTyp;
FUNCTION nth_highest_salary (n INTEGER) RETURN EmpRecTyp IS
BEGIN
    OPEN desc_salary;
    FOR i IN 1..n LOOP
        FETCH desc_salary INTO emp_rec;
    END LOOP;
    CLOSE desc_salary;
    RETURN emp_rec;
END nth_highest_salary;
BEGIN
    NULL;
END;
/

```

Like scalar variables, user-defined records can be declared as the formal parameters of subprograms:

Example 5–45 Using a Record as Parameter to a Procedure

```

DECLARE
    TYPE EmpRecTyp IS RECORD (
        emp_id      NUMBER(6),
        emp_sal     NUMBER(8,2) );
    PROCEDURE raise_salary (emp_info EmpRecTyp) IS
    BEGIN
        UPDATE employees SET salary = salary + salary * .10
        WHERE employee_id = emp_info.emp_id;
    END raise_salary;
BEGIN
    NULL;
END;
/

```

You can declare and reference nested records. That is, a record can be the component of another record.

Example 5–46 Declaring a Nested Record

```

DECLARE
    TYPE TimeTyp IS RECORD ( minutes SMALLINT, hours SMALLINT );
    TYPE MeetingTyp IS RECORD (
        day      DATE,
        time_of TimeTyp,          -- nested record
        dept     departments%ROWTYPE,
        -- nested record representing a table row
        place    VARCHAR2(20),
        purpose  VARCHAR2(50) );
    meeting MeetingTyp;
    seminar MeetingTyp;
BEGIN
    -- Can assign one nested record to another
    -- if they are of the same datatype
    seminar.time_of := meeting.time_of;
END;
/

```

Such assignments are allowed even if the containing records have different datatypes.

Assigning Values to Records

To set all the fields in a record to default values, assign to it an uninitialized record of the same type, as shown in [Example 5–47](#).

Example 5–47 Assigning Default Values to a Record

```
DECLARE
    TYPE RecordTyp IS RECORD (field1 NUMBER,
                              field2 VARCHAR2(32) DEFAULT 'something');

    rec1 RecordTyp;
    rec2 RecordTyp;
BEGIN
    -- At first, rec1 has the values we assign.
    rec1.field1 := 100; rec1.field2 := 'something else';
    -- Assigning an empty record to rec1
    -- resets fields to their default values.
    -- Field1 is NULL and field2 is 'something'
    -- due to the DEFAULT clause
    rec1 := rec2;
    DBMS_OUTPUT.PUT_LINE
        ('Field1 = ' || NVL(TO_CHAR(rec1.field1), '<NULL>') || ',
         field2 = ' || rec1.field2);
END;
/
```

You can assign a value to a field in a record using an assignment statement with dot notation:

```
emp_info.last_name := 'Fields';
```

Note that values are assigned separately to each field of a record in [Example 5–47](#). You cannot assign a list of values to a record using an assignment statement. There is no constructor-like notation for records.

You can assign values to all fields at once only if you assign a record to another record with the same datatype. Having fields that match exactly is not enough, as shown in [Example 5–48](#).

Example 5–48 Assigning All the Fields of a Record in One Statement

```
DECLARE
    -- Two identical type declarations.
    TYPE DeptRec1 IS RECORD
        (dept_num NUMBER(2), dept_name VARCHAR2(14));
    TYPE DeptRec2 IS RECORD
        (dept_num NUMBER(2), dept_name VARCHAR2(14));
    dept1_info DeptRec1;
    dept2_info DeptRec2;
    dept3_info DeptRec2;
BEGIN
    -- Not allowed; different datatypes,
    -- even though fields are the same.
    --     dept1_info := dept2_info;
    -- This assignment is OK because the records have the same type.
    dept2_info := dept3_info;
END;
```


/

You can assign a %ROWTYPE record to a user-defined record if their fields match in number and order, and corresponding fields have the same datatypes:

```

DECLARE
    TYPE RecordTyp IS RECORD (last employees.last_name%TYPE,
                              id employees.employee_id%TYPE);
    CURSOR c1 IS SELECT last_name, employee_id FROM employees;
-- Rec1 and rec2 have different types,
-- but because rec2 is based on a %ROWTYPE,
-- we can assign it to rec1 as long as they have
-- the right number of fields and
-- the fields have the right datatypes.
    rec1 RecordTyp;
    rec2 c1%ROWTYPE;
BEGIN
    SELECT last_name, employee_id INTO rec2
    FROM employees WHERE ROWNUM < 2;
    WHERE ROWNUM < 2;
    rec1 := rec2;
    DBMS_OUTPUT.PUT_LINE
    ('Employee #' || rec1.id || ' = ' || rec1.last);
END;
/

```

You can also use the `SELECT` or `FETCH` statement to fetch column values into a record. The columns in the select-list must appear in the same order as the fields in your record.

Example 5-49 Using SELECT INTO to Assign Values in a Record

```

DECLARE
    TYPE RecordTyp IS RECORD (last employees.last_name%TYPE,
                              id employees.employee_id%TYPE);
    rec1 RecordTyp;
BEGIN
    SELECT last_name, employee_id INTO rec1
    FROM employees WHERE ROWNUM < 2;
    WHERE ROWNUM < 2;
    DBMS_OUTPUT.PUT_LINE
    ('Employee #' || rec1.id || ' = ' || rec1.last);
END;
/

```

Topics:

- [Comparing Records](#)
- [Inserting Records Into the Database](#)
- [Updating the Database with Record Values](#)
- [Restrictions on Record Inserts and Updates](#)
- [Querying Data Into Collections of Records](#)

Comparing Records

Records cannot be tested for nullity, or compared for equality, or inequality. If you want to make such comparisons, write your own function that accepts two records as parameters and does the appropriate checks or comparisons on the corresponding fields.

Inserting Records Into the Database

A PL/SQL-only extension of the `INSERT` statement lets you insert records into database rows, using a single variable of type `RECORD` or `%ROWTYPE` in the `VALUES` clause instead of a list of fields. That makes your code more readable and maintainable.

If you issue the `INSERT` through the `FORALL` statement, you can insert values from an entire collection of records. The number of fields in the record must equal the number of columns listed in the `INTO` clause, and corresponding fields and columns must have compatible datatypes. To make sure the record is compatible with the table, you might find it most convenient to declare the variable as the type `table_name%ROWTYPE`.

[Example 5–50](#) declares a record variable using a `%ROWTYPE` qualifier. You can insert this variable without specifying a column list. The `%ROWTYPE` declaration ensures that the record attributes have exactly the same names and types as the table columns.

Example 5–50 Inserting a PL/SQL Record Using %ROWTYPE

```
DECLARE
    dept_info departments%ROWTYPE;
BEGIN
    -- department_id, department_name, and location_id
    -- are the table columns
    -- The record picks up these names from the %ROWTYPE
    dept_info.department_id := 300;
    dept_info.department_name := 'Personnel';
    dept_info.location_id := 1700;
    -- Using the %ROWTYPE means we can leave out the column list
    -- (department_id, department_name, and location_id)
    -- from the INSERT statement
    INSERT INTO departments VALUES dept_info;
END;
/
```

Updating the Database with Record Values

A PL/SQL-only extension of the `UPDATE` statement lets you update database rows using a single variable of type `RECORD` or `%ROWTYPE` on the right side of the `SET` clause, instead of a list of fields.

If you issue the `UPDATE` through the `FORALL` statement, you can update a set of rows using values from an entire collection of records. Also with an `UPDATE` statement, you can specify a record in the `RETURNING` clause to retrieve new values into a record. If you issue the `UPDATE` through the `FORALL` statement, you can retrieve new values from a set of updated rows into a collection of records.

The number of fields in the record must equal the number of columns listed in the `SET` clause, and corresponding fields and columns must have compatible datatypes.

You can use the keyword `ROW` to represent an entire row, as shown in [Example 5–51](#).

Example 5–51 Updating a Row Using a Record

```
DECLARE
    dept_info departments%ROWTYPE;
BEGIN
    -- department_id, department_name, and location_id
    -- are the table columns
    -- The record picks up these names from the %ROWTYPE.
    dept_info.department_id := 300;
```

```

dept_info.department_name := 'Personnel';
dept_info.location_id := 1700;
-- The fields of a %ROWTYPE
-- can completely replace the table columns
-- The row will have values for the filled-in columns, and null
-- for any other columns
UPDATE departments SET ROW = dept_info WHERE department_id = 300;
END;
/

```

The keyword `ROW` is allowed only on the left side of a `SET` clause. The argument to `SET ROW` must be a real PL/SQL record, not a subquery that returns a single row. The record can also contain collections or objects.

The `INSERT`, `UPDATE`, and `DELETE` statements can include a `RETURNING` clause, which returns column values from the affected row into a PL/SQL record variable. This eliminates the need to `SELECT` the row after an insert or update, or before a delete.

By default, you can use this clause only when operating on exactly one row. When you use bulk SQL, you can use the form `RETURNING BULK COLLECT INTO` to store the results in one or more collections.

[Example 5-52](#) updates the salary of an employee and retrieves the employee's name, job title, and new salary into a record variable.

Example 5-52 Using the RETURNING INTO Clause with a Record

```

DECLARE
TYPE EmpRec IS RECORD (last_name employees.last_name%TYPE,
                        salary      employees.salary%TYPE);
emp_info EmpRec;
emp_id   NUMBER := 100;
BEGIN
UPDATE employees SET salary = salary * 1.1
WHERE employee_id = emp_id
RETURNING last_name, salary INTO emp_info;
DBMS_OUTPUT.PUT_LINE
('Just gave a raise to ' || emp_info.last_name ||
', who now makes ' || emp_info.salary);
ROLLBACK;
END;
/

```

Restrictions on Record Inserts and Updates

Currently, the following restrictions apply to record inserts/updates:

- Record variables are allowed only in the following 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.
- The following are not supported:
 - Nested record types
 - Functions that return a record
 - Record inserts and updates using the EXECUTE IMMEDIATE statement.

Querying Data Into Collections of Records

You can use the BULK COLLECT clause with a SELECT INTO or FETCH statement to retrieve a set of rows into a collection of records.

Example 5–53 Using BULK COLLECT with a SELECT INTO Statement

```

DECLARE
    TYPE EmployeeSet IS TABLE OF employees%ROWTYPE;
    underpaid EmployeeSet;
    -- Holds set of rows from EMPLOYEES table.
    CURSOR c1 IS SELECT first_name, last_name FROM employees;
    TYPE NameSet IS TABLE OF c1%ROWTYPE;
    some_names NameSet;
    -- Holds set of partial rows from EMPLOYEES table.
BEGIN
    -- With one query,
    -- bring all relevant data into collection of records.
    SELECT * BULK COLLECT INTO underpaid FROM employees
        WHERE salary < 5000 ORDER BY salary DESC;
    -- Process data by examining collection or passing it to
    -- eparate procedure, instead of writing loop to FETCH each row.
    DBMS_OUTPUT.PUT_LINE
        (underpaid.COUNT || ' people make less than 5000.');
```

```

    FOR i IN underpaid.FIRST .. underpaid.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE
            (underpaid(i).last_name || ' makes ' || underpaid(i).salary);
    END LOOP;
    -- We can also bring in just some of the table columns.
    -- Here we get the first and last names of 10 arbitrary employees.
    SELECT first_name, last_name
        BULK COLLECT INTO some_names
        FROM employees
        WHERE ROWNUM < 11;
    FOR i IN some_names.FIRST .. some_names.LAST
    LOOP
        DBMS_OUTPUT.PUT_LINE
            ('Employee = ' || some_names(i).first_name
            || ' ' || some_names(i).last_name);
    END LOOP;
END;
/
```

Using Static SQL

Static SQL is SQL that belongs to the PL/SQL language. This chapter describes static SQL and explains how to use it in PL/SQL programs.

Topics:

- [Description of Static SQL](#)
- [Managing Cursors in PL/SQL](#)
- [Querying Data with PL/SQL](#)
- [Using Subqueries](#)
- [Using Cursor Variables \(REF CURSORS\)](#)
- [Using Cursor Expressions](#)
- [Overview of Transaction Processing in PL/SQL](#)
- [Doing Independent Units of Work with Autonomous Transactions](#)

Description of Static SQL

Static SQL is SQL that belongs to the PL/SQL language; that is:

- [Data Manipulation Language \(DML\) Statements](#) (except `EXPLAIN PLAN`)
- [Transaction Control Language \(TCL\) Statements](#)
- [SQL Functions](#)
- [SQL Pseudocolumns](#)
- [SQL Operators](#)

Static SQL conforms to the current ANSI/ISO SQL standard.

Data Manipulation Language (DML) Statements

To manipulate Oracle data you can include DML operations, such as `INSERT`, `UPDATE`, and `DELETE` statements, directly in PL/SQL programs, without any special notation, as shown in [Example 6-1](#). You can also include the SQL `COMMIT` statement directly in a PL/SQL program; see "[Overview of Transaction Processing in PL/SQL](#)" on page 6-32.

Example 6-1 Data Manipulation with PL/SQL

```
CREATE TABLE employees_temp
  AS SELECT employee_id, first_name, last_name
```

```
FROM employees;
DECLARE
  emp_id          employees_temp.employee_id%TYPE;
  emp_first_name  employees_temp.first_name%TYPE;
  emp_last_name   employees_temp.last_name%TYPE;
BEGIN
  INSERT INTO employees_temp VALUES(299, 'Bob', 'Henry');
  UPDATE employees_temp
    SET first_name = 'Robert' WHERE employee_id = 299;
  DELETE FROM employees_temp WHERE employee_id = 299
    RETURNING first_name, last_name
      INTO emp_first_name, emp_last_name;
  COMMIT;
  DBMS_OUTPUT.PUT_LINE( emp_first_name || ' ' || emp_last_name);
END;
/
```

See Also: *Oracle Database SQL Language Reference* for information about the COMMIT statement

To find out how many rows are affected by DML statements, you can check the value of SQL%ROWCOUNT as shown in [Example 6-2](#).

Example 6-2 Checking SQL%ROWCOUNT After an UPDATE

```
CREATE TABLE employees_temp AS SELECT * FROM employees;
BEGIN
  UPDATE employees_temp
    SET salary = salary * 1.05 WHERE salary < 5000;
  DBMS_OUTPUT.PUT_LINE('Updated ' || SQL%ROWCOUNT || ' salaries.');
```

Wherever you can use literal values, or bind variables in some other programming language, you can directly substitute PL/SQL variables as shown in [Example 6-3](#).

Example 6-3 Substituting PL/SQL Variables

```
CREATE TABLE employees_temp
  AS SELECT first_name, last_name FROM employees;
DECLARE
  x VARCHAR2(20) := 'my_first_name';
  y VARCHAR2(25) := 'my_last_name';
BEGIN
  INSERT INTO employees_temp VALUES(x, y);
  UPDATE employees_temp SET last_name = x WHERE first_name = y;
  DELETE FROM employees_temp WHERE first_name = x;
  COMMIT;
END;
/
```

With this notation, you can use variables in place of values in the WHERE clause. To use variables in place of table names, column names, and so on, requires the EXECUTE IMMEDIATE statement that is explained in ["Using Native Dynamic SQL"](#) on page 7-2.

For information on the use of PL/SQL records with SQL to update and insert data, see ["Inserting Records Into the Database"](#) on page 5-36 and ["Updating the Database with Record Values"](#) on page 5-36.

For additional information on assigning values to PL/SQL variables, see "[Assigning SQL Query Results to PL/SQL Variables](#)" on page 2-20.

Note: When issuing a data manipulation (DML) statement in PL/SQL, there are some situations when the value of a variable is undefined after the statement is executed. These include:

- If a `FETCH` or `SELECT` statement raises any exception, then the values of the define variables after that statement are undefined.
 - If a DML statement affects zero rows, the values of the `OUT` binds after the DML executes are undefined. This does not apply to a `BULK` or multiple-row operation.
-
-

Transaction Control Language (TCL) Statements

Oracle is transaction oriented; that is, Oracle uses transactions to ensure data integrity. A transaction is a series of SQL data manipulation statements that does a logical unit of work. For example, two `UPDATE` statements might credit one bank account and debit another. It is important not to allow one operation to succeed while the other fails.

At the end of a transaction that makes database changes, Oracle makes all the changes permanent or undoes them all. If your program fails in the middle of a transaction, Oracle detects the error and rolls back the transaction, restoring the database to its former state.

You use the `COMMIT`, `ROLLBACK`, `SAVEPOINT`, and `SET TRANSACTION` statements to control transactions. `COMMIT` makes permanent any database changes made during the current transaction. `ROLLBACK` ends the current transaction and undoes any changes made since the transaction began. `SAVEPOINT` marks the current point in the processing of a transaction. Used with `ROLLBACK`, `SAVEPOINT` undoes part of a transaction. `SET TRANSACTION` sets transaction properties such as read/write access and isolation level. See "[Overview of Transaction Processing in PL/SQL](#)" on page 6-32.

SQL Functions

[Example 6-4](#) shows some queries that invoke SQL functions.

Example 6-4 Invoking the SQL COUNT Function in PL/SQL

```
DECLARE
  job_count NUMBER;
  emp_count NUMBER;
BEGIN
  SELECT COUNT(DISTINCT job_id) INTO job_count FROM employees;
  SELECT COUNT(*) INTO emp_count FROM employees;
END;
/
```

SQL Pseudocolumns

PL/SQL recognizes the SQL pseudocolumns `CURRVAL`, `LEVEL`, `NEXTVAL`, `ROWID`, and `ROWNUM`. However, there are limitations on the use of pseudocolumns, including the restriction on the use of some pseudocolumns in assignments or conditional tests. For additional information, including restrictions, on the use of SQL pseudocolumns, see *Oracle Database SQL Language Reference*.

Topics:

- [CURRVAL and NEXTVAL](#)
- [LEVEL](#)
- [ROWID](#)
- [ROWNUM](#)

CURRVAL and NEXTVAL

A **sequence** is a schema object that generates sequential numbers. When you create a sequence, you can specify its initial value and an increment. `CURRVAL` returns the current value in a specified sequence. Before you can reference `CURRVAL` in a session, you must use `NEXTVAL` to generate a number. A reference to `NEXTVAL` stores the current sequence number in `CURRVAL`. `NEXTVAL` increments the sequence and returns the next value. To get the current or next value in a sequence, use dot notation:

```
sequence_name.CURRVAL  
sequence_name.NEXTVAL
```

The `sequence_name` can be either local or remote.

Each time you reference the `NEXTVAL` value of a sequence, the sequence is incremented immediately and permanently, whether you commit or roll back the transaction.

After creating a sequence, you can use it to generate unique sequence numbers for transaction processing.

[Example 6-5](#) generates a new sequence number and refers to that number in more than one statement. (The sequence must already exist. To create a sequence, use the SQL statement `CREATE SEQUENCE`.)

Example 6-5 Using CURRVAL and NEXTVAL

```
CREATE TABLE employees_temp  
AS SELECT employee_id, first_name, last_name  
FROM employees;  
  
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 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 the database,
-- there might be gaps in the sequence.

-- Use CURRVAL to specify the record to delete:
seq_value := employees_seq.CURRVAL;
DELETE FROM employees_temp2 WHERE employee_id = seq_value;

-- Update employee_id with NEXTVAL for specified record:
UPDATE employees_temp SET employee_id = employees_seq.NEXTVAL
  WHERE first_name = 'Lynette' AND last_name = 'Smith';

-- Display final value of CURRVAL:
seq_value := employees_seq.CURRVAL;
DBMS_OUTPUT.PUT_LINE
  ('Ending sequence value: ' || TO_CHAR(seq_value));
END;
/

```

Usage Notes

- You can use *sequence_name.CURRVAL* and *sequence_name.NEXTVAL* wherever you can use a NUMBER expression.
- Using *sequence_name.CURRVAL* or *sequence_name.NEXTVAL* to provide a default value for an object type 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 only once for every row in which it appears).

LEVEL

You use `LEVEL` with the `SELECT CONNECT BY` statement to organize rows from a database table into a tree structure. You might use sequence numbers to give each row a unique identifier, and refer to those identifiers from other rows to set up parent-child relationships. `LEVEL` returns the level number of a node in a tree structure. The root is level 1, children of the root are level 2, grandchildren are level 3, and so on.

In the `START WITH` clause, you specify a condition that identifies the root of the tree. You specify the direction in which the query traverses the tree (down from the root or up from the branches) with the `PRIOR` operator.

ROWID

`ROWID` returns the rowid (binary address) of a row in a database table. You can use variables of type `UROWID` to store rowids in a readable format.

When you select or fetch a physical rowid into a `UROWID` variable, you can use the function `ROWIDTOCHAR`, which converts the binary value to a character string. You can compare the `UROWID` variable to the `ROWID` pseudocolumn in the `WHERE` clause of an `UPDATE` or `DELETE` statement to identify the latest row fetched from a cursor. For an example, see ["Fetching Across Commits"](#) on page 6-39.

ROWNUM

`ROWNUM` returns a number indicating the order in which a row was selected from a table. The first row selected has a `ROWNUM` of 1, the second row has a `ROWNUM` of 2, and

so on. If a `SELECT` statement includes an `ORDER BY` clause, `ROWNUMs` are assigned to the retrieved rows before the sort is done; use a subselect to get the first n sorted rows. The value of `ROWNUM` increases only when a row is retrieved, so the only meaningful uses of `ROWNUM` in a `WHERE` clause are:

```
... WHERE ROWNUM < constant;
... WHERE ROWNUM <= constant;
```

You can use `ROWNUM` in an `UPDATE` statement to assign unique values to each row in a table, or in the `WHERE` clause of a `SELECT` statement to limit the number of rows retrieved, as shown in [Example 6-6](#).

Example 6-6 Using ROWNUM

```
CREATE TABLE employees_temp AS SELECT * FROM employees;
DECLARE
  CURSOR c1 IS SELECT employee_id, salary FROM employees_temp
    WHERE salary > 2000 AND ROWNUM <= 10; -- 10 arbitrary rows
  CURSOR c2 IS SELECT * FROM
    (SELECT employee_id, salary FROM employees_temp
      WHERE salary > 2000 ORDER BY salary DESC)
    WHERE ROWNUM < 5; -- first 5 rows, in sorted order
BEGIN
  -- Each row gets assigned a different number
  UPDATE employees_temp SET employee_id = ROWNUM;
END;
/
```

SQL Operators

PL/SQL lets you use all the SQL comparison, set, and row operators in SQL statements. This section briefly describes some of these operators. For more information, see *Oracle Database SQL Language Reference*.

Topics:

- [Comparison Operators](#)
- [Set Operators](#)
- [Row Operators](#)

Comparison Operators

Typically, you use comparison operators in the `WHERE` clause of a data manipulation statement to form predicates, which compare one expression to another and yield `TRUE`, `FALSE`, or `NULL`. You can use the comparison operators in the following list to form predicates. You can combine predicates using the logical operators `AND`, `OR`, and `NOT`.

Operator	Description
<code>ALL</code>	Compares a value to each value in a list or returned by a subquery and yields <code>TRUE</code> if all of the individual comparisons yield <code>TRUE</code> .
<code>ANY, SOME</code>	Compares a value to each value in a list or returned by a subquery and yields <code>TRUE</code> if any of the individual comparisons yields <code>TRUE</code> .
<code>BETWEEN</code>	Tests whether a value lies in a specified range.
<code>EXISTS</code>	Returns <code>TRUE</code> if a subquery returns at least one row.

Operator	Description
IN	Tests for set membership.
IS NULL	Tests for nulls.
LIKE	Tests whether a character string matches a specified pattern, which can include wildcards.

Set Operators

Set operators combine the results of two queries into one result. `INTERSECT` returns all distinct rows selected by both queries. `MINUS` returns all distinct rows selected by the first query but not by the second. `UNION` returns all distinct rows selected by either query. `UNION ALL` returns all rows selected by either query, including all duplicates.

Row Operators

Row operators return or reference particular rows. `ALL` retains duplicate rows in the result of a query or in an aggregate expression. `DISTINCT` eliminates duplicate rows from the result of a query or from an aggregate expression. `PRIOR` refers to the parent row of the current row returned by a tree-structured query.

Managing Cursors in PL/SQL

PL/SQL uses implicit and explicit cursors. PL/SQL declares a cursor implicitly for all SQL data manipulation statements, including queries that return only one row. If you want precise control over query processing, you can declare an explicit cursor in the declarative part of any PL/SQL block, subprogram, or package. You must declare an explicit cursor for queries that return more than one row.

Topics:

- [Implicit Cursors](#)
- [Explicit Cursors](#)

Implicit Cursors

Implicit cursors are managed automatically by PL/SQL so you are not required to write any code to handle these cursors. However, you can track information about the execution of an implicit cursor through its cursor attributes.

Topics:

- [Attributes of Implicit Cursors](#)
- [Guidelines for Using Attributes of Implicit Cursors](#)

Attributes of Implicit Cursors

Implicit cursor attributes return information about the execution of DML and DDL statements, such as `INSERT`, `UPDATE`, `DELETE`, `SELECT INTO`, `COMMIT`, or `ROLLBACK` statements. The cursor attributes are `%FOUND`, `%ISOPEN`, `%NOTFOUND`, and `%ROWCOUNT`. The values of the cursor attributes always refer to the most recently executed SQL statement. Before Oracle opens the SQL cursor, the implicit cursor attributes yield `NULL`.

The SQL cursor has another attribute, `%BULK_ROWCOUNT`, designed for use with the `FORALL` statement. For more information, see "[Counting Rows Affected by FORALL \(%BULK_ROWCOUNT Attribute\)](#)" on page 12-16.

Topics:

- [%FOUND Attribute: Has a DML Statement Changed Rows?](#)
- [%ISOPEN Attribute: Always FALSE for Implicit Cursors](#)
- [%NOTFOUND Attribute: Has a DML Statement Failed to Change Rows?](#)
- [%ROWCOUNT Attribute: How Many Rows Affected So Far?](#)

%FOUND Attribute: Has a DML Statement Changed Rows? Until a SQL data manipulation statement is executed, %FOUND yields NULL. Thereafter, %FOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected one or more rows, or a SELECT INTO statement returned one or more rows. Otherwise, %FOUND yields FALSE. In [Example 6-7](#), you use %FOUND to insert a row if a delete succeeds.

Example 6-7 Using SQL%FOUND

```
CREATE TABLE dept_temp AS SELECT * FROM departments;
DECLARE
    dept_no NUMBER(4) := 270;
BEGIN
    DELETE FROM dept_temp WHERE department_id = dept_no;
    IF SQL%FOUND THEN -- delete succeeded
        INSERT INTO dept_temp VALUES (270, 'Personnel', 200, 1700);
    END IF;
END;
/
```

%ISOPEN Attribute: Always FALSE for Implicit Cursors Oracle closes the SQL cursor automatically after executing its associated SQL statement. As a result, %ISOPEN always yields FALSE.

%NOTFOUND Attribute: Has a DML Statement Failed to Change Rows? %NOTFOUND is the logical opposite of %FOUND. %NOTFOUND yields TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, %NOTFOUND yields FALSE.

%ROWCOUNT Attribute: How Many Rows Affected So Far? %ROWCOUNT yields the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement. %ROWCOUNT yields 0 if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. In [Example 6-8](#), %ROWCOUNT returns the number of rows that were deleted.

Example 6-8 Using SQL%ROWCOUNT

```
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;
/
```

If a SELECT INTO statement returns more than one row, PL/SQL raises the predefined exception TOO_MANY_ROWS and %ROWCOUNT yields 1, not the actual number of rows that satisfy the query.

The value of the `SQL%ROWCOUNT` attribute refers to the most recently executed SQL statement from PL/SQL. To save an attribute value for later use, assign it to a local variable immediately.

The `SQL%ROWCOUNT` attribute is not related to the state of a transaction. When a rollback to a savepoint is performed, the value of `SQL%ROWCOUNT` is not restored to the old value before the savepoint was taken. Also, when an autonomous transaction is exited, `SQL%ROWCOUNT` is not restored to the original value in the parent transaction.

Guidelines for Using Attributes of Implicit Cursors

The following are considerations when using attributes of implicit cursors:

- The values of the cursor attributes always refer to the most recently executed SQL statement, wherever that statement is. It might be in a different scope (for example, in a sub-block). To save an attribute value for later use, assign it to a local variable immediately. Doing other operations, such as subprogram calls, might change the value of the variable before you can test it.
- The `%NOTFOUND` attribute is not useful in combination with the `SELECT INTO` statement:
 - If a `SELECT INTO` statement fails to return a row, PL/SQL raises the predefined exception `NO_DATA_FOUND` immediately, interrupting the flow of control before you can check `%NOTFOUND`.
 - A `SELECT INTO` statement that invokes a SQL aggregate function always returns a value or a null. After such a statement, the `%NOTFOUND` attribute is always `FALSE`, so checking it is unnecessary.

Explicit Cursors

When you need precise control over query processing, you can explicitly declare a cursor in the declarative part of any PL/SQL block, subprogram, or package.

You use three statements to control a cursor: `OPEN`, `FETCH`, and `CLOSE`. First, you initialize the cursor with the `OPEN` statement, which identifies the result set. Then, you can execute `FETCH` repeatedly until all rows have been retrieved, or you can use the `BULK COLLECT` clause to fetch all rows at once. When the last row has been processed, you release the cursor with the `CLOSE` statement.

This technique requires more code than other techniques such as the implicit cursor `FOR` loop. Its advantage is flexibility. You can:

- Process several queries in parallel by declaring and opening multiple cursors.
- Process multiple rows in a single loop iteration, skip rows, or split the processing into more than one loop.

Topics:

- [Declaring a Cursor](#)
- [Opening a Cursor](#)
- [Fetching with a Cursor](#)
- [Fetching Bulk Data with a Cursor](#)
- [Closing a Cursor](#)
- [Attributes of Explicit Cursors](#)

Declaring a Cursor

You must declare a cursor before referencing it in other statements. You give the cursor a name and associate it with a specific query. You can optionally declare a return type for the cursor, such as *table_name%ROWTYPE*. You can optionally specify parameters that you use in the `WHERE` clause instead of referring to local variables. These parameters can have default values. [Example 6–9](#) shows how you can declare cursors.

Example 6–9 Declaring a Cursor

```
DECLARE
  my_emp_id    NUMBER(6);      -- variable for employee_id
  my_job_id    VARCHAR2(10);  -- variable for job_id
  my_sal       NUMBER(8,2);   -- variable for salary
  CURSOR c1 IS SELECT employee_id, job_id, salary FROM employees
                WHERE salary > 2000;
  my_dept      departments%ROWTYPE; -- variable for departments row
  CURSOR c2 RETURN departments%ROWTYPE IS
                SELECT * FROM departments WHERE department_id = 110;
```

The cursor is not a PL/SQL variable: you cannot assign values to a cursor or use it in an expression. Cursors and variables follow the same scoping rules. Naming cursors after database tables is possible but not recommended.

A cursor can take parameters, which can appear in the associated query wherever constants can appear. The formal parameters of a cursor must be `IN` parameters; they supply values in the query, but do not return any values from the query. You cannot impose the constraint `NOT NULL` on a cursor parameter.

As the following example shows, you can initialize cursor parameters to default values. You can pass different numbers of actual parameters to a cursor, accepting or overriding the default values as you please. Also, you can add new formal parameters without having to change existing references to the cursor.

```
DECLARE
  CURSOR c1 (low NUMBER DEFAULT 0, high NUMBER DEFAULT 99) IS
    SELECT * FROM departments WHERE department_id > low
    AND department_id < high;
```

Cursor parameters can be referenced only within the query specified in the cursor declaration. The parameter values are used by the associated query when the cursor is opened.

Opening a Cursor

Opening the cursor executes the query and identifies the result set, which consists of all rows that meet the query search criteria. For cursors declared using the `FOR UPDATE` clause, the `OPEN` statement also locks those rows. An example of the `OPEN` statement follows:

```
DECLARE
  CURSOR c1 IS
    SELECT employee_id, last_name, job_id, salary
    FROM employees
    WHERE salary > 2000;
BEGIN
  OPEN c1;
```

Rows in the result set are retrieved by the `FETCH` statement, not when the `OPEN` statement is executed.

Fetching with a Cursor

Unless you use the `BULK COLLECT` clause, explained in "Fetching with a Cursor" on page 6-11, the `FETCH` statement retrieves the rows in the result set one at a time. Each fetch retrieves the current row and advances the cursor to the next row in the result set. You can store each column in a separate variable, or store the entire row in a record that has the appropriate fields, usually declared using `%ROWTYPE`.

For each column value returned by the query associated with the cursor, there must be a corresponding, type-compatible variable in the `INTO` list. Typically, you use the `FETCH` statement with a `LOOP` and `EXIT WHEN NOTFOUND` statements, as shown in [Example 6-10](#). Note the use of built-in regular expression functions in the queries.

Example 6-10 Fetching with a Cursor

```
DECLARE
  v_jobid      employees.job_id%TYPE;      -- variable for job_id
  v_lastname   employees.last_name%TYPE;   -- variable for last_name
  CURSOR c1 IS SELECT last_name, job_id FROM employees
                WHERE REGEXP_LIKE (job_id, 'S[HT]_CLERK');
  v_employees  employees%ROWTYPE;         -- record variable for row
  CURSOR c2 IS SELECT * FROM employees
                WHERE REGEXP_LIKE (job_id, '[ACADFIMKSA]_M[ANGR]');
BEGIN
  OPEN c1; -- open the cursor before fetching
  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;
/
```

The query can reference PL/SQL variables within its scope. Any variables in the query are evaluated only when the cursor is opened. In [Example 6-11](#), each retrieved salary is multiplied by 2, even though `factor` is incremented after every fetch.

Example 6-11 Referencing PL/SQL Variables Within Its Scope

```
DECLARE
  my_sal employees.salary%TYPE;
  my_job employees.job_id%TYPE;
  factor INTEGER := 2;
  CURSOR c1 IS
    SELECT factor*salary FROM employees WHERE job_id = my_job;
BEGIN
  OPEN c1; -- factor initially equals 2
  LOOP
    FETCH c1 INTO my_sal;
```

```

        EXIT WHEN c1%NOTFOUND;
        factor := factor + 1; -- does not affect FETCH
    END LOOP;
    CLOSE c1;
END;
/

```

To change the result set or the values of variables in the query, you must close and reopen the cursor with the input variables set to their new values. However, you can use a different INTO list on separate fetches with the same cursor. Each fetch retrieves another row and assigns values to the target variables, as shown in [Example 6-12](#).

Example 6-12 Fetching the Same Cursor Into Different Variables

```

DECLARE
    CURSOR c1 IS SELECT last_name FROM employees ORDER BY last_name;
    name1 employees.last_name%TYPE;
    name2 employees.last_name%TYPE;
    name3 employees.last_name%TYPE;
BEGIN
    OPEN c1;
    FETCH c1 INTO name1; -- this fetches first row
    FETCH c1 INTO name2; -- this fetches second row
    FETCH c1 INTO name3; -- this fetches third row
    CLOSE c1;
END;
/

```

If you fetch past the last row in the result set, the values of the target variables are undefined. Eventually, the FETCH statement fails to return a row. When that happens, no exception is raised. To detect the failure, use the cursor attribute %FOUND or %NOTFOUND. For more information, see ["Using Cursor Expressions"](#) on page 6-31.

Fetching Bulk Data with a Cursor

The BULK COLLECT clause lets you fetch all rows from the result set at once. See ["Retrieving Query Results into Collections \(BULK COLLECT Clause\)"](#) on page 12-19. In [Example 6-13](#), you bulk-fetch from a cursor into two collections.

Example 6-13 Fetching Bulk Data with a Cursor

```

DECLARE
    TYPE IdsTab IS TABLE OF employees.employee_id%TYPE;
    TYPE NameTab IS TABLE OF employees.last_name%TYPE;
    ids IdsTab;
    names NameTab;
    CURSOR c1 IS
        SELECT employee_id, last_name;
        FROM employees
        WHERE job_id = 'ST_CLERK';
BEGIN
    OPEN c1;
    FETCH c1 BULK COLLECT INTO ids, names;
    CLOSE c1;
    -- Here is where you process the elements in the collections
    FOR i IN ids.FIRST .. ids.LAST
    LOOP
        IF ids(i) > 140 THEN
            DBMS_OUTPUT.PUT_LINE( ids(i) );
        END IF;
    END LOOP;
END;

```



```

        END LOOP;
    FOR i IN names.FIRST .. names.LAST
    LOOP
        IF names(i) LIKE '%Ma%' THEN
            DBMS_OUTPUT.PUT_LINE( names(i) );
        END IF;
    END LOOP;
END;
/

```

Closing a Cursor

The `CLOSE` statement disables the cursor, and the result set becomes undefined. Once a cursor is closed, you can reopen it, which runs the query again with the latest values of any cursor parameters and variables referenced in the `WHERE` clause. Any other operation on a closed cursor raises the predefined exception `INVALID_CURSOR`.

Attributes of Explicit Cursors

Every explicit cursor and cursor variable has four attributes: `%FOUND`, `%ISOPEN`, `%NOTFOUND`, and `%ROWCOUNT`. When appended to the cursor or cursor variable name, these attributes return useful information about the execution of a SQL statement. You can use cursor attributes in procedural statements but not in SQL statements.

Explicit cursor attributes return information about the execution of a multiple-row query. When an explicit cursor or a cursor variable is opened, the rows that satisfy the associated query are identified and form the result set. Rows are fetched from the result set.

Topics:

- [%FOUND Attribute: Has a Row Been Fetched?](#)
- [%ISOPEN Attribute: Is the Cursor Open?](#)
- [%NOTFOUND Attribute: Has a Fetch Failed?](#)
- [%ROWCOUNT Attribute: How Many Rows Fetched So Far?](#)

%FOUND Attribute: Has a Row Been Fetched? After a cursor or cursor variable is opened but before the first fetch, `%FOUND` returns `NULL`. After any fetches, it returns `TRUE` if the last fetch returned a row, or `FALSE` if the last fetch did not return a row.

[Example 6-14](#) uses `%FOUND` to select an action.

Example 6-14 Using %FOUND

```

DECLARE
    CURSOR c1 IS SELECT last_name, salary FROM employees WHERE ROWNUM < 11;
    my_ename employees.last_name%TYPE;
    my_salary employees.salary%TYPE;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO my_ename, my_salary;
        IF c1%FOUND THEN -- fetch succeeded
            DBMS_OUTPUT.PUT_LINE('Name = ' || my_ename || ', salary = ' || my_salary);
        ELSE -- fetch failed, so exit loop
            EXIT;
        END IF;
    END LOOP;
END;
/

```

If a cursor or cursor variable is not open, referencing it with `%FOUND` raises the predefined exception `INVALID_CURSOR`.

%ISOPEN Attribute: Is the Cursor Open? `%ISOPEN` returns `TRUE` if its cursor or cursor variable is open; otherwise, `%ISOPEN` returns `FALSE`. [Example 6–15](#) uses `%ISOPEN` to select an action.

Example 6–15 Using %ISOPEN

```
DECLARE
    CURSOR c1 IS
        SELECT last_name, salary
        FROM employees WHERE ROWNUM < 11;
    the_name employees.last_name%TYPE;
    the_salary employees.salary%TYPE;
BEGIN
    IF c1%ISOPEN = FALSE THEN -- cursor was not already open
        OPEN c1;
    END IF;
    FETCH c1 INTO the_name, the_salary;
    CLOSE c1;
END;
/
```

%NOTFOUND Attribute: Has a Fetch Failed? `%NOTFOUND` is the logical opposite of `%FOUND`. `%NOTFOUND` yields `FALSE` if the last fetch returned a row, or `TRUE` if the last fetch failed to return a row. In [Example 6–16](#), you use `%NOTFOUND` to exit a loop when `FETCH` fails to return a row.

Example 6–16 Using %NOTFOUND

```
DECLARE
    CURSOR c1 IS SELECT last_name, salary
        FROM employees
        WHERE ROWNUM < 11;
    my_ename employees.last_name%TYPE;
    my_salary employees.salary%TYPE;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO my_ename, my_salary;
        IF c1%NOTFOUND THEN -- fetch failed, so exit loop
-- Another form of this test is
-- "EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;"
            EXIT;
        ELSE -- fetch succeeded
            DBMS_OUTPUT.PUT_LINE
                ('Name = ' || my_ename || ', salary = ' || my_salary);
        END IF;
    END LOOP;
END;
/
```

Before the first fetch, `%NOTFOUND` returns `NULL`. If `FETCH` never executes successfully, the loop is never exited, because the `EXIT WHEN` statement executes only if its `WHEN` condition is true. To be safe, you might want to use the following `EXIT` statement instead:

```
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
```

If a cursor or cursor variable is not open, referencing it with %NOTFOUND raises an INVALID_CURSOR exception.

%ROWCOUNT Attribute: How Many Rows Fetched So Far? When its cursor or cursor variable is opened, %ROWCOUNT is zeroed. Before the first fetch, %ROWCOUNT yields zero. Thereafter, it yields the number of rows fetched so far. The number is incremented if the last fetch returned a row. [Example 6-17](#) uses %ROWCOUNT to test if more than ten rows were fetched.

Example 6-17 Using %ROWCOUNT

```

DECLARE
    CURSOR c1 IS SELECT last_name FROM employees WHERE ROWNUM < 11;
    name employees.last_name%TYPE;
BEGIN
    OPEN c1;
    LOOP
        FETCH c1 INTO name;
        EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
        DBMS_OUTPUT.PUT_LINE(c1%ROWCOUNT || ' ' || name);
        IF c1%ROWCOUNT = 5 THEN
            DBMS_OUTPUT.PUT_LINE('--- Fetched 5th record ---');
        END IF;
    END LOOP;
    CLOSE c1;
END;
/
    
```

If a cursor or cursor variable is not open, referencing it with %ROWCOUNT raises INVALID_CURSOR.

[Table 6-1](#) shows the value of each cursor attribute before and after OPEN, FETCH, and CLOSE statements execute.

Table 6-1 Cursor Attribute Values

Point in Time	%FOUND Value	%ISOPEN Value	%NOTFOUND Value	%ROWCOUNT Value
Before OPEN	exception	FALSE	exception	exception
After OPEN	NULL	TRUE	NULL	0
Before first FETCH	NULL	TRUE	NULL	0
After first FETCH	TRUE	TRUE	FALSE	1
Before each successive FETCH except last	TRUE	TRUE	FALSE	1
After each successive FETCH except last	TRUE	TRUE	FALSE	data dependent
Before last FETCH	TRUE	TRUE	FALSE	data dependent
After last FETCH	FALSE	TRUE	TRUE	data dependent
Before CLOSE	FALSE	TRUE	TRUE	data dependent
After CLOSE	exception	FALSE	exception	exception

The following applies to the information in [Table 6-1](#):

- Referencing `%FOUND`, `%NOTFOUND`, or `%ROWCOUNT` before a cursor is opened or after it is closed raises `INVALID_CURSOR`.
- After the first `FETCH`, if the result set was empty, `%FOUND` yields `FALSE`, `%NOTFOUND` yields `TRUE`, and `%ROWCOUNT` yields 0.

Querying Data with PL/SQL

PL/SQL lets you perform queries (`SELECT` statements in SQL) and access individual fields or entire rows from the result set. In traditional database programming, you process query results using an internal data structure called a cursor. In most situations, PL/SQL can manage the cursor for you, so that code to process query results is straightforward and compact. This section explains how to process both simple queries where PL/SQL manages everything, and complex queries where you interact with the cursor.

Topics:

- [Selecting At Most One Row \(SELECT INTO Statement\)](#)
- [Selecting Multiple Rows \(BULK COLLECT Clause\)](#)
- [Looping Through Multiple Rows \(Cursor FOR Loop\)](#)
- [Performing Complicated Query Processing \(Explicit Cursors\)](#)
- [Cursor FOR LOOP](#)
- [Defining Aliases for Expression Values in a Cursor FOR Loop](#)

Selecting At Most One Row (SELECT INTO Statement)

If you expect a query to only return one row, you can write a regular SQL `SELECT` statement with an additional `INTO` clause specifying the PL/SQL variable to hold the result.

If the query might return more than one row, but you do not care about values after the first, you can restrict any result set to a single row by comparing the `ROWNUM` value. If the query might return no rows at all, use an exception handler to specify any actions to take when no data is found.

If you just want to check whether a condition exists in your data, you might be able to code the query with the `COUNT (*)` operator, which always returns a number and never raises the `NO_DATA_FOUND` exception.

Selecting Multiple Rows (BULK COLLECT Clause)

If you need to bring a large quantity of data into local PL/SQL variables, rather than looping through a result set one row at a time, you can use the `BULK COLLECT` clause. When you query only certain columns, you can store all the results for each column in a separate collection variable. When you query all the columns of a table, you can store the entire result set in a collection of records, which makes it convenient to loop through the results and refer to different columns. See [Example 6-13, "Fetching Bulk Data with a Cursor"](#) on page 6-12.

This technique can be very fast, but also very memory-intensive. If you use it often, you might be able to improve your code by doing more of the work in SQL:

- If you only need to loop once through the result set, use a `FOR` loop as described in the following sections. This technique avoids the memory overhead of storing a copy of the result set.

- If you are looping through the result set to scan for certain values or filter the results into a smaller set, do this scanning or filtering in the original query instead. You can add more `WHERE` clauses in simple cases, or use set operators such as `INTERSECT` and `MINUS` if you are comparing two or more sets of results.
- If you are looping through the result set and running another query or a DML statement for each result row, you can probably find a more efficient technique. For queries, look at including subqueries or `EXISTS` or `NOT EXISTS` clauses in the original query. For DML statements, look at the `FORALL` statement, which is much faster than coding these statements inside a regular loop.

Looping Through Multiple Rows (Cursor FOR Loop)

Perhaps the most common case of a query is one where you issue the `SELECT` statement, then immediately loop once through the rows of the result set. PL/SQL lets you use a simple `FOR` loop for this kind of query:

The iterator variable for the `FOR` loop does not need to be declared in advance. It is a `%ROWTYPE` record whose field names match the column names from the query, and that exists only during the loop. When you use expressions rather than explicit column names, use column aliases so that you can refer to the corresponding values inside the loop.

Performing Complicated Query Processing (Explicit Cursors)

For full control over query processing, you can use explicit cursors in combination with the `OPEN`, `FETCH`, and `CLOSE` statements.

You might want to specify a query in one place but retrieve the rows somewhere else, even in another subprogram. Or you might want to choose very different query parameters, such as `ORDER BY` or `GROUP BY` clauses, depending on the situation. Or you might want to process some rows differently than others, and so need more than a simple loop.

Because explicit cursors are so flexible, you can choose from different notations depending on your needs. The following sections describe all the query-processing features that explicit cursors provide.

Cursor FOR LOOP

Topics:

- [Implicit Cursor FOR LOOP](#)
- [Explicit Cursor FOR LOOP](#)

Implicit Cursor FOR LOOP

With PL/SQL, it is very simple to issue a query, retrieve each row of the result into a `%ROWTYPE` record, and process each row in a loop:

- You include the text of the query directly in the `FOR` loop.
- PL/SQL creates a record variable with fields corresponding to the columns of the result set.
- You refer to the fields of this record variable inside the loop. You can perform tests and calculations, display output, or store the results somewhere else.

Here is an example that you can run in SQL*Plus. It does a query to get the name and job Id of employees with manager Ids greater than 120.

```

BEGIN
  FOR item IN
    ( SELECT last_name, job_id
      FROM employees
      WHERE job_id LIKE '%CLERK%'
      AND manager_id > 120 )
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/

```

Before each iteration of the FOR loop, PL/SQL fetches into the implicitly declared record. The sequence of statements inside the loop is executed once for each row that satisfies the query. When you leave the loop, the cursor is closed automatically. The cursor is closed even if you use an EXIT or GOTO statement to leave the loop before all rows are fetched, or an exception is raised inside the loop. See ["LOOP Statements"](#) on page 13-93.

Explicit Cursor FOR LOOP

If you need to reference the same query from different parts of the same subprogram, you can declare a cursor that specifies the query, and process the results using a FOR loop.

```

DECLARE
  CURSOR c1 IS SELECT last_name, job_id FROM employees
                WHERE job_id LIKE '%CLERK%' AND manager_id > 120;
BEGIN
  FOR item IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || item.last_name || ', Job = ' || item.job_id);
  END LOOP;
END;
/

```

Tip: ["LOOP Statements"](#) on page 13-93

Defining Aliases for Expression Values in a Cursor FOR Loop

In a cursor FOR loop, PL/SQL creates a %ROWTYPE record with fields corresponding to columns in the result set. The fields have the same names as corresponding columns in the SELECT list.

The select list might contain an expression, such as a column plus a constant, or two columns concatenated together. If so, use a column alias to give unique names to the appropriate columns.

In [Example 6-18](#), full_name and dream_salary are aliases for expressions in the query:

Example 6-18 Using an Alias For Expressions in a Query

```

BEGIN
  FOR item IN
    ( SELECT first_name || ' ' || last_name AS full_name,
      salary * 10 AS dream_salary FROM employees WHERE ROWNUM <= 5 )

```

```

LOOP
  DBMS_OUTPUT.PUT_LINE
    (item.full_name || ' dreams of making ' || item.dream_salary);
END LOOP;
END;
/

```

Using Subqueries

A subquery is a query (usually enclosed by parentheses) that appears within another SQL data manipulation statement. The statement acts upon the single value or set of values returned by the subquery. For example:

- You can use a subquery to find the MAX, MIN, or AVG value for a column, and use that single value in a comparison in a WHERE clause.
- You can use a subquery to find a set of values, and use this values in an IN or NOT IN comparison in a WHERE clause. This technique can avoid joins.
- You can filter a set of values with a subquery, and apply other operations like ORDER BY and GROUP BY in the outer query.
- You can use a subquery in place of a table name, in the FROM clause of a query. This technique lets you join a table with a small set of rows from another table, instead of joining the entire tables.
- You can create a table or insert into a table, using a set of rows defined by a subquery.

[Example 6–19](#) illustrates two subqueries used in cursor declarations.

Example 6–19 *Using a Subquery in a Cursor*

```

DECLARE
  CURSOR c1 IS
  -- main query returns only rows
  -- where the salary is greater than the average
  SELECT employee_id, last_name FROM employees
     WHERE salary > (SELECT AVG(salary) FROM employees);
  CURSOR c2 IS
  -- subquery returns all the rows in descending order of salary
  -- main query returns just the top 10 highest-paid employees
  SELECT * FROM
    (SELECT last_name, salary)
    FROM employees ORDER BY salary DESC, last_name)
    ORDER BY salary DESC, last_name)
  WHERE ROWNUM < 11;
BEGIN
  FOR person IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Above-average salary: ' || person.last_name);
  END LOOP;
  FOR person IN c2
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Highest paid: ' || person.last_name || ' $' || person.salary);
  END LOOP;
  -- subquery identifies a set of rows
  -- to use with CREATE TABLE or INSERT
END;

```

/

Using a subquery in the FROM clause, the query in [Example 6–20](#) returns the number and name of each department with five or more employees.

Example 6–20 Using a Subquery in a FROM Clause

```
DECLARE
  CURSOR c1 IS
    SELECT t1.department_id, department_name, staff
      FROM departments t1,
      ( SELECT department_id, COUNT(*) as staff
        FROM employees GROUP BY department_id) t2
    WHERE
      t1.department_id = t2.department_id
      AND staff >= 5;
BEGIN
  FOR dept IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE ('Department = '
      || dept.department_name || ', staff = ' || dept.staff);
  END LOOP;
END;
```

/

Topics:

- [Using Correlated Subqueries](#)
- [Writing Maintainable PL/SQL Subqueries](#)

Using Correlated Subqueries

While a subquery is evaluated only once for each table, a correlated subquery is evaluated once for each row. [Example 6–21](#) 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 6–21 Using a Correlated Subquery

```
DECLARE
-- For each department, find the average salary.
-- Then find all the employees in
-- that department making more than that average salary.
  CURSOR c1 IS
    SELECT department_id, last_name, salary FROM employees t
      WHERE salary >
        ( SELECT AVG(salary)
          FROM employees
          WHERE t.department_id = department_id )
      ORDER BY department_id;
BEGIN
  FOR person IN c1
  LOOP
    DBMS_OUTPUT.PUT_LINE('Making above-average salary = ' || person.last_name);
  END LOOP;
END;
```

/

Writing Maintainable PL/SQL Subqueries

Instead of referring to local variables, you can declare a cursor that accepts parameters, and pass values for those parameters when you open the cursor. If the query is usually issued with certain values, you can make those values the defaults. You can use either positional notation or named notation to pass the parameter values.

[Example 6–22](#) displays the wages paid to employees earning over a specified wage in a specified department.

Example 6–22 *Passing Parameters to a Cursor FOR Loop*

```
DECLARE
  CURSOR c1 (job VARCHAR2, max_wage NUMBER) IS
    SELECT * FROM employees
      WHERE job_id = job
      AND salary > max_wage;
BEGIN
  FOR person IN c1('CLERK', 3000)
  LOOP
    -- process data record
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || person.last_name || ', salary = ' ||
       person.salary || ', Job Id = ' || person.job_id );
  END LOOP;
END;
/
```

In [Example 6–23](#), several ways are shown to open a cursor.

Example 6–23 *Passing Parameters to Explicit Cursors*

```
DECLARE
  emp_job      employees.job_id%TYPE := 'ST_CLERK';
  emp_salary   employees.salary%TYPE := 3000;
  my_record    employees%ROWTYPE;
  CURSOR c1 (job VARCHAR2, max_wage NUMBER) IS
    SELECT * FROM employees
      WHERE job_id = job
      AND salary > max_wage;
BEGIN
  -- Any of the following statements opens the cursor:
  -- OPEN c1('ST_CLERK', 3000); OPEN c1('ST_CLERK', emp_salary);
  -- OPEN c1(emp_job, 3000); OPEN c1(emp_job, emp_salary);
  OPEN c1(emp_job, emp_salary);
  LOOP
    FETCH c1 INTO my_record;
    EXIT WHEN c1%NOTFOUND;
    -- process data record
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || my_record.last_name || ', salary = ' ||
       my_record.salary || ', Job Id = ' || my_record.job_id );
  END LOOP;
END;
/
```

To avoid confusion, use different names for cursor parameters and the PL/SQL variables that you pass into those parameters.

A formal parameter declared with a default value does not need a corresponding actual parameter. If you omit the actual parameter, the formal parameter assumes its

default value when the `OPEN` statement executes. If the default value of a formal parameter is an expression, and you provide a corresponding actual parameter in the `OPEN` statement, the expression is not evaluated.

Using Cursor Variables (REF CURSORS)

Like a cursor, a cursor variable points to the current row in the result set of a multiple-row query. A cursor variable is more flexible because it is not tied to a specific query. You can open a cursor variable for any query that returns the right set of columns.

You pass a cursor variable as a parameter to local and stored subprograms. Opening the cursor variable in one subprogram, and processing it in a different subprogram, helps to centralize data retrieval. This technique is also useful for multi-language applications, where a PL/SQL subprogram might return a result set to a subprogram written in a different language, such as Java or Visual Basic.

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, then pass it as an input host variable (bind variable) to PL/SQL. Application development tools such as Oracle Forms, which have a PL/SQL engine, can use cursor variables entirely on the client side. Or, you can pass cursor variables back and forth between a client and the database server through remote subprogram calls.

Topics:

- [What Are Cursor Variables \(REF CURSORS\)?](#)
- [Why Use Cursor Variables?](#)
- [Declaring REF CURSOR Types and Cursor Variables](#)
- [Passing Cursor Variables As Parameters](#)
- [Controlling Cursor Variables \(OPEN-FOR, FETCH, and CLOSE Statements\)](#)
- [Reducing Network Traffic When Passing Host Cursor Variables to PL/SQL](#)
- [Avoiding Errors with Cursor Variables](#)
- [Restrictions on Cursor Variables](#)

What Are Cursor Variables (REF CURSORS)?

Cursor variables are like pointers to result sets. You use them when you want to perform a query in one subprogram, and process the results in a different subprogram (possibly one written in a different language). A cursor variable has datatype `REF CURSOR`, and you might see them referred to informally as `REF CURSORS`.

Unlike an explicit cursor, which always refers to the same query work area, a cursor variable can refer to different work areas. You cannot use a cursor variable where a cursor is expected, or vice versa.

Why Use Cursor Variables?

You use cursor variables to pass query result sets between PL/SQL stored subprograms and various clients. PL/SQL and its clients share a pointer to the query work area in which the result set is stored. For example, an OCI client, Oracle Forms application, and Oracle database server can all refer to the same work area.

A query work area remains accessible as long as any cursor variable points to it, as you pass the value of a cursor variable from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block embedded in a Pro*C program, the work area to which the cursor variable points remains accessible after the block completes.

If you have a PL/SQL engine on the client side, calls from client to server impose no restrictions. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, then continue to fetch from it back on the client side. You can also reduce network traffic by having a PL/SQL block open or close several host cursor variables in a single round trip.

Declaring REF CURSOR Types and Cursor Variables

To create cursor variables, you define a REF CURSOR type, then declare cursor variables of that type. You can define REF CURSOR types in any PL/SQL block, subprogram, or package. In the following example, you declare a REF CURSOR type that represents a result set from the DEPARTMENTS table:

```
DECLARE
  TYPE DeptCurTyp IS REF CURSOR RETURN departments%ROWTYPE
```

REF CURSOR types can be strong (with a return type) or weak (with no return type). Strong REF CURSOR types are less error prone because the PL/SQL compiler lets you associate a strongly typed cursor variable only with queries that return the right set of columns. Weak REF CURSOR types are more flexible because the compiler lets you associate a weakly typed cursor variable with any query. Because there is no type checking with a weak REF CURSOR, all such types are interchangeable. Instead of creating a new type, you can use the predefined type SYS_REFCURSOR.

Once you define a REF CURSOR type, you can declare cursor variables of that type in any PL/SQL block or subprogram.

```
DECLARE
  -- Strong:
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  -- Weak:
  TYPE genericcurtyp IS REF CURSOR;
  cursor1 empcurtyp;
  cursor2 genericcurtyp;
  my_cursor SYS_REFCURSOR; -- didn't need to declare a new type
  TYPE deptcurtyp IS REF CURSOR RETURN departments%ROWTYPE;
  dept_cv deptcurtyp; -- declare cursor variable
```

To avoid declaring the same REF CURSOR type in each subprogram that uses it, you can put the REF CURSOR declaration in a package spec. You can declare cursor variables of that type in the corresponding package body, or within your own subprogram.

In the RETURN clause of a REF CURSOR type definition, you can use %ROWTYPE to refer to a strongly typed cursor variable, as shown in [Example 6–24](#).

Example 6–24 Cursor Variable Returning a %ROWTYPE Variable

```
DECLARE
  TYPE TmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
  tmp_cv TmpCurTyp; -- declare cursor variable
  TYPE EmpCurTyp IS REF CURSOR RETURN tmp_cv%ROWTYPE;
  emp_cv EmpCurTyp; -- declare cursor variable
```

You can also use %ROWTYPE to provide the datatype of a record variable, as shown in [Example 6–25](#).

Example 6–25 Using the %ROWTYPE Attribute to Provide the Datatype

```
DECLARE
    dept_rec departments%ROWTYPE; -- declare record variable
    TYPE DeptCurTyp IS REF CURSOR RETURN dept_rec%TYPE;
    dept_cv DeptCurTyp; -- declare cursor variable
```

[Example 6–26](#) specifies a user-defined RECORD type in the RETURN clause:

Example 6–26 Cursor Variable Returning a Record Type

```
DECLARE
    TYPE EmpRecTyp IS RECORD (
        employee_id NUMBER,
        last_name VARCHAR2(25),
        salary NUMBER(8,2));
    TYPE EmpCurTyp IS REF CURSOR RETURN EmpRecTyp;
    emp_cv EmpCurTyp; -- declare cursor variable
```

Passing Cursor Variables As Parameters

You can declare cursor variables as the formal parameters of subprograms.

[Example 6–27](#) defines a REF CURSOR type, then declares a cursor variable of that type as a formal parameter.

Example 6–27 Passing a REF CURSOR as a Parameter

```
DECLARE
    TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
    emp empcurtyp;
-- after result set is built,
-- process all the rows inside a single procedure
-- rather than invoking a procedure for each row
PROCEDURE process_emp_cv (emp_cv IN empcurtyp) IS
    person employees%ROWTYPE;
BEGIN
    DBMS_OUTPUT.PUT_LINE('-----');
    DBMS_OUTPUT.PUT_LINE
        ('Here are the names from the result set:');
    LOOP
        FETCH emp_cv INTO person;
        EXIT WHEN emp_cv%NOTFOUND;
        DBMS_OUTPUT.PUT_LINE('Name = ' || person.first_name ||
            ' ' || person.last_name);
    END LOOP;
END;
BEGIN
-- First find 10 arbitrary employees.
OPEN emp FOR SELECT * FROM employees WHERE ROWNUM < 11;
process_emp_cv(emp);
CLOSE emp;
-- find employees matching a condition.
OPEN emp FOR SELECT * FROM employees WHERE last_name LIKE 'R%';
process_emp_cv(emp);
CLOSE emp;
END;
/
```

Like all pointers, cursor variables increase the possibility of parameter aliasing. See ["Overloading PL/SQL Subprogram Names"](#) on page 8-12.

Controlling Cursor Variables (OPEN-FOR, FETCH, and CLOSE Statements)

You use three statements to control a cursor variable: `OPEN-FOR`, `FETCH`, and `CLOSE`. First, you `OPEN` a cursor variable `FOR` a multiple-row query. Then, you `FETCH` rows from the result set. When all the rows are processed, you `CLOSE` the cursor variable.

Topics:

- [Opening a Cursor Variable](#)
- [Using a Cursor Variable as a Host Variable](#)
- [Fetching from a Cursor Variable](#)
- [Closing a Cursor Variable](#)

Opening a Cursor Variable

The `OPEN-FOR` statement associates a cursor variable with a multiple-row query, executes the query, and identifies the result set. The cursor variable can be declared directly in PL/SQL, or in a PL/SQL host environment such as an OCI program. For the syntax of the `OPEN-FOR` statement, see ["OPEN-FOR Statement"](#) on page 13-104.

The `SELECT` statement for the query can be coded directly in the statement, or can be a string variable or string literal. When you use a string as the query, it can include placeholders for bind variables, and you specify the corresponding values with a `USING` clause.

This section explains the static SQL case, in which `select_statement` is used. For the dynamic SQL case, in which `dynamic_string` is used, see ["OPEN-FOR Statement"](#) on page 13-104.

Unlike cursors, cursor variables take no parameters. Instead, you can pass whole queries (not just parameters) to a cursor variable. The query can reference host variables and PL/SQL variables, parameters, and functions.

[Example 6-28](#) opens a cursor variable. Notice that you can apply cursor attributes (`%FOUND`, `%NOTFOUND`, `%ISOPEN`, and `%ROWCOUNT`) to a cursor variable.

Example 6-28 *Checking If a Cursor Variable is Open*

```
DECLARE
    TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
    emp_cv empcurtyp;
BEGIN
    IF NOT emp_cv%ISOPEN THEN -- open cursor variable
        OPEN emp_cv FOR SELECT * FROM employees;
    END IF;
    CLOSE emp_cv;
END;
```

Other `OPEN-FOR` statements can open the same cursor variable for different queries. You need not close a cursor variable before reopening it. Note that consecutive `OPENS` of a static cursor raise the predefined exception `CURSOR_ALREADY_OPEN`. When you reopen a cursor variable for a different query, the previous query is lost.

Typically, you open a cursor variable by passing it to a stored subprogram that declares an IN OUT parameter that is a cursor variable. In [Example 6–29](#) the subprogram opens a cursor variable.

Example 6–29 Stored Procedure to Open a Ref Cursor

```
CREATE 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 PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (emp_cv IN OUT EmpCurTyp) IS
  BEGIN
    OPEN emp_cv FOR SELECT * FROM employees;
  END open_emp_cv;
END emp_data;
/
```

You can also use a standalone stored subprogram to open the cursor variable. Define the REF CURSOR type in a package, then reference that type in the parameter declaration for the stored subprogram.

To centralize data retrieval, you can group type-compatible queries in a stored subprogram. In [Example 6–30](#), the packaged subprogram declares a selector as one of its formal parameters. When invoked, the subprogram opens the cursor variable emp_cv for the chosen query.

Example 6–30 Stored Procedure to Open Ref Cursors with Different Queries

```
CREATE 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 PACKAGE BODY emp_data AS
  PROCEDURE open_emp_cv (emp_cv IN OUT empcurtyp, choice INT) IS
  BEGIN
    IF choice = 1 THEN
      OPEN emp_cv FOR SELECT *
        FROM employees
        WHERE commission_pct IS NOT NULL;
    ELSIF choice = 2 THEN
      OPEN emp_cv FOR SELECT *
        FROM employees
        WHERE salary > 2500;
    ELSIF choice = 3 THEN
      OPEN emp_cv FOR SELECT *
        FROM employees
        WHERE department_id = 100;
    END IF;
  END;
END emp_data;
/
```

For more flexibility, a stored subprogram can execute queries with different return types, shown in [Example 6–31](#).

Example 6-31 Cursor Variable with Different Return Types

```

CREATE PACKAGE admin_data AS
  TYPE gencurtyp IS REF CURSOR;
  PROCEDURE open_cv (generic_cv IN OUT gencurtyp, choice INT);
END admin_data;
/
CREATE 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;
/

```

Using a Cursor Variable as a Host Variable

You can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program. To use the cursor variable, you must pass it as a host variable to PL/SQL. In the following Pro*C example, you pass a host cursor variable and selector to a PL/SQL block, which opens the cursor variable for the chosen query.

```

EXEC SQL BEGIN DECLARE SECTION;
...
/* Declare host cursor variable. */
SQL_CURSOR generic_cv;
int choice;
EXEC SQL END DECLARE SECTION;
...
/* Initialize host cursor variable. */
EXEC SQL ALLOCATE :generic_cv;
...
/* Pass host cursor variable and selector to PL/SQL block. */
/
EXEC SQL EXECUTE
BEGIN
  IF :choice = 1 THEN
    OPEN :generic_cv FOR SELECT * FROM employees;
  ELSIF :choice = 2 THEN
    OPEN :generic_cv FOR SELECT * FROM departments;
  ELSIF :choice = 3 THEN
    OPEN :generic_cv FOR SELECT * FROM jobs;
  END IF;
END;
END-EXEC;

```

Host cursor variables are compatible with any query return type. They act just like weakly typed PL/SQL cursor variables.

Fetching from a Cursor Variable

The `FETCH` statement retrieves rows from the result set of a multiple-row query. It works the same with cursor variables as with explicit cursors. [Example 6–32](#) fetches rows one at a time from a cursor variable into a record.

Example 6–32 Fetching from a Cursor Variable into a Record

```
DECLARE
  TYPE empcurtyp IS REF CURSOR RETURN employees%ROWTYPE;
  emp_cv empcurtyp;
  emp_rec employees%ROWTYPE;
BEGIN
  OPEN emp_cv FOR SELECT * FROM employees WHERE employee_id < 120;
  LOOP
    FETCH emp_cv INTO emp_rec; -- fetch from cursor variable
    EXIT WHEN emp_cv%NOTFOUND; -- exit when last row is fetched
    -- process data record
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || emp_rec.first_name || ' ' || emp_rec.last_name);
  END LOOP;
  CLOSE emp_cv;
END;
/
```

Using the `BULK COLLECT` clause, you can bulk fetch rows from a cursor variable into one or more collections as shown in [Example 6–33](#).

Example 6–33 Fetching from a Cursor Variable into Collections

```
DECLARE
  TYPE empcurtyp IS REF CURSOR;
  TYPE namelist IS TABLE OF employees.last_name%TYPE;
  TYPE sallist IS TABLE OF employees.salary%TYPE;
  emp_cv empcurtyp;
  names namelist;
  sals sallist;
BEGIN
  OPEN emp_cv FOR SELECT last_name, salary FROM employees
    WHERE job_id = 'SA_REP';
  FETCH emp_cv BULK COLLECT INTO names, sals;
  CLOSE emp_cv;
  -- loop through the names and sals collections
  FOR i IN names.FIRST .. names.LAST
  LOOP
    DBMS_OUTPUT.PUT_LINE
      ('Name = ' || names(i) || ', salary = ' || sals(i));
  END LOOP;
END;
/
```

Any variables in the associated query are evaluated only when the cursor variable is opened. To change the result set or the values of variables in the query, reopen the cursor variable with the variables set to new values. You can use a different `INTO` clause on separate fetches with the same cursor variable. Each fetch retrieves another row from the same result set.

PL/SQL makes sure the return type of the cursor variable is compatible with the `INTO` clause of the `FETCH` statement. If there is a mismatch, an error occurs at compile time if the cursor variable is strongly typed, or at run time if it is weakly typed. At run time,

PL/SQL raises the predefined exception `ROWTYPE_MISMATCH` before the first fetch. If you trap the error and execute the `FETCH` statement using a different (compatible) `INTO` clause, no rows are lost.

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

If you try to fetch from a closed or never-opened cursor variable, PL/SQL raises the predefined exception `INVALID_CURSOR`.

Closing a Cursor Variable

The `CLOSE` statement disables a cursor variable and makes the associated result set undefined. Close the cursor variable after the last row is processed.

When declaring a cursor variable as the formal parameter of a subprogram that closes the cursor variable, you must specify the `IN` or `IN OUT` mode. If you try to close an already-closed or never-opened cursor variable, PL/SQL raises the predefined exception `INVALID_CURSOR`.

Reducing Network Traffic When Passing Host Cursor Variables to PL/SQL

When passing host cursor variables to PL/SQL, you can reduce network traffic by grouping `OPEN-FOR` statements. For example, the following PL/SQL block opens multiple cursor variables in a single round trip:

```
/* anonymous PL/SQL block in host environment */
BEGIN
  OPEN :emp_cv FOR SELECT * FROM employees;
  OPEN :dept_cv FOR SELECT * FROM departments;
  OPEN :loc_cv FOR SELECT * FROM locations;
END;
/
```

This technique might be useful in Oracle Forms, for example, when you want to populate a multi-block form. When you pass host cursor variables to a PL/SQL block for opening, the query work areas to which they point remain accessible after the block completes, so your OCI or Pro*C program can use these work areas for ordinary cursor operations. For example, you open several such work areas in a single round trip:

```
BEGIN
  OPEN :c1 FOR SELECT 1 FROM DUAL;
  OPEN :c2 FOR SELECT 1 FROM DUAL;
  OPEN :c3 FOR SELECT 1 FROM DUAL;
END;
/
```

The cursors assigned to `c1`, `c2`, and `c3` act normally, and you can use them for any purpose. When finished, release the cursors as follows:

```
BEGIN
  CLOSE :c1; CLOSE :c2; CLOSE :c3;
END;
/
```

Avoiding Errors with Cursor Variables

If both cursor variables involved in an assignment are strongly typed, they must have exactly the same datatype (not just the same return type). If one or both cursor variables are weakly typed, they can have different datatypes.

If you try to fetch from, close, or refer to cursor attributes of a cursor variable that does not point to a query work area, PL/SQL raises the `INVALID_CURSOR` exception. You can make a cursor variable (or parameter) point to a query work area in two ways:

- OPEN the cursor variable FOR the query.
- Assign to the cursor variable the value of an already opened host cursor variable or PL/SQL cursor variable.

If you assign an unopened cursor variable to another cursor variable, the second one remains invalid even after you open the first one.

Be careful when passing cursor variables as parameters. At run time, PL/SQL raises `ROWTYPE_MISMATCH` if the return types of the actual and formal parameters are incompatible.

Restrictions on Cursor Variables

Currently, cursor variables are subject to the following restrictions:

- You cannot declare cursor variables in a package specification, as illustrated in [Example 6–34](#).
- If you bind a host cursor variable into PL/SQL from an OCI client, you cannot fetch from it on the server side unless you also open it there on the same server call.
- You cannot use comparison operators to test cursor variables for equality, inequality, or nullity.
- Database columns cannot store the values of cursor variables. There is no equivalent type to use in a `CREATE TABLE` statement.
- You cannot store cursor variables in an associative array, nested table, or varray.
- Cursors and cursor variables are not interoperable; that is, you cannot use one where the other is expected. For example, you cannot reference a cursor variable in a cursor FOR loop.

Example 6–34 Declaration of Cursor Variables in a Package

```
CREATE PACKAGE emp_data AS
  TYPE EmpCurTyp IS REF CURSOR RETURN employees%ROWTYPE;
  -- emp_cv EmpCurTyp; -- not allowed
  PROCEDURE open_emp_cv;
END emp_data;
/
CREATE PACKAGE BODY emp_data AS
  -- emp_cv EmpCurTyp; -- not allowed
  PROCEDURE open_emp_cv IS
    emp_cv EmpCurTyp; -- this is legal
  BEGIN
    OPEN emp_cv FOR SELECT * FROM employees;
  END open_emp_cv;
END emp_data;
/
```

Note:

- Using a REF CURSOR variable in a server-to-server RPC results in an error. However, a REF CURSOR variable is permitted in a server-to-server RPC if the remote database is a non-Oracle database accessed through a Procedural Gateway.
- LOB parameters are not permitted in a server-to-server RPC.

Using Cursor Expressions

A cursor expression returns a nested cursor. Each row in the result set can contain values as usual, plus cursors produced by subqueries involving the other values in the row. A single query can return a large set of related values retrieved from multiple tables. You can process the result set with nested loops that fetch first from the rows of the result set, then from any nested cursors within those rows.

PL/SQL supports queries with cursor expressions as part of cursor declarations, REF CURSOR declarations and ref cursor variables. You can also use cursor expressions in dynamic SQL queries. Here is the syntax:

```
CURSOR (subquery)
```

A nested cursor is implicitly opened when the containing row is fetched from the parent cursor. The nested cursor is closed only when:

- The nested cursor is explicitly closed by the user
- The parent cursor is re-executed
- The parent cursor is closed
- The parent cursor is canceled
- An error arises during a fetch on one of its parent cursors. The nested cursor is closed as part of the clean-up.

Topics:

- [Restrictions on Cursor Expressions](#)
- [Example: Cursor Expressions](#)
- [Constructing REF CURSORS with Cursor Subqueries](#)

Restrictions on Cursor Expressions

The following are restrictions on cursor expressions:

- You cannot use a cursor expression with an implicit cursor.
- Cursor expressions can appear only:
 - In a SELECT statement that is not nested in any other query expression, except when it is a subquery of the cursor expression itself.
 - As arguments to table functions, in the FROM clause of a SELECT statement.
- Cursor expressions can appear only in the outermost SELECT list of the query specification.
- Cursor expressions cannot appear in view declarations.
- You cannot perform BIND and EXECUTE operations on cursor expressions.

Example: Cursor Expressions

In [Example 6–35](#), we find a specified location ID, and a cursor from which we can fetch all the departments in that location. As we fetch each department's name, we also get another cursor that lets us fetch their associated employee details from another table.

Example 6–35 Using a 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,
-- second item in the result set is another result set,
-- which is represented as a ref cursor and labelled "employees".
    CURSOR
      ( SELECT e.last_name FROM employees e
        WHERE e.department_id = d.department_id) employees
    FROM departments d WHERE department_name like 'A%';
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO dept_name, emp_cur;
    EXIT WHEN c1%NOTFOUND;
    DBMS_OUTPUT.PUT_LINE('Department: ' || dept_name);
-- for each row in the result set,
-- the result set from a subquery is processed
-- the set can be passed to a procedure
-- for processing rather than the loop
    LOOP
      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;
/
```

Constructing REF CURSORS with Cursor Subqueries

You can use cursor subqueries, also known as cursor expressions, to pass sets of rows as parameters to functions. For example, this statement passes a parameter to the `StockPivot` function consisting of a `REF CURSOR` that represents the rows returned by the cursor subquery:

```
SELECT * FROM TABLE (StockPivot(
                        CURSOR(SELECT * FROM StockTable)));
```

Cursor subqueries are often used with table functions, which are explained in ["Performing Multiple Transformations with Pipelined Table Functions"](#) on page 12-31.

Overview of Transaction Processing in PL/SQL

This section explains transaction processing with PL/SQL using `SQL COMMIT`, `SAVEPOINT`, and `ROLLBACK` statements that ensure the consistency of a database. You can include these SQL statements directly in your PL/SQL programs. Transaction

processing is an Oracle feature, available through all programming languages, that lets multiple users 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.

You usually do not need to write extra code to prevent problems with multiple users accessing data concurrently. Oracle uses locks to control concurrent access to data, and locks only the minimum amount of data necessary, for as little time as possible. You can request locks on tables or rows if you really do need this level of control. You can choose from several modes of locking such as `row share` and `exclusive`.

Topics:

- [Using COMMIT in PL/SQL](#)
- [Using ROLLBACK in PL/SQL](#)
- [Using SAVEPOINT in PL/SQL](#)
- [How Oracle Database Does Implicit Rollbacks](#)
- [Ending Transactions](#)
- [Setting Transaction Properties \(SET TRANSACTION Statement\)](#)
- [Overriding Default Locking](#)

See Also:

- *Oracle Database Concepts* for information about transactions
- *Oracle Database SQL Language Reference* for information about the COMMIT statement
- *Oracle Database SQL Language Reference* for information about the SAVEPOINT statement
- *Oracle Database SQL Language Reference* for information about the ROLLBACK statement

Using COMMIT in PL/SQL

The COMMIT statement ends the current transaction, making any changes made during that transaction permanent, and visible to other users. Transactions are not tied to PL/SQL BEGIN-END blocks. A block can contain multiple transactions, and a transaction can span multiple blocks.

[Example 6-36](#) illustrates a transaction that transfers money from one bank account to another. It is important that the money come out of one account, and into the other, at exactly the same moment. Otherwise, a problem partway through might make the money be lost from both accounts or be duplicated in both accounts.

Example 6-36 Using COMMIT with the WRITE Clause

```
CREATE TABLE accounts (account_id NUMBER(6), balance NUMBER (10,2));
INSERT INTO accounts VALUES (7715, 6350.00);
INSERT INTO accounts VALUES (7720, 5100.50);
DECLARE
    transfer NUMBER(8,2) := 250;
BEGIN
    UPDATE accounts SET balance = balance - transfer
        WHERE account_id = 7715;
    UPDATE accounts SET balance = balance + transfer
        WHERE account_id = 7720;
    COMMIT COMMENT 'Transfer from 7715 to 7720'
```

```
        WRITE IMMEDIATE NOWAIT;  
END;  
/
```

The optional `COMMENT` clause lets you specify a comment to be associated with a distributed transaction. If a network or computer fails during the commit, the state of the distributed transaction might be unknown or in doubt. In that case, Oracle stores the text specified by `COMMENT` in the data dictionary along with the transaction ID.

Asynchronous commit provides more control for the user with the `WRITE` clause. This option specifies the priority with which the redo information generated by the commit operation is written to the redo log.

For more information on using `COMMIT`, see "Committing Transactions" in *Oracle Database Advanced Application Developer's Guide*. For information about distributed transactions, see *Oracle Database Concepts*.

See Also: ["COMMIT Statement"](#) on page 13-28

Using ROLLBACK in PL/SQL

The `ROLLBACK` statement ends the current transaction and undoes any changes made during that transaction. If you make a mistake, such as deleting the wrong row from a table, a rollback restores the original data. If you cannot finish a transaction because an exception is raised or a SQL statement fails, a rollback lets you take corrective action and perhaps start over.

[Example 6–37](#) inserts information about an employee into three different database tables. If an `INSERT` statement tries to store a duplicate employee number, the predefined exception `DUP_VAL_ON_INDEX` is raised. To make sure that changes to all three tables are undone, the exception handler executes a `ROLLBACK`.

Example 6–37 Using ROLLBACK

```
CREATE TABLE emp_name AS SELECT employee_id, last_name  
    FROM employees;  
CREATE UNIQUE INDEX empname_ix ON emp_name (employee_id);  
CREATE TABLE emp_sal AS SELECT employee_id, salary FROM employees;  
CREATE UNIQUE INDEX empsal_ix ON emp_sal (employee_id);  
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 VALUES (emp_id, emp_lastname);  
    INSERT INTO emp_sal VALUES (emp_id, emp_salary);  
    INSERT INTO emp_job VALUES (emp_id, emp_jobid);  
EXCEPTION  
    WHEN DUP_VAL_ON_INDEX THEN  
        ROLLBACK;  
        DBMS_OUTPUT.PUT_LINE('Inserts were rolled back');  
END;
```

/

See Also: ["ROLLBACK Statement"](#) on page 13-124

Using SAVEPOINT in PL/SQL

SAVEPOINT 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 6–38](#) marks a savepoint before doing an insert. If the INSERT statement tries to store a duplicate value in the `employee_id` column, the predefined exception `DUP_VAL_ON_INDEX` is raised. In that case, you roll back to the savepoint, undoing just the insert.

Example 6–38 Using SAVEPOINT with ROLLBACK

```
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 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 within a recursive subprogram, new instances of the SAVEPOINT statement are executed 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 within a transaction moves the savepoint from its old position to the current point in the transaction. This means that a rollback to the savepoint affects only the current part of your transaction, as shown in [Example 6–39](#).

Example 6–39 Reusing a SAVEPOINT with ROLLBACK

```
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;
    -- Move my_savepoint to current point
    SAVEPOINT my_savepoint;
    INSERT INTO emp_name 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;
/

Tip: ["SAVEPOINT Statement"](#) on page 13-127

How Oracle Database Does Implicit Rollbacks

Before executing an INSERT, UPDATE, or DELETE statement, Oracle marks an implicit savepoint (unavailable to you). If the statement fails, Oracle 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.

Oracle can also roll back single SQL statements to break deadlocks. Oracle signals an error to one of the participating transactions and rolls back the current statement in that transaction.

Before executing a SQL statement, Oracle must parse it, that is, examine it to make sure it follows syntax rules and refers to valid schema objects. Errors detected while executing 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.

Ending Transactions

Explicitly commit or roll back every transaction. Whether you issue the commit or rollback in your PL/SQL program or from a client program depends on the application logic. If you do not commit or roll back a transaction explicitly, the client environment determines its final state.

For example, in the SQL*Plus environment, if your PL/SQL block does not include a COMMIT or ROLLBACK statement, the final state of your transaction depends on what you do after running the block. If you execute a data definition, data control, or COMMIT statement or if you issue the EXIT, DISCONNECT, or QUIT statement, Oracle commits the transaction. If you execute a ROLLBACK statement or stop the SQL*Plus session, Oracle rolls back the transaction.

Setting Transaction Properties (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 6-40](#) a store manager uses a read-only transaction to 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.

Example 6-40 Using SET TRANSACTION to Begin a Read-only Transaction

```
DECLARE
    daily_order_total    NUMBER(12,2);
    weekly_order_total   NUMBER(12,2);
    monthly_order_total  NUMBER(12,2);
BEGIN
    COMMIT; -- ends previous transaction
    SET TRANSACTION READ ONLY NAME 'Calculate Order Totals';
    SELECT SUM (order_total) INTO daily_order_total FROM orders
        WHERE order_date = SYSDATE;
    SELECT SUM (order_total) INTO weekly_order_total FROM orders
        WHERE order_date = SYSDATE - 7;
    SELECT SUM (order_total) INTO monthly_order_total FROM orders
        WHERE order_date = SYSDATE - 30;
    COMMIT; -- ends read-only transaction
END;
/
```

The `SET TRANSACTION` statement must be the first SQL statement in a read-only transaction and can only appear once in a transaction. If you set a transaction to `READ ONLY`, subsequent queries see only changes committed before the transaction began. The use of `READ ONLY` does not affect other users or transactions.

Restrictions on SET TRANSACTION

Only the `SELECT INTO`, `OPEN`, `FETCH`, `CLOSE`, `LOCK TABLE`, `COMMIT`, and `ROLLBACK` statements are allowed in a read-only transaction. Queries cannot be `FOR UPDATE`.

Overriding Default Locking

By default, Oracle locks data structures for you automatically, which is a major strength of the Oracle database: different applications can read and write to the same data without harming each other's data or coordinating with each other.

You can request data locks on specific rows or entire tables if you need to override default locking. Explicit locking lets you deny access to data for the duration of a transaction.:

- With the `LOCK TABLE` statement, you can explicitly lock entire tables.
- With the `SELECT FOR UPDATE` statement, you can explicitly lock specific rows of a table to make sure they do not change after you have read them. That way, you can check which or how many rows will be affected by an `UPDATE` or `DELETE` statement before issuing the statement, and no other application can change the rows in the meantime.

Topics:

- [Using FOR UPDATE](#)
- [Using LOCK TABLE](#)
- [Fetching Across Commits](#)

Using FOR UPDATE

When you declare a cursor that will be referenced in the `CURRENT OF` clause of an `UPDATE` or `DELETE` statement, you must use the `FOR UPDATE` clause to acquire exclusive row locks. An example follows:

```
DECLARE
  CURSOR c1 IS SELECT employee_id, salary FROM employees
                WHERE job_id = 'SA_REP' AND commission_pct > .10
                FOR UPDATE NOWAIT;
```

The `SELECT FOR UPDATE` statement identifies the rows that will be updated or deleted, then locks each row in the result set. This is useful when you want to base an update on the existing values in a row. In that case, you must make sure the row is not changed by another user before the update.

The optional keyword `NOWAIT` tells Oracle not to wait if requested rows have been locked by another user. Control is immediately returned to your program so that it can do other work before trying again to acquire the lock. If you omit the keyword `NOWAIT`, Oracle waits until the rows are available.

All rows are locked when you open the cursor, not as they are fetched. The rows are unlocked when you commit or roll back the transaction. Since the rows are no longer locked, you cannot fetch from a `FOR UPDATE` cursor after a commit.

When querying multiple tables, you can use the `FOR UPDATE` clause to confine row locking to particular tables. Rows in a table are locked only if the `FOR UPDATE OF` clause refers to a column in that table. For example, the following query locks rows in the `employees` table but not in the `departments` table:

```
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;
```

As shown in [Example 6–41](#), you use the `CURRENT OF` clause in an `UPDATE` or `DELETE` statement to refer to the latest row fetched from a cursor.

Example 6–41 Using CURRENT OF to Update the Latest Row Fetched from a Cursor

```
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;
/

```

Using LOCK TABLE

You use the `LOCK TABLE` statement to lock entire database tables in a specified lock mode so that you can share or deny access to them. Row share locks allow concurrent access to a table; they prevent other users from locking the entire table for exclusive use. Table locks are released when your transaction issues a commit or rollback.

```
LOCK TABLE employees IN ROW SHARE MODE NOWAIT;
```

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. For more information about lock modes, see *Oracle Database Advanced Application Developer's Guide*.

A table lock never keeps 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 will one transaction wait for the other to complete.

Fetching Across Commits

PL/SQL raises an exception if you try to fetch from a `FOR UPDATE` cursor after doing a commit. The `FOR UPDATE` clause locks the rows when you open the cursor, and unlocks them when you commit.

```

DECLARE
-- if "FOR UPDATE OF salary" is included on following line,
-- an error is raised
    CURSOR c1 IS SELECT * FROM employees;
    emp_rec employees%ROWTYPE;
BEGIN
    OPEN c1;
    LOOP
        -- FETCH fails on the second iteration with FOR UPDATE
        FETCH c1 INTO emp_rec;
        EXIT WHEN c1%NOTFOUND;
        IF emp_rec.employee_id = 105 THEN
            UPDATE employees SET salary = salary * 1.05
                WHERE employee_id = 105;
        END IF;
        COMMIT; -- releases locks
    END LOOP;
END;
/

```

If you want to fetch across commits, use the `ROWID` pseudocolumn to mimic the `CURRENT OF` clause. Select the rowid of each row into a `UROWID` variable, then use the rowid to identify the current row during subsequent updates and deletes.

Example 6–42 Fetching Across COMMITs Using ROWID

```

DECLARE
    CURSOR c1 IS SELECT last_name, job_id, rowid
        FROM employees;

```

```

my_lastname employees.last_name%TYPE;
my_jobid    employees.job_id%TYPE;
my_rowid    UROWID;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO my_lastname, my_jobid, my_rowid;
    EXIT WHEN c1%NOTFOUND;
    UPDATE employees SET salary = salary * 1.02 WHERE rowid = my_rowid;
    -- this mimics WHERE CURRENT OF c1
    COMMIT;
  END LOOP;
  CLOSE c1;
END;
/

```

Because the fetched rows are not locked by a FOR UPDATE clause, other users might unintentionally overwrite your changes. The extra space needed for read consistency is not released until the cursor is closed, which can slow down processing for large updates.

The next example shows that you can use the %ROWTYPE attribute with cursors that reference the ROWID pseudocolumn:

```

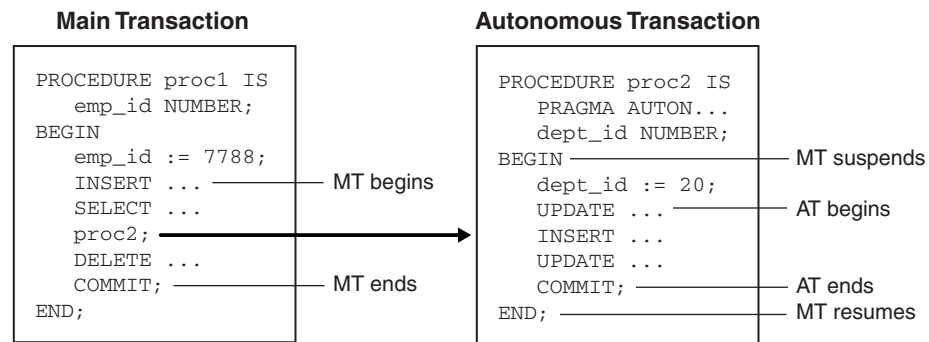
DECLARE
  CURSOR c1 IS SELECT employee_id, last_name, salary, rowid FROM employees;
  emp_rec c1%ROWTYPE;
BEGIN
  OPEN c1;
  LOOP
    FETCH c1 INTO emp_rec;
    EXIT WHEN c1%NOTFOUND;
    IF emp_rec.salary = 0 THEN
      DELETE FROM employees WHERE rowid = emp_rec.rowid;
    END IF;
  END LOOP;
  CLOSE c1;
END;
/

```

Doing Independent Units of Work with Autonomous Transactions

An autonomous transaction is an independent transaction started by another transaction, the main transaction. Autonomous transactions do SQL operations and commit or roll back, without committing or rolling back the main transaction. For example, if you write auditing data to a log table, you want to commit the audit data even if the operation you are auditing later fails; if something goes wrong recording the audit data, you do not want the main operation to be rolled back.

[Figure 6–1](#) shows how control flows from the main transaction (MT) to an autonomous transaction (AT) and back again.

Figure 6–1 Transaction Control Flow

Topics:

- [Advantages of Autonomous Transactions](#)
- [Defining Autonomous Transactions](#)
- [Controlling Autonomous Transactions](#)
- [Using Autonomous Triggers](#)
- [Invoking Autonomous Functions from SQL](#)

Advantages of Autonomous Transactions

Once started, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. You can log events, increment retry counters, and so on, even if the main transaction rolls back.

More important, autonomous transactions help you build modular, reusable software components. You can encapsulate autonomous transactions within stored subprograms. A calling application does not need to know whether operations done by that stored subprogram succeeded or failed.

Defining Autonomous Transactions

To define autonomous transactions, you use the pragma (compiler directive) `AUTONOMOUS_TRANSACTION`. The pragma instructs the PL/SQL compiler to mark a routine as autonomous (independent). In this context, the term routine includes

- Top-level (not nested) anonymous PL/SQL blocks
- Local, standalone, and packaged subprograms
- Methods of a SQL object type
- Database triggers

You can code the pragma anywhere in the declarative section of a routine. But, for readability, code the pragma at the top of the section. The syntax is `PRAGMA AUTONOMOUS_TRANSACTION`.

Example 6–43 marks a packaged function as autonomous. You cannot use the pragma to mark all subprograms in a package (or all methods in an object type) as autonomous. Only individual routines can be marked autonomous.

Example 6–43 Declaring an Autonomous Function in a Package

```
CREATE OR REPLACE PACKAGE emp_actions AS -- package specification
```

```

        FUNCTION raise_salary (emp_id NUMBER, sal_raise NUMBER)
            RETURN NUMBER;
    END emp_actions;
/
CREATE OR REPLACE PACKAGE BODY emp_actions AS -- package body
-- code for function raise_salary
    FUNCTION raise_salary (emp_id NUMBER, sal_raise NUMBER)
        RETURN NUMBER IS
        PRAGMA AUTONOMOUS_TRANSACTION;
        new_sal NUMBER(8,2);
    BEGIN
        UPDATE employees SET salary =
            salary + sal_raise WHERE employee_id = emp_id;
        COMMIT;
        SELECT salary INTO new_sal FROM employees
            WHERE employee_id = emp_id;
        RETURN new_sal;
    END raise_salary;
END emp_actions;
/

```

[Example 6-44](#) marks a standalone subprogram as autonomous.

Example 6-44 Declaring an Autonomous Standalone Procedure

```

CREATE 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 6-45](#) marks a PL/SQL block as autonomous. However, you cannot mark a nested PL/SQL block as autonomous.

Example 6-45 Declaring an Autonomous PL/SQL Block

```

DECLARE
    PRAGMA AUTONOMOUS_TRANSACTION;
    emp_id NUMBER(6);
    amount NUMBER(6,2);
BEGIN
    emp_id := 200;
    amount := 200;
    UPDATE employees SET salary = salary - amount WHERE employee_id = emp_id;
    COMMIT;
END;
/

```

[Example 6-46](#) marks a database trigger as autonomous. Unlike regular triggers, autonomous triggers can contain transaction control statements such as COMMIT and ROLLBACK.

Example 6-46 Declaring an Autonomous Trigger

```

CREATE TABLE emp_audit ( emp_audit_id NUMBER(6), up_date DATE,
                        new_sal NUMBER(8,2), old_sal NUMBER(8,2) );

```

```

CREATE OR REPLACE TRIGGER audit_sal
  AFTER UPDATE OF salary ON employees FOR EACH ROW
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  -- bind variables are used here for values
  INSERT INTO emp_audit VALUES( :old.employee_id, SYSDATE,
                                :new.salary, :old.salary );

  COMMIT;
END;
/

```

Topics:

- [Comparison of Autonomous Transactions and Nested Transactions](#)
- [Transaction Context](#)
- [Transaction Visibility](#)

Comparison of Autonomous Transactions and Nested Transactions

Although an autonomous transaction is started by another transaction, it is not a nested transaction:

- 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.

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;
```

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 executed since the last commit or rollback make up the current transaction.

To control autonomous transactions, use the following statements, which apply only to the current (active) transaction:

- COMMIT
- ROLLBACK [TO *savepoint_name*]
- SAVEPOINT *savepoint_name*
- SET TRANSACTION

Note:

- Transaction properties set in the main transaction apply only to that transaction, not to its autonomous transactions, and vice versa.
 - Cursor attributes are not affected by autonomous transactions.
-
-

Topics:

- [Entering and Exiting](#)
- [Committing and Rolling Back](#)
- [Using Savepoints](#)
- [Avoiding Errors with Autonomous Transactions](#)

Entering and Exiting

When you enter the executable section of an autonomous routine, the main transaction suspends. When you exit the routine, the main transaction resumes.

To exit normally, you must explicitly commit or roll back all autonomous transactions. If the routine (or any routine invoked by it) has pending transactions, an exception is raised, and the pending transactions are rolled back.

Committing and Rolling Back

COMMIT and ROLLBACK end the active autonomous transaction but do not exit the autonomous routine. When one transaction ends, the next SQL statement begins another transaction. A single autonomous routine can contain several autonomous transactions, if it issues several COMMIT statements.

Using Savepoints

The scope of a savepoint is the transaction in which it is defined. Savepoints defined in the main transaction are unrelated to savepoints defined in its autonomous transactions. In fact, the main transaction and an autonomous transaction can use the same savepoint names.

You can roll back only to savepoints marked in the current transaction. In an autonomous transaction, you cannot roll back to a savepoint marked in the main transaction. To do so, you must resume the main transaction by exiting the autonomous routine.

When in the main transaction, rolling back to a savepoint marked before you started an autonomous transaction does *not* roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

Avoiding Errors with Autonomous Transactions

To avoid some common errors, keep the following points in mind:

- If an autonomous transaction attempts to access a resource held by the main transaction, a deadlock can occur. Oracle raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.
- The Oracle 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, Oracle raises an exception. If the exception goes unhandled, the transaction is rolled back.

Using Autonomous Triggers

Among other things, you can use database triggers to log events transparently. Suppose you want to track all inserts into a table, even those that roll back. In [Example 6-47](#), you use a trigger to insert duplicate rows into a shadow table. Because it is autonomous, the trigger can commit changes to the shadow table whether or not you commit changes to the main table.

Example 6-47 Using Autonomous Triggers

```
CREATE TABLE emp_audit ( emp_audit_id NUMBER(6), up_date DATE,
                        new_sal NUMBER(8,2), old_sal NUMBER(8,2) );

-- create an autonomous trigger that inserts
-- into the audit table before each update
-- of salary in the employees table
CREATE OR REPLACE TRIGGER audit_sal
  BEFORE UPDATE OF salary ON employees FOR EACH ROW
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO emp_audit VALUES( :old.employee_id, SYSDATE,
                                :new.salary, :old.salary );

  COMMIT;
END;
/

-- update the salary of an employee, and then commit the insert
UPDATE employees SET salary
  salary * 1.05 WHERE employee_id = 115;
COMMIT;

-- update another salary, then roll back the update
UPDATE employees SET salary = salary * 1.05 WHERE employee_id = 116;
ROLLBACK;

-- show that both committed and rolled-back updates
-- add rows to audit table
SELECT * FROM emp_audit WHERE emp_audit_id = 115 OR emp_audit_id = 116;
```

Unlike regular triggers, autonomous triggers can execute DDL statements using native dynamic SQL, explained in [Chapter 7, "Using Dynamic SQL"](#). In the following example, trigger `drop_temp_table` drops a temporary database table after a row is inserted in table `emp_audit`.

```
CREATE TABLE emp_audit ( emp_audit_id NUMBER(6), up_date DATE,
```

```

                new_sal NUMBER(8,2), old_sal NUMBER(8,2) );
CREATE TABLE temp_audit ( emp_audit_id NUMBER(6), up_date DATE);

CREATE OR REPLACE TRIGGER drop_temp_table
  AFTER INSERT ON emp_audit
DECLARE
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  EXECUTE IMMEDIATE 'DROP TABLE temp_audit';
  COMMIT;
END;
/

```

For more information about database triggers, see [Chapter 9, "Using Triggers"](#).

Invoking Autonomous Functions from SQL

A function invoked from SQL statements must obey certain rules meant to control side effects. See ["Controlling Side Effects of PL/SQL Subprograms"](#) on page 8-26. To check for violations of the rules, you can use the pragma `RESTRICT_REFERENCES`. The pragma asserts that a function does not read or write database tables or package variables. For more information, See *Oracle Database Advanced Application Developer's Guide*.

However, by definition, autonomous routines never violate the rules `read no database state (RNDS)` and `write no database state (WNDS)` no matter what they do. This can be useful, as [Example 6–48](#) shows. When you invoke the packaged function `log_msg` from a query, it inserts a message into database table `debug_output` without violating the rule `write no database state`.

Example 6–48 Invoking an Autonomous Function

```

-- create the debug table
CREATE TABLE debug_output (msg VARCHAR2(200));

-- create the package spec
CREATE PACKAGE debugging AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2;
  PRAGMA RESTRICT_REFERENCES(log_msg, WNDS, RNDS);
END debugging;
/
-- create the package body
CREATE PACKAGE BODY debugging AS
  FUNCTION log_msg (msg VARCHAR2) RETURN VARCHAR2 IS
    PRAGMA AUTONOMOUS_TRANSACTION;
  BEGIN
    -- the following insert does not violate the constraint
    -- WNDS because this is an autonomous routine
    INSERT INTO debug_output VALUES (msg);
    COMMIT;
    RETURN msg;
  END;
END debugging;
/
-- invoke the packaged function from a 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;
/
```

Using Dynamic SQL

Dynamic SQL is a programming methodology for generating and executing 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 execute DDL statements, or when you do not know at compilation time the full text of a SQL statement or the number or datatypes 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 executing dynamic SQL statements
- `DBMS_SQL` package, an API for building, executing, 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 datatypes 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](#)
- [Using Native Dynamic SQL](#)
- [Using DBMS_SQL Package](#)
- [Avoiding SQL Injection in PL/SQL](#)

When You Need Dynamic SQL

In PL/SQL, you need dynamic SQL in order to execute the following:

- 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 the following 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 Concepts*.

For information about using static SQL statements with PL/SQL, see [Chapter 6, "Using Static SQL"](#).

Using Native Dynamic SQL

Native dynamic SQL processes most dynamic SQL statements by means of the `EXECUTE IMMEDIATE` statement.

If the dynamic SQL statement is a `SELECT` statement that returns multiple rows, native dynamic SQL gives you the following 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 ["Managing Cursors in PL/SQL"](#) on page 6-7.

Topics:

- [Using the EXECUTE IMMEDIATE Statement](#)
- [Using the OPEN-FOR, FETCH, and CLOSE Statements](#)
- [Repeating Placeholder Names in Dynamic SQL Statements](#)

Using the 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 that no bind argument has a datatype 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 13-51.

Example 7-1 Invoking a Subprogram from a Dynamic PL/SQL Block

```
-- Subprogram that dynamic PL/SQL block invokes:
CREATE 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 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 Datatype in Native Dynamic SQL

```
DECLARE
    FUNCTION f (x INTEGER)
        RETURN BOOLEAN
    AS
    BEGIN
        ...
    END f;
    dyn_stmt VARCHAR2(200);
    b1       BOOLEAN;
BEGIN
    dyn_stmt := 'BEGIN :b := f(5); END;';
    -- Fails because SQL does not support BOOLEAN datatype:
    EXECUTE IMMEDIATE dyn_stmt USING OUT b1;
```

```
END;
```

Example 7-3 Uninitialized Variable for NULL in USING Clause

```
CREATE TABLE employees_temp AS
  SELECT * FROM EMPLOYEES
/
DECLARE
  a_null CHAR(1); -- Set to NULL automatically at run time
BEGIN
  EXECUTE IMMEDIATE 'UPDATE employees_temp SET commission_pct = :x'
    USING a_null;
END;
/
```

Using the 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 13-104.

2. Use the `FETCH` statement to retrieve result set rows one at a time, several at a time, or all at once.

For syntax details, see "[FETCH Statement](#)" on page 13-67.

3. Use the `CLOSE` statement to close the cursor variable.

For syntax details, see "[CLOSE Statement](#)" on page 13-18.

[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;
/

```

Repeating 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 the following 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 executes the following 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 executes the following 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 do not need to 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;  
/
```

Using 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 can also use the SQL cursor number to obtain information about the SQL statement that you are executing.

You must use the DBMS_SQL package to execute a dynamic SQL statement when you don't know either of the following until run-time:

- SELECT list
- What placeholders in a SELECT or DML statement must be bound

In the following 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 ["Using 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 the following:

- [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 executing a dynamic SQL statement that has an unknown number of input or output variables ("Method 4").

DBMS_SQL.TO_REFCURSOR Function

The `DBMS_SQL.TO_REFCURSOR` function converts a SQL cursor number to a weakly-typed variable of the PL/SQL datatype `REF CURSOR`, which you can use in native dynamic SQL statements.

Before passing a SQL cursor number to the `DBMS_SQL.TO_REFCURSOR` function, you must `OPEN`, `PARSE`, and `EXECUTE` it (otherwise an error occurs).

After you convert a SQL cursor number to a `REF CURSOR` variable, `DBMS_SQL` operations can access it only as the `REF CURSOR` variable, not as the SQL cursor number. For example, using the `DBMS_SQL.IS_OPEN` function to see if a converted SQL cursor number is still open causes an error.

[Example 7-6](#) uses the `DBMS_SQL.TO_REFCURSOR` function to switch from the `DBMS_SQL` package to native dynamic SQL.

Example 7-6 Switching from DBMS_SQL Package to Native Dynamic SQL

```
CREATE OR REPLACE TYPE vc_array IS TABLE OF VARCHAR2(200);
/
CREATE OR REPLACE TYPE numlist IS TABLE OF NUMBER;
/
CREATE OR REPLACE PROCEDURE do_query_1 (
    placeholder vc_array,
    bindvars vc_array,
    sql_stmt VARCHAR2
)
IS
    TYPE curtype IS REF CURSOR;
    src_cur      curtype;
    curid        NUMBER;
    bindnames    vc_array;
    empnos       numlist;
    depts        numlist;
    ret          NUMBER;
    isopen       BOOLEAN;
BEGIN
    -- Open SQL cursor number:
    curid := DBMS_SQL.OPEN_CURSOR;

    -- Parse SQL cursor number:
    DBMS_SQL.PARSE(curid, sql_stmt, DBMS_SQL.NATIVE);

    bindnames := placeholder;

    -- Bind arguments:
    FOR i IN 1 .. bindnames.COUNT LOOP
        DBMS_SQL.BIND_VARIABLE(curid, bindnames(i), bindvars(i));
    END LOOP;

    -- Execute 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 causes an error because curid was converted to a REF CURSOR:
    isopen := DBMS_SQL.IS_OPEN(curid);

    CLOSE src_cur;
```

```
END;
/
```

DBMS_SQL.TO_CURSOR_NUMBER Function

The `DBMS_SQL.TO_CURSOR` function converts a `REF CURSOR` variable (either strongly or weakly typed) to a SQL cursor number, which you can pass to `DBMS_SQL` subprograms.

Before passing a `REF CURSOR` variable to the `DBMS_SQL.TO_CURSOR` function, you must `OPEN` it.

After you convert a `REF CURSOR` variable to a SQL cursor number, native dynamic SQL operations cannot access it.

After a `FETCH` operation begins, passing the `DBMS_SQL` cursor number to the `DBMS_SQL.TO_REFCURSOR` or `DBMS_SQL.TO_CURSOR` function causes an error.

[Example 7-7](#) uses the `DBMS_SQL.TO_CURSOR` function to switch from native dynamic SQL to the `DBMS_SQL` package.

Example 7-7 Switching from Native Dynamic SQL to DBMS_SQL Package

```
CREATE OR REPLACE PROCEDURE do_query_2 (sql_stmt VARCHAR2) IS
  TYPE curtype IS REF CURSOR;
  src_cur curtype;
  curid NUMBER;
  desctab DBMS_SQL.DESC_TAB;
  colcnt NUMBER;
  namevar VARCHAR2(50);
  numvar NUMBER;
  datevar DATE;
  empno NUMBER := 100;
BEGIN
  -- sql_stmt := SELECT ... FROM employees WHERE employee_id = :b1';

  -- Open REF CURSOR variable:
  OPEN src_cur FOR sql_stmt USING empno;

  -- Switch from native dynamic SQL to DBMS_SQL package:
  curid := DBMS_SQL.TO_CURSOR_NUMBER(src_cur);
  DBMS_SQL.DESCRIBE_COLUMNS(curid, colcnt, desctab);

  -- Define columns:
  FOR i IN 1 .. colcnt LOOP
    IF desctab(i).col_type = 2 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, numvar);
    ELSIF desctab(i).col_type = 12 THEN
      DBMS_SQL.DEFINE_COLUMN(curid, i, datevar);
      -- statements
    ELSE
      DBMS_SQL.DEFINE_COLUMN(curid, i, namevar);
    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;
/

```

Avoiding SQL Injection in PL/SQL

SQL injection is a technique for maliciously exploiting applications that use client-supplied data in SQL statements, thereby gaining unauthorized access to a database in order to view or manipulate restricted data. This section describes SQL injection vulnerabilities in PL/SQL and explains how you can guard against them.

Topics:

- [Overview of SQL Injection Techniques](#)
- [Guarding Against SQL Injection](#)

Overview of SQL Injection Techniques

SQL injection techniques differ, but they all exploit a single vulnerability: string input is not correctly validated and is concatenated into a dynamic SQL statement. This topic classifies SQL injection attacks as follows:

- [Statement Modification](#)
- [Statement Injection](#)

The following topics describe these techniques. To try the examples in your sample database, run the script in [Example 7–8](#).

Example 7–8 Setup for SQL Injection Examples

```

CONNECT HR/password
SET SERVEROUTPUT ON
SET LINESIZE 150
SET ECHO OFF

DROP TABLE user_pwd;
CREATE TABLE user_pwd (username VARCHAR2(100), password VARCHAR2(100));
INSERT INTO user_pwd VALUES ('whitehat', 'secret');
DROP TABLE delemp;
CREATE TABLE delemp AS SELECT * FROM employees;
COMMIT;

```

Statement Modification

Statement modification means deliberately altering a dynamic SQL statement so that it executes 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.

Suppose a Web form prompts a user to enter a username and password. When the user clicks **Submit**, the form invokes a PL/SQL stored subprogram that concatenates the username and password entered in the form to build a dynamic SQL statement. The subprogram executes the `SELECT` statement to authenticate the user.

The password-checking procedure `ckpwd` in [Example 7-9](#) uses concatenation to build a `SELECT` statement whose `FROM` clause specifies the `user_pwd` table. If the user enters a username and password stored in the `user_pwd` table, then the execution of the `SELECT` statement retrieves a single row and the user is authenticated. The `ckpwd` procedure displays the concatenated `SELECT` statement so that you can see which statement is executed.

Example 7-9 Password-Checking Procedure Vulnerable to Statement Modification

```
CREATE OR REPLACE PROCEDURE ckpwd (
  p_user IN VARCHAR2,
  p_pass IN VARCHAR2          )
IS
  v_query  VARCHAR2(100);
  v_output NUMBER;
BEGIN
  v_query :=
    q'{SELECT COUNT(*) FROM user_pwd }'
    || q'{WHERE username = }'
    || p_user
    || q{' AND password = }'
    || p_pass
    || q{'}';
  DBMS_OUTPUT.PUT_LINE (
    CHR(10)
    || 'Built the following statement: '
    || CHR(10)
    || v_query
    );
  EXECUTE IMMEDIATE v_query INTO v_output;
  IF v_output = 1 THEN
    DBMS_OUTPUT.PUT_LINE(CHR(10) || p_user || ' is authenticated');
  ELSE
    DBMS_OUTPUT.PUT_LINE(CHR(10) || ' access denied');
  END IF;
END;
/
```

Suppose that the user `whitehat` enters the password `secret` in a Web form. You can simulate this scenario by executing the following code:

```
BEGIN
  ckpwd (p_user => q'{whitehat}',
        p_pass => q'{secret}'
        );
END;
/
```

The resulting output is:

```
Built the following statement: SELECT COUNT(*) FROM user_pwd
WHERE username = 'whitehat' AND password = 'secret'
whitehat is authenticated
```

A malicious user can exploit the concatenation vulnerability and enter the username `x` in the Web-based form and the following text as a password:

```
x' OR 'x' = 'x
```

You can simulate this scenario by executing the following code:

```
BEGIN
  ckpwd (p_user => q'{x}', p_pass => q'{x' OR 'x' = 'x}');
END;
/
```

The resulting output is:

```
Built the following statement: SELECT COUNT(*) FROM user_pwd
WHERE username = 'x' AND password = 'x' OR 'x' = 'x'
x is authenticated
```

By using the cleverly constructed password, you alter the concatenated SQL statement so that the OR condition always returns TRUE. Thus, the SELECT statement whose FROM clause specifies the user_pwd table always succeeds no matter which username is entered.

Statement Injection

Statement injection means that a user appends one or more new SQL statements to a dynamically generated SQL statement. Anonymous PL/SQL blocks are vulnerable to this technique.

Suppose a Web form prompts a user to enter a username and password. When the user clicks Submit, the form invokes a PL/SQL stored subprogram that concatenates the username and password entered in the form into an anonymous PL/SQL block. The subprogram then executes the anonymous block to authenticate the user.

The procedure call_ckpwd in [Example 7-10](#) uses concatenation to build an anonymous block that invokes the ckpwd procedure from [Example 7-9](#). If the user enters a username and password stored in the user_pwd table, then the execution of the block retrieves a single row and the user is authenticated. The call_ckpwd procedure prints the concatenated text so that you can see which block is executed.

Example 7-10 Password-Checking Procedure Vulnerable to Statement Injection

```
CREATE OR REPLACE PROCEDURE call_ckpwd (
  p_user IN VARCHAR2,
  p_pass IN VARCHAR2
)
IS
  v_block VARCHAR2(100);
BEGIN
  v_block :=
    q'{BEGIN ckpwd('}
    || p_user
    || q'{, '}'
    || p_pass
    || q{'}; END; }';
  DBMS_OUTPUT.PUT_LINE (
    CHR(10)
    || 'Built the following anonymous block: '
    || CHR(10)
    || v_block
  );
  EXECUTE IMMEDIATE v_block;
END;
/
```

Suppose that the user whitehat enters the password secret in a Web form. You can simulate this scenario by executing the following code:

```
BEGIN
  call_ckpwd (p_user => q'{whitehat}', p_pass => q'{secret}');
END;
/
```

The resulting output is:

```
Built the following anonymous block: BEGIN ckpwd('whitehat', 'secret'); END;
  WHERE username = 'whitehat' AND password = 'secret'
whitehat is authenticated
```

The user `whitehat` could enter the following fraudulent string as the password in a Web form:

```
secret'); DELETE FROM hr.delemp WHERE UPPER('x') = UPPER('x
```

You can simulate this scenario by executing the following code:

```
BEGIN
  call_ckpwd (p_user => q'{whitehat}', p_pass => q'{secret}');
  DELETE FROM hr.delemp WHERE UPPER('x') = UPPER('x');
END;
/
```

The resulting output is:

```
Built the following anonymous block:
BEGIN
  ckpwd( 'whitehat', 'secret');
  DELETE FROM hr.delemp WHERE UPPER('x') = UPPER('x');
END;
Built the following statement:
SELECT COUNT(*) FROM user_pwd
  WHERE username = 'whitehat'
  AND password = 'secret'
whitehat is authenticated
```

The fraudulent password causes the system to authenticate `whitehat` and silently execute the injected `DELETE` statement. A `SELECT` statement whose `FROM` clause specifies the `delemp` table shows that the injected statement silently removed all rows from the table:

```
SELECT * FROM delemp;
no rows selected
```

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:

- [Using Bind Arguments to Guard Against SQL Injection](#)
- [Using Validation Checks to Guard Against SQL Injection](#)

Using Bind Arguments to Guard Against SQL Injection

Besides improving performance, bind arguments make your PL/SQL code immune to SQL injection attacks.

The procedure `ckpwd_bind` in [Example 7-11](#) is immune to SQL injection because it builds the dynamic SQL statement `v_query` with bind arguments (not by concatenation as in the vulnerable procedure in [Example 7-9](#)).

Example 7-11 Password-Checking Procedure Immune to SQL Injection

```

CREATE OR REPLACE PROCEDURE ckpwd_bind (
  p_user IN VARCHAR2,
  p_pass IN VARCHAR2
)
IS
  v_query VARCHAR2(100);
  v_output NUMBER;
BEGIN
  v_query := q'{SELECT COUNT(*) FROM user_pwd
    WHERE username = :1
      AND password = :2}';
  DBMS_OUTPUT.PUT_LINE(
    CHR(10) || 'Built the following statement: ' || CHR(10) || v_query);
  EXECUTE IMMEDIATE v_query
    INTO v_output
    USING p_user, p_pass;
  IF v_output = 1 THEN
    DBMS_OUTPUT.PUT_LINE(CHR(10) || p_user || ' is authenticated');
  ELSE
    DBMS_OUTPUT.PUT_LINE(CHR(10) || 'access denied');
  END IF;
END;
/

```

If the user passes the fraudulent password shown in [Example 7-9](#) to the `ckpwd_bind` procedure, the user is not authenticated.

You can simulate this scenario by executing the following code:

```

BEGIN
  ckpwd_bind (p_user => q'{x}', p_pass => q'{x' OR 'x' = 'x}');
END;
/

```

The resulting output is:

```

Built the following statement:
SELECT COUNT(*) FROM user_pwd
  WHERE username = :1 AND password = :2
access denied

```

The same binding technique fixes the vulnerable `call_ckpwd` procedure shown in [Example 7-10](#).

By using bind arguments exclusively in your code, you avoid concatenating SQL statements, preventing malicious users from altering or injecting additional statements. Oracle database uses the value of the bind argument exclusively and does not interpret its contents in any way. This technique is the most effective way to prevent SQL injection in PL/SQL programs.

Using Validation Checks to Guard Against SQL Injection

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`.

In [Example 7-12](#), 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 PL/SQL block invokes the procedure from both a dynamic PL/SQL block and a dynamic SQL statement.

Example 7-12 Dynamic SQL

```

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;
/

```

In validation-checking code, the subprograms in the package `DBMS_ASSERT` are often useful. For example, you can use the `DBMS_ASSERT.ENQUOTE_LITERAL` function to enclose a string literal in quotation marks. 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: *Oracle Database PL/SQL Packages and Types Reference* for information about `DBMS_ASSERT` subprograms.

The procedure `ckpwd` in [Example 7-13](#), which checks the validity of user input (user name and password), uses the `DBMS_ASSERT.ENQUOTE_LITERAL` function to enquote string literals and the `DBMS_ASSERT.SCHEMA_NAME` function to verify that `p_user` is an existing schema name.

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) will realize that he or she has guessed one of these correctly.

Example 7-13 Using `DBMS_ASSERT.ENQUOTE_LITERAL` When Validating User Input

```
CREATE OR REPLACE PROCEDURE ckpwd (
  p_user IN VARCHAR2,
  p_pass IN VARCHAR2          )
IS
  v_query  VARCHAR2(100);
  v_output NUMBER;
BEGIN
  v_query :=
    q'{SELECT COUNT(*) FROM user_pwd }'
    || q'{WHERE username = }'
    || DBMS_ASSERT.ENQUOTE_LITERAL(DBMS_ASSERT.SCHEMA_NAME(p_user))
    || q{' AND password = }'
    || DBMS_ASSERT.ENQUOTE_LITERAL(p_pass);
  DBMS_OUTPUT.PUT_LINE (
    CHR(10)
    || 'Built the following statement: '
    || CHR(10)
    || v_query
  );
  EXECUTE IMMEDIATE v_query INTO v_output;
  IF v_output = 1 THEN
    DBMS_OUTPUT.PUT_LINE(CHR(10) || p_user || ' is authenticated');
  ELSE
    DBMS_OUTPUT.PUT_LINE(CHR(10) || 'access denied');
  END IF;
END;
/
```

Using PL/SQL Subprograms

This chapter explains how to turn sets of statements into reusable subprograms. Subprograms are the building blocks of modular, maintainable applications.

Topics:

- [What Are PL/SQL Subprograms?](#)
- [Why Use PL/SQL Subprograms?](#)
- [Using the RETURN Statement](#)
- [Declaring Nested PL/SQL Subprograms](#)
- [Passing Parameters to PL/SQL Subprograms](#)
- [Overloading PL/SQL Subprogram Names](#)
- [How PL/SQL Subprogram Calls Are Resolved](#)
- [Using Invoker's Rights or Definer's Rights \(AUTHID Clause\)](#)
- [Using Recursive PL/SQL Subprograms](#)
- [Invoking External Subprograms](#)
- [Controlling Side Effects of PL/SQL Subprograms](#)
- [Understanding PL/SQL Subprogram Parameter Aliasing](#)
- [Using the Cross-Session PL/SQL Function Result Cache](#)

What Are PL/SQL Subprograms?

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

Topics:

- [Subprogram Parts](#)
- [Subprogram Types](#)
- [Subprogram Calls](#)
- [Subprogram Examples](#)

Subprogram Parts

A subprogram consists of a specification ("spec") and a body. To declare a subprogram, you must provide the spec, which includes descriptions of any parameters. To define a

subprogram, you must provide both the spec and the body. You can either declare a subprogram first and define it later in the same block or subprogram, or you can declare and define it at the same time.

A subprogram body has the parts described in [Table 8-1](#).

Table 8-1 Subprogram Body Parts

Part	Description
PRAGMA AUTONOMOUS_TRANSACTION (optional)	Makes the subprogram autonomous (independent). For more information, see "Doing Independent Units of Work with Autonomous Transactions" on page 6-40.
Declarative part (optional)	Contains declarations of local types, cursors, constants, variables, exceptions, and nested subprograms (these items cease to exist when the subprogram ends). Does not begin with the keyword DECLARE, as the declarative part of a block does.
Executable part	Contains statements that assign values, control execution, and manipulate data.
Exception-handling part (optional)	Contains code that handles run-time errors.

Subprogram Types

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

A procedure and a function have the same structure, except that only a function has the following items:

Item	Description
RETURN clause	Specifies the datatype of the return value (required).
RETURN statement	Specifies the return value (required).
DETERMINISTIC option	Helps the optimizer avoid redundant function calls.
PARALLEL_ENABLED option	Allows the function to be used safely in slave sessions of parallel DML evaluations.
PIPELINED option	Returns the results of a table function iteratively.
RESULT_CACHE option	Stores function results in the cross-session function result cache.
RESULT_CACHE clause	Specifies the data sources on which the results of a function.

For the syntax of subprogram declarations and definitions, including descriptions of the items in the preceding table, see ["Procedure Declaration and Definition"](#) on page 13-110 and ["Function Declaration and Definition"](#) on page 13-74. For additional information, see ["Using the RETURN Statement"](#) on page 8-5 and ["Using the Cross-Session PL/SQL Function Result Cache"](#) on page 8-29.

Note: PL/SQL procedures and functions are different from SQL procedures and functions created with the SQL statements CREATE PROCEDURE and CREATE FUNCTION.

See Also:

- *Oracle Database SQL Language Reference* for information about the CREATE PROCEDURE statement
- *Oracle Database SQL Language Reference* for information about the CREATE FUNCTION statement

Subprogram Calls

A subprogram call has this form:

```
subprogram_name [ (parameter [, parameter]... )
```

A procedure call is a PL/SQL statement. For example:

```
raise_salary(emp_id, amount);
```

A function call is part of an expression. For example:

```
IF sal_ok(new_sal, new_title) THEN ...
```

For information about subprogram parameters, see ["Passing Parameters to PL/SQL Subprograms"](#) on page 8-6.

Subprogram Examples

In [Example 8-1](#), a block declares, defines, and invokes a string-manipulation procedure, `double`, which accepts input and output parameters and handles potential errors. The procedure `double` includes the optional exception-handling part.

Example 8-1 Simple PL/SQL Procedure

```
-- Declarative part of block begins
DECLARE
  in_string VARCHAR2(100) := 'This is my test string.';
  out_string VARCHAR2(200);
  -- Procedure spec begins
  PROCEDURE double (original IN VARCHAR2,
                   new_string OUT VARCHAR2) AS
  -- Procedure spec ends
  -- Procedure body begins
  -- Executable part begins
  BEGIN
    new_string := original || ' + ' || original;
  -- Executable part ends
  -- Exception-handling part begins
  EXCEPTION
    WHEN VALUE_ERROR THEN
      DBMS_OUTPUT.PUT_LINE('Output buffer not long enough.');
```

```
  END;
  -- Exception-handling part ends
  -- Procedure body ends
-- Declarative part of block ends
-- Executable part of block begins
BEGIN
  double(in_string, out_string);
  DBMS_OUTPUT.PUT_LINE(in_string || ' - ' || out_string);
END;
-- Executable part of block ends
/
```

In [Example 8-2](#), a block declares, defines, and invokes a numeric function, `square`, which declares a local variable to hold temporary results and returns a value when finished. The function `square` omits the optional exception-handling part.

Example 8-2 Simple PL/SQL Function

```
-- Declarative part of block begins
DECLARE
  -- Function spec begins
  FUNCTION square (original NUMBER)
    RETURN NUMBER
  AS
    original_squared NUMBER;
  -- Function spec ends
  -- Function body begins
  -- Executable part begins
  BEGIN
    original_squared := original * original;
    RETURN original_squared;
  END;
  -- Executable part ends
  -- Function body ends
-- Declarative part of block ends
-- Executable part of block begins
BEGIN
  DBMS_OUTPUT.PUT_LINE(square(100));
END;
-- Executable part of block ends
/
```

Why Use PL/SQL Subprograms?

- Subprograms let you extend the PL/SQL language.

Procedures act like new statements. Functions act like new expressions and operators.
- Subprograms let you break a program down into manageable, well-defined modules.

You can use top-down design and the stepwise refinement approach to problem solving.
- Subprograms promote reusability.

Once tested, a subprogram can be reused in any number of applications. You can invoke PL/SQL subprograms from many different environments, so that you do not have to reinvent the wheel each time you use a new language or API to access the database.
- Subprograms promote maintainability.

You can change the internals of a subprogram without changing other subprograms that invoke it. Subprograms play a big part in other maintainability features, such as packages and object types.
- Dummy subprograms (stubs) let you defer the definition of procedures and functions until after testing the main program.

You can design applications from the top down, thinking abstractly, without worrying about implementation details.

- Subprograms can be grouped into PL/SQL packages.
Packages make code even more reusable and maintainable, and can be used to define an API. For more information about packages, see [Chapter 10, "Using PL/SQL Packages"](#).

Using the RETURN Statement

The RETURN statement immediately ends the execution of a subprogram and returns control to the caller. Execution continues with the statement following the subprogram call. (Do not confuse the RETURN statement with the RETURN clause in a function spec, which specifies the datatype of the return value.)

A subprogram can contain several RETURN statements. The subprogram does not have to conclude with a RETURN statement. Executing any RETURN statement completes the subprogram immediately.

In procedures, a RETURN statement does not return a value and so cannot contain an expression. The statement returns control to the caller before the end of the procedure.

In functions, a RETURN statement must contain an expression, which is evaluated when the RETURN statement is executed. The resulting value is assigned to the function identifier, which acts like a variable of the type specified in the RETURN clause. See the use of the RETURN statement in [Example 8–2](#) on page 8-4.

The expression in a function RETURN statement can be arbitrarily complex:

```
CREATE OR REPLACE FUNCTION half_of_square(original NUMBER)
  RETURN NUMBER IS
BEGIN
  RETURN (original * original)/2 + (original * 4);
END half_of_square;
/
```

In a function, there must be at least one execution path that leads to a RETURN statement. Otherwise, you get a function returned without value error at run time.

Declaring Nested PL/SQL Subprograms

You can declare subprograms in any PL/SQL block, subprogram, or package. The subprograms must go at the end of the declarative section, after all other items.

You must declare a subprogram before invoking it. This requirement can make it difficult to declare several nested subprograms that invoke each other.

You can declare interrelated nested subprograms using a forward declaration: a subprogram spec terminated by a semicolon, with no body.

Although the formal parameter list appears in the forward declaration, it must also appear in the subprogram body. You can place the subprogram body anywhere after the forward declaration, but they must appear in the same program unit.

Example 8–3 Forward Declaration for a Nested Subprogram

```
DECLARE
PROCEDURE proc1(number1 NUMBER); -- forward declaration
PROCEDURE proc2(number2 NUMBER) IS
  BEGIN
    proc1(number2); -- invokes proc1
  END;
```

```

PROCEDURE proc1(number1 NUMBER) IS
  BEGIN
    proc2 (number1); -- invokes proc2
  END;
BEGIN
  NULL;
END;
/

```

Passing Parameters to PL/SQL Subprograms

This section explains how to pass parameters to PL/SQL subprograms.

Topics:

- [Formal and Actual Subprogram Parameters](#)
- [Using Positional, Named, or Mixed Notation for PL/SQL Subprogram Parameters](#)
- [Specifying Subprogram Parameter Modes](#)
- [Using Default Values for Subprogram Parameters](#)

Formal and Actual Subprogram Parameters

Formal parameters are the variables declared in the subprogram specification and referenced in the subprogram body. **Actual parameters** are the variables or expressions that you pass to the subprogram when you invoke it. Corresponding formal and actual parameters must have compatible datatypes.

A good programming practice is to use different names for actual and formal parameters.

In [Example 8-4](#), `emp_id` and `amount` are formal parameters and `emp_num` and `bonus` are the corresponding actual parameters.

Example 8-4 Formal Parameters and Actual Parameters

```

DECLARE
  emp_num NUMBER(6) := 120;
  bonus   NUMBER(6) := 100;
  merit   NUMBER(4) := 50;
  PROCEDURE raise_salary (emp_id NUMBER,
                          amount NUMBER) IS
    BEGIN
      UPDATE employees SET salary =
        salary + amount WHERE employee_id = emp_id;
    END raise_salary;
BEGIN
  -- Procedure call specifies actual parameters
  raise_salary(emp_num, bonus);
  -- Expressions can be used as parameters
  raise_salary(emp_num, merit + bonus);
END;
/

```

When you invoke a subprogram, PL/SQL evaluates each actual parameter and assigns its value to the corresponding formal parameter. If necessary, PL/SQL implicitly converts the datatype of the actual parameter to the datatype of the corresponding formal parameter before the assignment (this is why corresponding formal and actual

parameters must have compatible datatypes). For information about implicit conversion, see ["Implicit Conversion"](#) on page 3-27.

A good programming practice is to avoid implicit conversion, either by using explicit conversion (explained in ["Explicit Conversion"](#) on page 3-26) or by declaring the variables that you intend to use as actual parameters with the same datatypes as their corresponding formal parameters. For example, suppose that `pkg` has this specification:

```
PACKAGE pkg IS
  PROCEDURE s (n IN PLS_INTEGER);
END pkg;
```

The following invocation of `pkg.s` avoids implicit conversion:

```
DECLARE
  y PLS_INTEGER :=1;
BEGIN
  pkg.s(y);
END;
```

The following invocation of `pkg.s` causes implicit conversion:

```
DECLARE
  y INTEGER :=1;
BEGIN
  pkg.s(y);
END;
```

Note: The specifications of many of Oracle's supplied packages and types declare formal parameters with the following 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's implementation of the supplied packages types.

Using Positional, Named, or Mixed Notation for PL/SQL Subprogram Parameters

When invoking a subprogram, you can specify the actual parameters using either positional, named, or mixed notation. [Table 8-2](#) compares these notations.

Table 8-2 PL/SQL Subprogram Parameter Notations

Notation	Description	Usage Notes
Positional	Specify the same parameters in the same order as the procedure declares them.	Compact and readable, but has these disadvantages: <ul style="list-style-type: none"> ▪ If you specify the parameters (especially literals) in the wrong order, the bug can be hard to detect. ▪ If the procedure's parameter list changes, you must change your code.

Table 8–2 (Cont.) PL/SQL Subprogram Parameter Notations

Notation	Description	Usage Notes
Named	Specify the name and value of each parameter, using the association operator, =>. Order of parameters is insignificant.	More verbose than positional notation, but easier to read and maintain. You can sometimes avoid changing your code if the procedure's parameter list changes (for example, if parameters are reordered or a new optional parameter is added). Safer than positional notation when you invoke an API that you did not define, or define an API for others to use.
Mixed	Start with positional notation, then use named notation for the remaining parameters.	Recommended when you invoke procedures that have required parameters followed by optional parameters, and you need to specify only a few of the optional parameters.

[Example 8–5](#) shows equivalent subprogram calls using positional, named, and mixed notation.

Example 8–5 Subprogram Calls Using Positional, Named, and Mixed Notation

```

DECLARE
    emp_num NUMBER(6) := 120;
    bonus   NUMBER(6) := 50;
    PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS
        BEGIN
            UPDATE employees SET salary =
                salary + amount WHERE employee_id = emp_id;
        END raise_salary;
BEGIN
    -- Positional notation:
    raise_salary(emp_num, bonus);
    -- Named notation (parameter order is insignificant):
    raise_salary(amount => bonus, emp_id => emp_num);
    raise_salary(emp_id => emp_num, amount => bonus);
    -- Mixed notation:
    raise_salary(emp_num, amount => bonus);
END;
/

CREATE OR REPLACE FUNCTION compute_bonus (emp_id NUMBER, bonus NUMBER)
RETURN NUMBER
IS
    emp_sal NUMBER;
BEGIN
    SELECT salary INTO emp_sal
    FROM employees
    WHERE employee_id = emp_id;
    RETURN emp_sal + bonus;
END compute_bonus;
/
    
```

The following equivalent SELECT statements invoke the PL/SQL subprogram in [Example 8–5](#) using positional, named, and mixed notation:

```

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
    
```

Specifying Subprogram Parameter Modes

Parameter modes define the action of formal parameters. The three parameter modes are `IN` (the default), `OUT`, and `IN OUT`.

Any parameter mode can be used with any subprogram. Avoid using the `OUT` and `IN OUT` modes with functions. To have a function return multiple values is poor programming practice. Also, make functions free from side effects, which change the values of variables not local to the subprogram.

Topics:

- [Using IN Mode](#)
- [Using OUT Mode](#)
- [Using IN OUT Mode](#)
- [Summary of Subprogram Parameter Modes](#)

Using IN Mode

An `IN` parameter lets you pass a value to the subprogram being invoked. Inside the subprogram, an `IN` parameter acts like a constant. It cannot be assigned a value.

You can pass a constant, literal, initialized variable, or expression as an `IN` parameter.

An `IN` parameter can be initialized to a default value, which is used if that parameter is omitted from the subprogram call. For more information, see "[Using Default Values for Subprogram Parameters](#)" on page 8-10.

Using OUT Mode

An `OUT` parameter returns a value to the caller of a subprogram. Inside the subprogram, an `OUT` parameter acts like a variable. You can change its value and reference the value after assigning it (see [Example 8-6](#)).

Example 8-6 Using OUT Mode

```
DECLARE
  emp_num          NUMBER(6) := 120;
  bonus            NUMBER(6) := 50;
  emp_last_name    VARCHAR2(25);
  PROCEDURE raise_salary (emp_id IN NUMBER, amount IN NUMBER,
                          emp_name OUT VARCHAR2) IS
  BEGIN
    UPDATE employees SET salary =
      salary + amount WHERE employee_id = emp_id;
    SELECT last_name INTO emp_name
      FROM employees
      WHERE employee_id = emp_id;
  END raise_salary;
BEGIN
  raise_salary(emp_num, bonus, emp_last_name);
  DBMS_OUTPUT.PUT_LINE
    ('Salary was updated for: ' || emp_last_name);
END;
/
```

You must pass a variable, not a constant or an expression, to an `OUT` parameter. Its previous value is lost unless you specify the `NOCOPY` keyword or the subprogram exits with an unhandled exception. See "[Using Default Values for Subprogram Parameters](#)" on page 8-10.

The initial value of an `OUT` parameter is `NULL`; therefore, the datatype of an `OUT` parameter cannot be a subtype defined as `NOT NULL`, such as the built-in subtype `NATURALN` or `POSITIVEN`. Otherwise, when you invoke the subprogram, PL/SQL raises `VALUE_ERROR`.

Before exiting a subprogram, assign values to all `OUT` formal parameters. Otherwise, the corresponding actual parameters will be null. If you exit successfully, PL/SQL assigns values to the actual parameters. If you exit with an unhandled exception, PL/SQL does not assign values to the actual parameters.

Using IN OUT Mode

An `IN OUT` parameter passes an initial value to a subprogram and returns an updated value to the caller. It can be assigned a value and its value can be read. Typically, an `IN OUT` parameter is a string buffer or numeric accumulator, that is read inside the subprogram and then updated.

The actual parameter that corresponds to an `IN OUT` formal parameter must be a variable, not a constant or an expression.

If you exit a subprogram successfully, PL/SQL assigns values to the actual parameters. If you exit with an unhandled exception, PL/SQL does not assign values to the actual parameters.

Summary of Subprogram Parameter Modes

Table 8-3 summarizes the characteristics of parameter modes.

Table 8-3 Parameter Modes

IN	OUT	IN OUT
The default	Must be specified	Must be specified
Passes a value to the subprogram	Returns a value to the caller	Passes an initial value to the subprogram and returns an updated value to the caller
Formal parameter acts like a constant	Formal parameter acts like an uninitialized variable	Formal parameter acts like an initialized variable
Formal parameter cannot be assigned a value	Formal parameter must be assigned a value	Formal parameter should be assigned a value
Actual parameter can be a constant, initialized variable, literal, or expression	Actual parameter must be a variable	Actual parameter must be a variable
Actual parameter is passed by reference (the caller passes the subprogram a pointer to the value)	Actual parameter is passed by value (the subprogram passes the caller a copy of the value) unless <code>NOCOPY</code> is specified	Actual parameter is passed by value (the caller passes the subprogram a copy of the value and the subprogram passes the caller a copy of the value) unless <code>NOCOPY</code> is specified

Using Default Values for Subprogram Parameters

By initializing formal `IN` parameters to default values, you can pass different numbers of actual parameters to a subprogram, accepting the default values for omitted actual parameters. You can also add new formal parameters without having to change every call to the subprogram.

If an actual parameter is omitted, the default value of its corresponding formal parameter is used.

You cannot skip a formal parameter by omitting its actual parameter. To omit the first parameter and specify the second, use named notation (see ["Using Positional, Named, or Mixed Notation for PL/SQL Subprogram Parameters"](#) on page 8-7).

You cannot assign NULL to an uninitialized formal parameter by omitting its actual parameter. You must either assign NULL as a default value or pass NULL explicitly.

[Example 8-7](#) illustrates the use of default values for subprogram parameters.

Example 8-7 Procedure with Default Parameter Values

```

DECLARE
    emp_num NUMBER(6) := 120;
    bonus   NUMBER(6);
    merit   NUMBER(4);
    PROCEDURE raise_salary (emp_id IN NUMBER,
        amount IN NUMBER DEFAULT 100,
                               extra IN NUMBER DEFAULT 50) IS
    BEGIN
        UPDATE employees SET salary = salary + amount + extra
            WHERE employee_id = emp_id;
    END raise_salary;
BEGIN
    -- Same as raise_salary(120, 100, 50)
    raise_salary(120);
    -- Same as raise_salary(120, 100, 25)
    raise_salary(emp_num, extra => 25);
END;
/

```

If the default value of a formal parameter is an expression, and you provide a corresponding actual parameter when you invoke the subprogram, the expression is not evaluated (see [Example 8-8](#)).

Example 8-8 Formal Parameter with Expression as Default Value

```

DECLARE
    cnt pls_integer := 0;
    FUNCTION dflt RETURN pls_integer IS
    BEGIN
        cnt := cnt + 1;
        RETURN 42;
    END dflt;
    -- Default is expression
    PROCEDURE p(i IN pls_integer DEFAULT dflt()) IS
    BEGIN
        DBMS_Output.Put_Line(i);
    END p;
BEGIN
    FOR j IN 1..5 LOOP
        p(j); -- Actual parameter is provided
    END loop;
    DBMS_Output.Put_Line('cnt: '||cnt);
    p(); -- Actual parameter is not provided
    DBMS_Output.Put_Line('cnt: '||cnt);
END;

```

The output of [Example 8-8](#) is:

```

1
2
3
4
5
Cnt: 0
42
Cnt: 1

```

Overloading PL/SQL Subprogram Names

PL/SQL lets you overload local subprograms, packaged subprograms, and type methods. You can use the same name for several different subprograms as long as their formal parameters differ in number, order, or datatype family.

[Example 8–9](#) defines two subprograms with the same name, `initialize`. The procedures initialize different types of collections. Because the processing in these two procedures is the same, it is logical to give them the same name.

You can place the two `initialize` procedures in the same block, subprogram, package, or object type. 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 8–9 Overloading a Subprogram Name

```

DECLARE
  TYPE date_tab_typ IS TABLE OF DATE   INDEX BY PLS_INTEGER;
  TYPE num_tab_typ  IS TABLE OF NUMBER INDEX BY PLS_INTEGER;

  hiredate_tab  date_tab_typ;
  sal_tab       num_tab_typ;

  PROCEDURE initialize (tab OUT date_tab_typ, n INTEGER) IS
  BEGIN
    FOR i IN 1..n LOOP
      tab(i) := SYSDATE;
    END LOOP;
  END initialize;

  PROCEDURE initialize (tab OUT num_tab_typ, n INTEGER) IS
  BEGIN
    FOR i IN 1..n LOOP
      tab(i) := 0.0;
    END LOOP;
  END initialize;

BEGIN
  initialize(hiredate_tab, 50); -- Invokes first (date_tab_typ) version
  initialize(sal_tab, 100);    -- Invokes second (num_tab_typ) version
END;
/

```

For an example of an overloaded procedure in a package, see [Example 10–3](#) on page 10-6.

Topics:

- [Guidelines for Overloading with Numeric Types](#)
- [Restrictions on Overloading](#)

- [When Compiler Catches Overloading Errors](#)

Guidelines for Overloading with Numeric Types

You can overload subprograms if their formal parameters differ only in numeric datatype. This technique is useful in writing mathematical application programming interfaces (APIs), because several versions of a function can use the same name, and each can accept a different numeric type. For example, a function that accepts `BINARY_FLOAT` might be faster, while a function that accepts `BINARY_DOUBLE` might provide more precision.

To avoid problems or unexpected results passing parameters to such overloaded subprograms:

- Ensure that the expected version of a subprogram is invoked for each set of expected parameters. For example, if you have overloaded functions that accept `BINARY_FLOAT` and `BINARY_DOUBLE`, which is invoked if you pass a `VARCHAR2` literal such as '5.0'?
- Qualify numeric literals and use conversion functions to make clear what the intended parameter types are. For example, use literals such as `5.0 f` (for `BINARY_FLOAT`), `5.0 d` (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 datatype)
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 datatype before passing it to the `SQRT` function.

If PL/SQL must convert a parameter to another datatype, it first tries to convert it to a higher datatype. 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`.

Restrictions on Overloading

You cannot overload the following subprograms:

- Standalone subprograms
- Subprograms whose formal parameters differ only in mode; for example:

```
PACKAGE pkg IS
  PROCEDURE s (p IN VARCHAR2);
  PROCEDURE s (p OUT VARCHAR2);
END pkg;
```

- Subprograms whose formal parameters differ only in subtype; for example:

```
PACKAGE pkg IS
  PROCEDURE s (p INTEGER);
  PROCEDURE s (p REAL);
END pkg;
```

`INTEGER` and `REAL` are subtypes of `NUMBER`, so they belong to the same datatype family.

- Functions that differ only in return value datatype, even if the datatypes are in different families; for example:

```
PACKAGE pkg IS
  FUNCTION f (p INTEGER) RETURN BOOLEAN;
  FUNCTION f (p INTEGER) RETURN INTEGER;
END pkg;
```

When Compiler Catches Overloading Errors

The PL/SQL compiler catches overloading errors as soon as it can determine that it will be unable to tell which subprogram was invoked. When subprograms have identical headings, the compiler catches the overloading error when you try to compile the subprograms themselves (if they are local) or when you try to compile the package specification that declares them (if they are packaged); otherwise, it catches the error when you try to compile an ambiguous invocation of a subprogram.

When you try to compile the package specification in [Example 8–10](#), which declares subprograms with identical headings, you get compile-time error PLS-00305.

Example 8–10 Package Specification with Overloading Violation that Causes Compile-Time Error

```
PACKAGE pkg1 IS
  PROCEDURE s (p VARCHAR2);
  PROCEDURE s (p VARCHAR2);
END pkg1;
```

Although the package specification in [Example 8–11](#) violates the rule that you cannot overload subprograms whose formal parameters differ only in subtype, you can compile it without error.

Example 8–11 Package Specification with Overloading Violation that Compiles Without Error

```
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`, such as the one in [Example 8–12](#), you get compile-time error PLS-00307.

Example 8–12 Invocation of Improperly Overloaded Subprogram

```
PROCEDURE p IS
  a pkg.t1 := 'a';
BEGIN
  pkg.s(a) -- Causes compile-time error PLS-00307;
END p;
```

Suppose that you correct the overloading violation in [Example 8–11](#) by giving the formal parameters of the overloaded subprograms different names, as follows:

```
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. For example:

```
PROCEDURE p IS
  a pkg.t1 := 'a';
BEGIN
  pkg.s(p1=>a); -- Compiles without error
END p;
```

If you specify the actual parameter with positional notation, as in [Example 8–12](#), you still get compile-time error PLS-00307.

The package specification in [Example 8–13](#) violates no overloading 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 8–14](#).

Example 8–13 Package Specification Without Overloading Violations

```
PACKAGE pkg3 IS
  PROCEDURE s (p1 VARCHAR2);
  PROCEDURE s (p1 VARCHAR2, p2 VARCHAR2 := 'p2');
END pkg3;
```

Example 8–14 Improper Invocation of Properly Overloaded Subprogram

```
PROCEDURE p IS
  a1 VARCHAR2(10) := 'a1';
  a2 VARCHAR2(10) := 'a2';
BEGIN
  pkg.s(p1=>a1, p2=>a2); -- Compiles without error
  pkg.s(p1=>a1);        -- Causes compile-time error PLS-00307
```

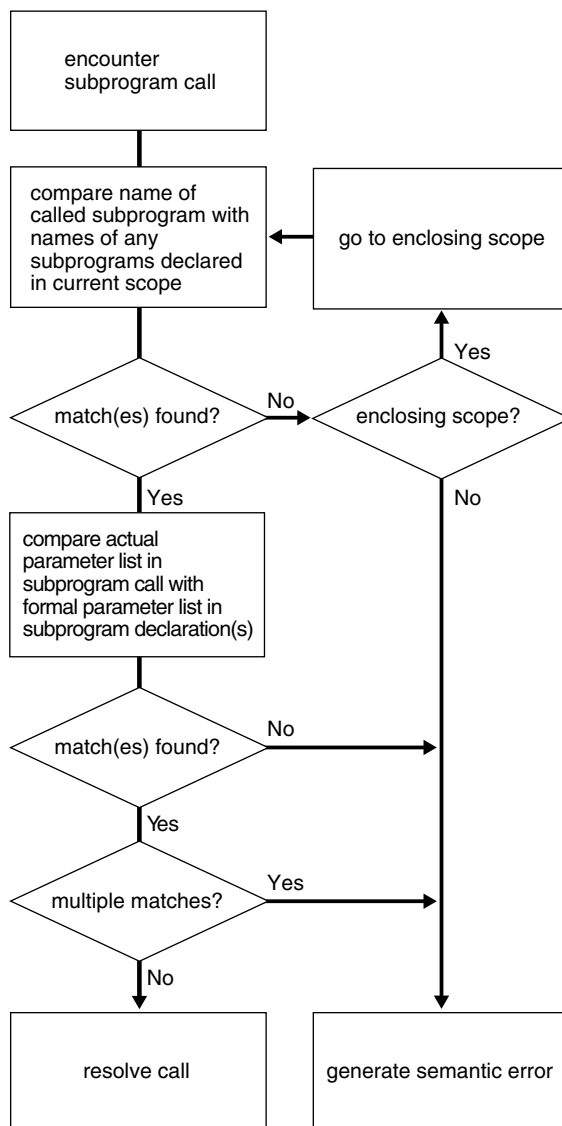
```
END p;
```

How PL/SQL Subprogram Calls Are Resolved

Figure 8–1 shows how the PL/SQL compiler resolves subprogram calls. When the compiler encounters a subprogram call, it tries to find a declaration that matches the call. The compiler searches first in the current scope and then, if necessary, in successive enclosing scopes. The compiler looks more closely when it finds one or more subprogram declarations in which the subprogram name matches the name of the called subprogram.

To resolve a call among possibly like-named subprograms at the same level of scope, the compiler must find an exact match between the actual and formal parameters. They must match in number, order, and datatype (unless some formal parameters were assigned default values). If no match is found or if multiple matches are found, the compiler generates a semantic error.

Figure 8–1 How the PL/SQL Compiler Resolves Calls



[Example 8-15](#) invokes the enclosing procedure `swap` from the function `balance`, generating an error because neither declaration of `swap` within the current scope matches the procedure call.

Example 8-15 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
    DBMS_OUTPUT.PUT_LINE('The following raises an error');
    -- swap(num1, num2);
    -- wrong number or types of arguments in call to 'SWAP'
    RETURN x;
  END balance;
  BEGIN NULL;END swap;
BEGIN
  NULL;
END;
/
```

Using Invoker's Rights or Definer's Rights (AUTHID Clause)

By default, a stored procedure or SQL method executes with the privileges of its owner, not its current user. Such definer's rights (DR) subprograms are bound to the schema in which they reside, allowing you to refer to objects in the same schema without qualifying their names. For example, if schemas `HR` and `OE` both have a table called `departments`, a subprogram owned by `HR` can refer to `departments` rather than `HR.departments`. If user `OE` invokes `HR`'s subprogram, the subprogram still accesses the `departments` table owned by `HR`.

If you compile the same subprogram in both schemas, you can define the schema name as a variable in `SQL*Plus` and refer to the table like `&schema.departments`. The code is portable, but if you change it, you must recompile it in each schema.

A more maintainable way is to create the subprogram with the `AUTHID CURRENT_USER` clause, which makes it execute with the privileges and schema context of the calling user. You can create one instance of the subprogram, and many users can invoke it to access their own data.

Such invoker's-rights (IR) subprograms are not bound to a particular schema. In [Example 8-16](#), the procedure `create_dept` executes with the privileges of the calling user and inserts rows into that user's `departments` table:

Example 8-16 Specifying Invoker's Rights with a Procedure

```
CREATE OR REPLACE PROCEDURE create_dept (
  v_deptno NUMBER,
  v_dname VARCHAR2,
  v_mgr NUMBER,
  v_loc NUMBER)
AUTHID CURRENT_USER AS
BEGIN
  INSERT INTO departments
  VALUES (v_deptno, v_dname, v_mgr, v_loc);
```

```
END;  
/  
CALL create_dept(44, 'Information Technology', 200, 1700);
```

Topics:

- [Advantages of Invoker's Rights](#)
- [Specifying Subprogram Privileges \(AUTHID Clause\)](#)
- [Who Is the Current User During Subprogram Execution?](#)
- [How External References Are Resolved in IR Subprograms](#)
- [Need for Template Objects in IR Subprograms](#)
- [Overriding Default Name Resolution in IR Subprograms](#)
- [Granting Privileges to IR Subprograms](#)
- [Using Views and Database Triggers with IR Subprograms](#)
- [Using Database Links with IR Subprograms](#)
- [Using Object Types with IR Subprograms](#)
- [Invoking IR Instance Methods](#)

Advantages of Invoker's Rights

Invoker's-rights (IR) subprograms let you reuse code and centralize application logic. They are especially useful in applications that store data using identical tables in different schemas. All the schemas in one instance can invoke procedures owned by a central schema. You can even have schemas in different instances invoke centralized procedures using a database link.

Consider a company that uses a stored procedure to analyze sales. If the company has several schemas, each with a similar `SALES` table, it usually also needs several copies of the stored procedure, one in each schema.

To solve the problem, the company installs an IR version of the stored procedure in a central schema. Now, all the other schemas can invoke the same procedure, which queries the appropriate `SALES` table in each case.

You can restrict access to sensitive data by invoking, from an IR subprogram, a DR subprogram that queries or updates the table containing the sensitive data. Although multiple users can invoke the IR subprogram, they do not have direct access to the sensitive data.

Specifying Subprogram Privileges (AUTHID Clause)

To implement invoker's rights, create the subprogram with the `AUTHID CURRENT_USER` clause, which specifies that the subprogram executes with the privileges of its current user. (The default is `AUTHID DEFINER`, which specifies that the subprogram executes with the privileges of its owner.) The `AUTHID` clause also specifies whether external references (that is, references to objects outside the subprogram) are resolved in the schema of the owner or the current user.

The `AUTHID` clause is allowed only in the header of a standalone subprogram, a package spec, or an object type spec. In the `CREATE FUNCTION`, `CREATE PROCEDURE`, `CREATE PACKAGE`, or `CREATE TYPE` statement, you can include either `AUTHID CURRENT_USER` or `AUTHID DEFINER` immediately before the `IS` or `AS` keyword that begins the declaration section.

In a package or object type, the AUTHID clause applies to all subprograms.

Most supplied PL/SQL packages (such as DBMS_LOB, DBMS_PIPE, DBMS_ROWID, DBMS_SQL, and UTL_REF) are IR packages.

Who Is the Current User During Subprogram Execution?

Initially, the current user is the session user. When an IR subprogram is invoked, the current user does not change. When a DR subprogram is invoked, the owner of that subprogram becomes the current user. Note that when an IR subprogram is invoked from a DR subprogram, the current user is the owner of the DR subprogram, not the session user.

To verify who the current user is at any time, you can check the static data dictionary view USER_USERS. Inside an IR subprogram, the value from this view might be different from the value of the USER built-in function, which always returns the name of the session user. In addition, the current user setting can be checked by invoking the built-in function SYS_CONTEXT('USERENV', 'CURRENT_USER').

How External References Are Resolved in IR Subprograms

External references in an IR subprogram are resolved as follows:

- In the following statements, external references are resolved in the schema of the current user, whose privileges are checked at run time:
 - Data manipulation statements SELECT, INSERT, UPDATE, and DELETE
 - Transaction control statement LOCK TABLE
 - Cursor-control statements OPEN and OPEN-FOR
 - Dynamic SQL statements EXECUTE IMMEDIATE and OPEN-FOR-USING
 - SQL statements parsed with DBMS_SQL.PARSE procedure
- In all other statements, external references are resolved in the schema of the owner, whose privileges are checked at compile time.
- External references to constants follow the preceding rules only if the constants are not inlined.

One optimization that the PL/SQL compiler can perform is **constant inlining**. Constant inlining replaces a reference to a constant with its value. The value of an inlined constant becomes part of the calling program, and is therefore available to it. Outside the calling program, however, the caller cannot access the constant.

In [Example 8–17](#), the IR procedure `above_salary` includes two external references to the function `num_above_salary` in the package `emp_actions` in [Example 1–14](#). The external reference in the SELECT statement is resolved in the schema of the current user. The external reference in the assignment statement is resolved in the schema of the owner of the procedure `above_salary`.

Example 8–17 Resolving External References in IR Subprogram

```
CREATE PROCEDURE above_salary
  (emp_id IN NUMBER)
  AUTHID CURRENT_USER
AS
  emps NUMBER;
BEGIN
  -- External reference is resolved in schema of current user:
```

```

SELECT * FROM employees
WHERE employee_id = emp_actions.num_above_salary(emp_id);

-- External reference is resolved in schema of owner of above_salary:
emps := emp_actions.num_above_salary(emp_id);
DBMS_OUTPUT.PUT_LINE
('Number of employees with higher salary: ' || TO_CHAR(emps));
END;
/
CALL above_salary(120);
/

```

In [Example 8-18](#), the IR procedure `test` is owned by `schema_a` and has an external reference to the constant `c`. When compiling `test`, the PL/SQL compiler inlines the external reference to `c`, replacing it with the value of `c` (which is one). The value of `c` (one) becomes part of the procedure `test`. Therefore, when `schema_b` invokes `test`, the external reference to `c` within `test` succeeds. However, `schema_b` cannot access `c` directly.

Example 8-18 Inlined External Reference to Constant in IR Subprogram

```

-- In schema_a:
CREATE OR REPLACE PACKAGE pkg IS
  c CONSTANT NUMBER := 1;
END;
/
CREATE OR REPLACE PACKAGE c_test AUTHID CURRENT_USER IS
  PROCEDURE test; -- IR procedure owned by schema_a
END;
/
CREATE OR REPLACE PACKAGE BODY c_test IS
  PROCEDURE test IS
    n NUMBER;
  BEGIN
    -- External reference to constant c in package pkg:
    SELECT schema_a.pkg.c INTO n FROM DUAL;
    -- If external reference succeeds:
    DBMS_OUTPUT.PUT_LINE ('c_test: referenced schema_a.pkg.c');
    -- If external reference fails:
    EXCEPTION WHEN OTHERS THEN
      DBMS_OUTPUT.PUT_LINE
        ('c_test: caught exception referencing schema_a.pkg.c: '
         || SQLCODE || ': ' || SQLERRM);
  END;
END;
/
-- Allow schema_b to execute package c_test:
GRANT EXECUTE ON schema_a.c_test TO schema_b;

-- In schema_b, invoke procedure test in package c_test:
SQL> BEGIN schema_a.c_test.test; END;
2 /
-- External reference to constant c from inside procedure test succeeds:
c_test: referenced schema_a.pkg.c

PL/SQL procedure successfully completed.

-- External reference to constant c from outside procedure test fails:
SQL> DECLARE n NUMBER; BEGIN n := schema_a.pkg.c; END;
2 /

```



```

DECLARE n NUMBER; BEGIN n := schema_a.pkg.c; END;
                                *
ERROR at line 1:
ORA-06550: line 1, column 30:
PLS-00201: identifier 'schema_a.pkg' must be declared
ORA-06550: line 1, column 25:
PL/SQL: Statement ignored

```

Need for Template Objects in IR Subprograms

The PL/SQL compiler must resolve all references to tables and other objects at compile time. The owner of an IR subprogram 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 subprogram.

Overriding Default Name Resolution in IR Subprograms

If you want an unqualified name to refer to a particular schema, rather than to the schema of the invoker, create a public synonym for the object; for example:

```
CREATE PUBLIC SYNONYM emp FOR hr.employees;
```

When the IR subprogram refers to `emp`, it matches the public synonym, which resolves to `hr.employees`.

Note: If the invoking schema has an object or private synonym named `emp`, fully qualify the reference in the IR subprogram.

Granting Privileges to IR Subprograms

To invoke a subprogram directly, users must have the `EXECUTE` privilege on that subprogram. By granting the privilege, you allow a user to:

- Invoke the subprogram directly
- Compile functions and procedures that invoke the subprogram

For external references resolved in the current user's schema (such as those in DML statements), the current user must have the privileges needed to access schema objects referenced by the subprogram. For all other external references (such as function calls), the owner's privileges are checked at compile time, and no run-time check is done.

A DR subprogram operates under the security domain of its owner, no matter who is executing it. The owner must have the privileges needed to access schema objects referenced by the subprogram.

You can write a program consisting of multiple subprograms, some with definer's rights and others with invoker's rights. Then, you can use the `EXECUTE` privilege to restrict program entry points. That way, users of an entry-point subprogram can execute the other subprograms indirectly but not directly.

Example: Granting Privileges on an IR Subprogram

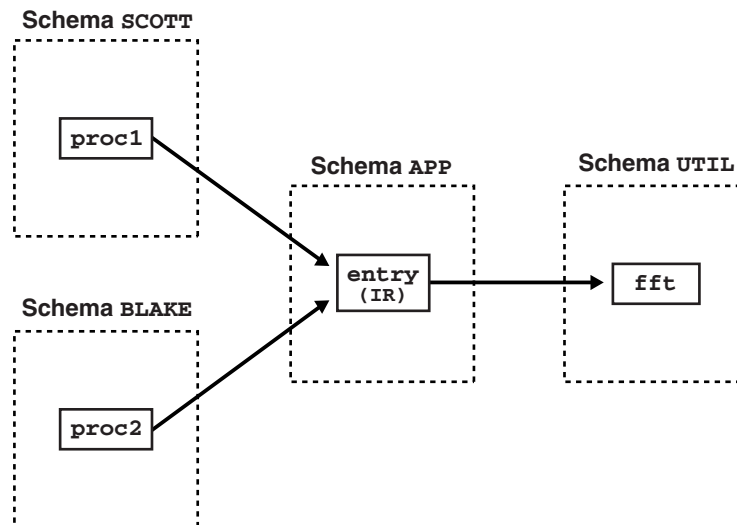
Suppose user `UTIL` grants the `EXECUTE` privilege on subprogram `FFT` to user `APP`:

```
GRANT EXECUTE ON util.fft TO app;
```

Now, user APP can compile functions and procedures that invoke subprogram FFT. At run time, no privilege checks on the calls are done. As [Figure 8–2](#) shows, user UTIL need not grant the EXECUTE privilege to every user who might invoke FFT indirectly.

Since subprogram `util.fft` is invoked directly only from IR subprogram `app.entry`, user `util` must grant the EXECUTE privilege only to user APP. When `util.fft` is executed, its current user can be APP, SCOTT, or BLAKE even though SCOTT and BLAKE were not granted the EXECUTE privilege.

Figure 8–2 Indirect Calls to an IR Subprogram



Using Views and Database Triggers with IR Subprograms

For IR subprograms executed within a view expression, the schema that created the view, not the schema that is querying the view, is considered to be the current user. This rule also applies to database triggers.

Using Database Links with IR Subprograms

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, Oracle uses the username of the current user (who must be a global user). Suppose an IR subprogram owned by user OE references the following database link. If global user HR invokes the subprogram, it connects to the Dallas database as user HR, who is the current user.

```
CREATE DATABASE LINK dallas CONNECT TO CURRENT_USER USING ...
```

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.

Using Object Types with IR Subprograms

To define object types for use in any schema, specify the `AUTHID CURRENT_USER` clause. For information on object types, see *Oracle Database Object-Relational Developer's Guide*.

Suppose user HR creates the following object type:

Example 8–19 Creating an Object Type with AUTHID CURRENT USER

```
CREATE TYPE person_typ AUTHID CURRENT_USER AS OBJECT (
  person_id NUMBER,
  person_name VARCHAR2(30),
  person_job VARCHAR2(10),
  STATIC PROCEDURE new_person_typ (
    person_id NUMBER, person_name VARCHAR2, person_job VARCHAR2,
    schema_name VARCHAR2, table_name VARCHAR2),
  MEMBER PROCEDURE change_job (SELF IN OUT NOCOPY person_typ,
    new_job VARCHAR2)
);
/
CREATE TYPE BODY person_typ AS
  STATIC PROCEDURE new_person_typ (
    person_id NUMBER, person_name VARCHAR2, person_job VARCHAR2,
    schema_name VARCHAR2, table_name VARCHAR2) IS
    sql_stmt VARCHAR2(200);
  BEGIN
    sql_stmt := 'INSERT INTO ' || schema_name || '.'
      || table_name || ' VALUES (HR.person_typ(:1, :2, :3))';
    EXECUTE IMMEDIATE sql_stmt
      USING person_id, person_name, person_job;
  END;
  MEMBER PROCEDURE change_job (SELF IN OUT NOCOPY person_typ,
    new_job VARCHAR2) IS
  BEGIN
    person_job := new_job;
  END;
END;
/
```

Then, user HR grants the `EXECUTE` privilege on object type `person_typ` to user OE:

```
GRANT EXECUTE ON person_typ TO OE;
```

Finally, user OE creates an object table to store objects of type `person_typ`, then invokes procedure `new_person_typ` to populate the table:

```
CONNECT OE/password;
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');
                                'oe', 'person_tab');
END;
/

```

The calls succeed because the procedure executes with the privileges of its current user (OE), not its owner (HR).

For subtypes in an object type hierarchy, the following 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.

Invoking IR Instance Methods

An IR instance method executes with the privileges of the invoker, not the creator of the instance. Suppose that `person_typ` is an IR object type as created in [Example 8–19](#), and that 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 shown in [Example 8–20](#).

Example 8–20 Invoking an IR Instance Methods

```

-- oe creates a procedure that invokes change_job
CREATE PROCEDURE reassign
  (p IN OUT NOCOPY hr.person_typ, new_job VARCHAR2) AS
BEGIN
  p.change_job(new_job); -- executes with the privileges of oe
END;
/
-- OE grants EXECUTE to HR on procedure reassign
GRANT EXECUTE ON reassign to HR;
CONNECT HR/password

-- user hr passes a person_typ object to the 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;
/

```

Using Recursive PL/SQL Subprograms

A recursive subprogram is one that invokes itself. Each recursive call creates a new instance of any items declared in the subprogram, including parameters, variables, cursors, and exceptions. Likewise, new instances of SQL statements are created at each level in the recursive descent.

Be careful where you place a recursive call. If you place it inside a cursor FOR loop or between OPEN and CLOSE statements, another cursor is opened at each call, which might exceed the limit set by the Oracle initialization parameter `OPEN_CURSORS`.

There must be at least two paths through a recursive subprogram: one that leads to the recursive call and one that does not. At least one path must lead to a terminating condition. Otherwise, the recursion continues until PL/SQL runs out of memory and raises the predefined exception `STORAGE_ERROR`.

Recursion is a powerful technique for simplifying the design of algorithms. Basically, recursion means self-reference. In a recursive mathematical sequence, each term is derived by applying a formula to preceding terms. The Fibonacci sequence (0, 1, 1, 2, 3, 5, 8, 13, 21, ...), is an example. Each term in the sequence (after the second) is the sum of the two terms that immediately precede it.

In a recursive definition, something is defined as simpler versions of itself. Consider the definition of n factorial ($n!$), the product of all integers from 1 to n :

$$n! = n * (n - 1)!$$

Invoking External Subprograms

Although PL/SQL is a powerful, flexible language, some tasks are more easily done in another language. Low-level languages such as C are very fast. Widely used languages such as Java have reusable libraries for common design patterns.

You can use PL/SQL call specifications to invoke external subprograms written in other languages, making their capabilities and libraries available from PL/SQL. For example, you can invoke Java stored procedures from any PL/SQL block, subprogram, or package. For more information about Java stored procedures, see *Oracle Database Java Developer's Guide*.

If the following Java class is stored in the database, it can be invoked as shown in [Example 8-21](#).

```
import java.sql.*;
import oracle.jdbc.driver.*;
public class Adjuster {
    public static void raiseSalary (int empNo, float percent)
        throws SQLException {
        Connection conn = new OracleDriver().defaultConnection();
        String sql = "UPDATE employees SET salary = salary * ?
                    WHERE employee_id = ?";
        try {
            PreparedStatement pstmt = conn.prepareStatement(sql);
            pstmt.setFloat(1, (1 + percent / 100));
            pstmt.setInt(2, empNo);
            pstmt.executeUpdate();
            pstmt.close();
        } catch (SQLException e)
            {System.err.println(e.getMessage());}
    }
}
```

The class `Adjuster` has one method, which raises the salary of an employee by a given percentage. Because `raiseSalary` is a void method, you publish it as a procedure using the call specification shown in [Example 8-21](#) and then can invoke the procedure `raise_salary` from an anonymous PL/SQL block.

Example 8-21 Invoking an External Procedure from PL/SQL

```
CREATE OR REPLACE PROCEDURE raise_salary (empid NUMBER, pct NUMBER)
AS LANGUAGE JAVA
NAME 'Adjuster.raiseSalary(int, float)';
/
```

```

DECLARE
    emp_id NUMBER := 120;
    percent NUMBER := 10;
BEGIN
    -- get values for emp_id and percent
    raise_salary(emp_id, percent); -- invoke external subprogram
END;
/

```

Java call specifications cannot be declared as nested procedures, but can be specified in object type specifications, object type bodies, PL/SQL package specifications, PL/SQL package bodies, and as top level PL/SQL procedures and functions.

[Example 8–22](#) shows a call to a Java function from a PL/SQL procedure.

Example 8–22 Invoking a Java Function from PL/SQL

```

-- the following invalid nested Java call spec throws PLS-00999
-- CREATE PROCEDURE sleep (milli_seconds in number) IS
--     PROCEDURE java_sleep (milli_seconds IN NUMBER) AS ...

-- Create Java call spec, then call from PL/SQL procedure
CREATE PROCEDURE java_sleep (milli_seconds IN NUMBER)
    AS LANGUAGE JAVA NAME 'java.lang.Thread.sleep(long)';
/
CREATE PROCEDURE sleep (milli_seconds in number) IS
-- the following nested PROCEDURE spec is not legal
-- PROCEDURE java_sleep (milli_seconds IN NUMBER)
--     AS LANGUAGE JAVA NAME 'java.lang.Thread.sleep(long)';
BEGIN
    DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.get_time());
    java_sleep (milli_seconds);
    DBMS_OUTPUT.PUT_LINE(DBMS_UTILITY.get_time());
END;
/

```

External C subprograms are used to interface with embedded systems, solve engineering problems, analyze data, or control real-time devices and processes. External C subprograms extend the functionality of the database server, and move computation-bound programs from client to server, where they execute faster. For more information about external C subprograms, see *Oracle Database Advanced Application Developer's Guide*.

Controlling Side Effects of PL/SQL Subprograms

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

To be callable from SQL statements, a stored function (and any subprograms that it invokes) must obey the following purity rules, which are meant to control side effects:

- When invoked from a `SELECT` statement or a parallelized `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot modify any database tables.
- When invoked from an `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot query or modify any database tables modified by that statement.
- When invoked from a `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement, the function cannot execute SQL transaction control statements (such as `COMMIT`),

session control statements (such as `SET ROLE`), or system control statements (such as `ALTER SYSTEM`). Also, it cannot execute DDL statements (such as `CREATE`) because they are followed by an automatic commit.

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

To check for purity rule violations at compile time, use the `RESTRICT_REFERENCES` pragma to assert that a function does not read or write database tables or package variables (for syntax, see "[RESTRICT_REFERENCES Pragma](#)" on page 13-118).

In [Example 8-23](#), the `RESTRICT_REFERENCES` pragma asserts that packaged function `credit_ok` writes no database state (`WNDS`) and reads no package state (`RNPS`).

Example 8-23 `RESTRICT_REFERENCES` Pragma

```
CREATE PACKAGE loans AS
    FUNCTION credit_ok RETURN BOOLEAN;
    PRAGMA RESTRICT_REFERENCES (credit_ok, WNDS, RNPS);
END loans;
/
```

A static `INSERT`, `UPDATE`, or `DELETE` statement always violates `WNDS`, and if it reads columns, it also violates `RNDS` (reads no database state). A dynamic `INSERT`, `UPDATE`, or `DELETE` statement always violates both `WNDS` and `RNDS`.

Understanding PL/SQL Subprogram Parameter Aliasing

To optimize a subprogram call, the PL/SQL compiler can choose between two methods of parameter passing. With the `BY VALUE` method, the value of an actual parameter is passed to the subprogram. With the `BY REFERENCE` method, only a pointer to the value is passed; the actual and formal parameters reference the same item.

The `NOCOPY` compiler hint increases the possibility of aliasing (that is, having two different names refer to the same memory location). This can occur when a global variable appears as an actual parameter in a subprogram call and then is referenced within the subprogram. The result is indeterminate because it depends on the method of parameter passing chosen by the compiler.

In [Example 8-24](#), procedure `ADD_ENTRY` refers to varray `LEXICON` both as a parameter and as a global variable. When `ADD_ENTRY` is invoked, the identifiers `WORD_LIST` and `LEXICON` point to the same varray.

Example 8-24 *Aliasing from Passing Global Variable with `NOCOPY` Hint*

```
DECLARE
    TYPE Definition IS RECORD (
        word    VARCHAR2(20),
        meaning VARCHAR2(200));
    TYPE Dictionary IS VARRAY(2000) OF Definition;
    lexicon Dictionary := Dictionary();
    PROCEDURE add_entry (word_list IN OUT NOCOPY Dictionary) IS
    BEGIN
        word_list(1).word := 'aardvark';
        lexicon(1).word := 'aardwolf';
    END;
BEGIN
    lexicon.EXTEND;
    add_entry(lexicon);
```

```

    DBMS_OUTPUT.PUT_LINE(lexicon(1).word);
END;
/

```

The program prints `aardwolf` if the compiler obeys the `NOCOPY` hint. The assignment to `WORD_LIST` is done immediately through a pointer, then is overwritten by the assignment to `LEXICON`.

The program prints `aardvark` if the `NOCOPY` hint is omitted, or if the compiler does not obey the hint. The assignment to `WORD_LIST` uses an internal copy of the varray, which is copied back to the actual parameter (overwriting the contents of `LEXICON`) when the procedure ends.

Aliasing can also occur when the same actual parameter appears more than once in a subprogram call. In [Example 8–25](#), `n2` is an `IN OUT` parameter, so the value of the actual parameter is not updated until the procedure exits. That is why the first `PUT_LINE` prints 10 (the initial value of `n`) and the third `PUT_LINE` prints 20. However, `n3` is a `NOCOPY` parameter, so the value of the actual parameter is updated immediately. That is why the second `PUT_LINE` prints 30.

Example 8–25 Aliasing Passing Same Parameter Multiple Times

```

DECLARE
    n NUMBER := 10;
    PROCEDURE do_something (
        n1 IN NUMBER,
        n2 IN OUT NUMBER,
        n3 IN OUT NOCOPY NUMBER) IS
    BEGIN
        n2 := 20;
        DBMS_OUTPUT.put_line(n1); -- prints 10
        n3 := 30;
        DBMS_OUTPUT.put_line(n1); -- prints 30
    END;
BEGIN
    do_something(n, n, n);
    DBMS_OUTPUT.put_line(n); -- prints 20
END;
/

```

Because they are pointers, cursor variables also increase the possibility of aliasing. In [Example 8–26](#), after the assignment, `emp_cv2` is an alias of `emp_cv1`; both point to the same query work area. The first fetch from `emp_cv2` fetches the third row, not the first, because the first two rows were already fetched from `emp_cv1`. The second fetch from `emp_cv2` fails because `emp_cv1` is closed.

Example 8–26 Aliasing from Assigning Cursor Variables to Same Work Area

```

DECLARE
    TYPE EmpCurTyp IS REF CURSOR;
    c1 EmpCurTyp;
    c2 EmpCurTyp;
    PROCEDURE get_emp_data (emp_cv1 IN OUT EmpCurTyp,
                           emp_cv2 IN OUT EmpCurTyp) IS
        emp_rec employees%ROWTYPE;
    BEGIN
        OPEN emp_cv1 FOR SELECT * FROM employees;
        emp_cv2 := emp_cv1;
        FETCH emp_cv1 INTO emp_rec; -- fetches first row
        FETCH emp_cv1 INTO emp_rec; -- fetches second row

```



```

    FETCH emp_cv2 INTO emp_rec; -- fetches third row
    CLOSE emp_cv1;
    DBMS_OUTPUT.put_line('The following raises an invalid cursor');
--  FETCH emp_cv2 INTO emp_rec;
--  raises invalid cursor when get_emp_data is invoked
    END;
BEGIN
    get_emp_data(c1, c2);
END;
/

```

Using the Cross-Session PL/SQL Function Result Cache

The cross-session PL/SQL function result caching mechanism provides a language-supported and system-managed means for caching the results of PL/SQL functions in a shared global area (SGA), which is available to every session that runs your application. The caching mechanism is both efficient and easy to use, and it relieves you of the burden of designing and developing your own caches and cache-management policies.

To enable result-caching for a function, use the `RESULT_CACHE` clause. When a result-cached function is invoked, the system checks the cache. If the cache contains the result from a previous call to the function with the same parameter values, the system returns the cached result to the invoker and does not re-execute the function body. If the cache does not contain the result, the system executes the function body and adds the result (for these parameter values) to the cache before returning control to the 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.

You can specify the database objects that are used to compute a cached result, so that if any of them are updated, 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, do the following:

- In the function declaration, include the option `RESULT_CACHE`.
- In the function definition:

- Include the `RESULT_CACHE` clause.
- In the optional `RELIES_ON` clause, specify any tables or views on which the function results depend.

For the syntax of the `RESULT_CACHE` and `RELIES_ON` clauses, see ["Function Declaration and Definition"](#) on page 13-74.

In [Example 8–27](#), the package `PKG` declares and then defines a result-cached function, `get_dept_info`, which returns the average salary and number of employees in a given department. `get_dept_info` depends on the database table `EMPLOYEES`.

Example 8–27 Declaration and Definition of Result-Cached Function

```
-- Package specification
CREATE OR REPLACE PACKAGE department_pks IS
    TYPE dept_info_record IS RECORD (average_salary      NUMBER,
                                     number_of_employees NUMBER);
    -- Function declaration
    FUNCTION get_dept_info (dept_id NUMBER) RETURN dept_info_record
    RESULT_CACHE;
    ...
END department_pks;
/
CREATE OR REPLACE PACKAGE BODY department_pks AS
    -- Function definition
    FUNCTION get_dept_info (dept_id NUMBER) RETURN dept_info_record
    RESULT_CACHE RELIES_ON (EMPLOYEES)
    IS
        rec dept_info_record;
    BEGIN
        SELECT AVG(SALARY), COUNT(*) INTO result
            FROM EMPLOYEES
            WHERE DEPARTMENT_ID = dept_id;
        RETURN rec;
    END get_dept_info;
    ...
END department_pks;
/
DECLARE
    dept_id  NUMBER := 50;
    avg_sal  NUMBER;
    no_of_emp NUMBER;
BEGIN
    avg_sal := department_pks.get_dept_info(50).average_salary;
    no_of_emp := department_pks.get_dept_info(50).number_of_employees;
    DBMS_OUTPUT.PUT_LINE('dept_id:= ' || dept_id );
    DBMS_OUTPUT.PUT_LINE(' average_salary= ' || avg_sal);
    DBMS_OUTPUT.PUT_LINE(' Number_of_employees' || no_of_emp );
END;
/
```

You invoke the function `get_dept_info` as you invoke any function. For example, the following call returns the average salary and the number of employees in department number 10:

```
PKG.get_dept_info(10);
```

The following call returns only the average salary in department number 10:

```
PKG.get_dept_info(10).average_salary;
```

If the result for `get_dept_info(10)` is already in the result cache, the result is returned from the cache; otherwise, the result is computed and added to the cache. Because the `RELIES_ON` clause specifies `EMPLOYEES`, any update to `EMPLOYEES` invalidates all cached results for `PKG.get_dept_info`, relieving you of programming cache invalidation logic everywhere that `EMPLOYEES` might change.

Developing Applications with Result-Cached Functions

When developing an application that uses a result-cached function, make no assumptions about the number of times the body of the function will execute for a given set of parameter values.

Some situations in which the body of a result-cached function executes 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**
A cached result becomes invalid when any database object specified in the `RELIES_ON` clause of the function definition changes.
- 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 ["Bypassing the Result Cache"](#) on page 8-35)

Restrictions on Result-Cached Functions

To be result-cached, a function must meet all of the following 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 has no `OUT` or `IN OUT` parameters.
- No `IN` parameter has one of the following types:
 - BLOB
 - CLOB
 - NCLOB
 - REF CURSOR
 - Collection
 - Object
 - Record
- The return type is none of the following:
 - BLOB
 - CLOB
 - NCLOB
 - REF CURSOR
 - Object

- Record or PL/SQL collection that contains one of the preceding unsupported return types

It is recommended that a result-cached function also meet the following criteria:

- It has no side effects.
For example, it does not modify the database state, or modify the external state by invoking DBMS_OUTPUT or sending email.
- It does not depend on session-specific settings.
For more information, see ["Making Result-Cached Functions Handle Session-Specific Settings"](#) on page 8-35.
- It does not depend on session-specific application contexts.
For more information, see ["Making Result-Cached Functions Handle Session-Specific Application Contexts"](#) on page 8-36.

Examples of Result-Cached Functions

The best candidates for result-caching are functions that are invoked frequently but depend on information that changes infrequently (as might be the case in the first example). Result-caching avoids redundant computations in recursive functions.

Examples:

- [Result-Cached Application Configuration Parameters](#)
- [Result-Cached Recursive Function](#)

Result-Cached Application Configuration Parameters

Consider an application that has configuration parameters that can be set at either the global level, the application level, or the role level. The application stores the configuration information in the following tables:

```
-- Global Configuration Settings
CREATE TABLE global_config_params
  ( name VARCHAR2(20), -- parameter NAME
    value VARCHAR2(20), -- parameter VALUE
    PRIMARY KEY (name)
  )
/

-- Application-Level Configuration Settings
CREATE TABLE app_level_config_params
  ( app_id VARCHAR2(20), -- application ID
    name VARCHAR2(20), -- parameter NAME
    value VARCHAR2(20), -- parameter VALUE
    PRIMARY KEY (app_id, name)
  )
/

-- Role-Level Configuration Settings
CREATE TABLE role_level_config_params
  ( role_id VARCHAR2(20), -- application (role) ID
    name VARCHAR2(20), -- parameter NAME
    value VARCHAR2(20), -- parameter VALUE
    PRIMARY KEY (role_id, name)
  )
/
```

For each configuration parameter, the role-level setting overrides the application-level setting, which overrides the global setting. To determine which setting applies to a parameter, the application defines the PL/SQL function `get_value`. Given a parameter name, application ID, and role ID, `get_value` returns the setting that applies to the parameter.

The function `get_value` is a good candidate for result-caching if it is invoked frequently and if the configuration information changes infrequently. To ensure that a committed change to `global_config_params`, `app_level_config_params`, or `role_level_config_params` invalidates the cached results of `get_value`, include their names in the `RELIES_ON` clause.

[Example 8–28](#) shows a possible definition for `get_value`.

Example 8–28 Result-Cached Function that Returns Configuration Parameter Setting

```

FUNCTION get_value (p_param VARCHAR2,
                   p_app_id NUMBER,
                   p_role_id NUMBER) RETURN VARCHAR2
    RESULT_CACHE RELIES_ON (role_level_config_params,
                           app_level_config_params,
                           global_config_params)
IS
    answer VARCHAR2(20);
BEGIN
    -- Is parameter set at role level?
    BEGIN
        SELECT value INTO answer
        FROM   role_level_config_params
        WHERE  role_id = p_role_id
        AND    name = p_param;
        RETURN answer; -- Found
    EXCEPTION
        WHEN no_data_found THEN
            NULL; -- Fall through to following code
    END;
    -- Is parameter set at application level?
    BEGIN
        SELECT value INTO answer
        FROM   app_level_config_params
        WHERE  app_id = p_app_id
        AND    name = p_param;
        RETURN answer; -- Found
    EXCEPTION
        WHEN no_data_found THEN
            NULL; -- Fall through to following code
    END;
    -- Is parameter set at global level?
    SELECT value INTO answer
    FROM   global_config_params
    WHERE  name = p_param;
    RETURN answer;
END;
```

Result-Cached Recursive Function

A recursive function for finding the *n*th term of a Fibonacci series that mirrors the mathematical definition of the series might do many redundant computations. For example, to evaluate `fibonacci(7)`, the function must compute `fibonacci(6)` and

`fibonacci(5)`. To compute `fibonacci(6)`, the function must compute `fibonacci(5)` and `fibonacci(4)`. Therefore, `fibonacci(5)` and several other terms are computed redundantly. Result-caching avoids these redundant computations. A `RELIES_ON` clause is unnecessary.

```
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](#)
- [Bypassing the Result Cache](#)
- [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](#)
- [Managing the Result Cache](#)
- [Hot-Patching PL/SQL Program 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 is the same as 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 not the same. (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.

Bypassing the Result Cache

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 Program Units on Which Result-Cached Functions Depend"](#) on page 8-39).

- A session is performing a DML statement on a table or view that was specified in the RELIES_ON clause of a result-cached function. The session bypasses the result cache for that function until the DML statement is completed (either committed or rolled back), and then resumes using the cache for that function.

Cache bypass ensures the following:

- The user of each session sees his or her own uncommitted changes.
- The cross-session 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 the following function:

```
FUNCTION get_hire_date (emp_id NUMBER) RETURN VARCHAR
  RESULT_CACHE RELIES_ON (HR.EMPLOYEES) IS
  date_hired DATE;
BEGIN
  SELECT hire_date INTO date_hired
  FROM HR.EMPLOYEES
  WHERE EMPLOYEE_ID = emp_id;
  RETURN TO_CHAR(date_hired);
END;
```

The preceding function, `get_hire_date`, uses the `TO_CHAR` function to convert a `DATE` item to a `VARCHAR` item. The function `get_hire_date` does not specify a format mask, so the format mask defaults to the one that `NLS_DATE_FORMAT` specifies. If sessions that call `get_hire_date` have different `NLS_DATE_FORMAT` settings, cached results can have different formats. If a cached result computed by one session ages out, and another session recomputes it, the format might vary even for the same parameter value. If a session gets a cached result whose format differs from its own format, that result will probably be 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:

```
FUNCTION get_hire_date (emp_id NUMBER, fmt VARCHAR) RETURN VARCHAR
  RESULT_CACHE RELIES_ON (HR.EMPLOYEES) 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 at least one of the following:

- Directly invokes the built-in function `SYS_CONTEXT`, which returns the value of a specified attribute in a specified context
- Indirectly invokes `SYS_CONTEXT` by using Virtual Private Database (VPD) mechanisms for fine-grained security

(For information about VPD, see *Oracle Database Security Guide*.)

The PL/SQL function result-caching feature does not automatically handle dependence on session-specific application contexts. If you must cache the results of a function that depends on session-specific application contexts, you must pass the application context to the function as a parameter. You can give the parameter a default value, so that not every user must specify it.

In [Example 8–29](#), assume that `config_tab` has a VPD policy that translates this query:

```

SELECT value FROM config_tab
WHERE name = param_name;

```

To this query:

```

SELECT value FROM config_tab
WHERE name = param_name
AND app_id = SYS_CONTEXT('Config', 'App_ID');

```

Example 8–29 Result-Cached Function that Depends on Session-Specific Application Context

```

FUNCTION get_param_value
( param_name VARCHAR,
  appctx      VARCHAR
  DEFAULT SYS_CONTEXT('Config', 'App_ID')
)
RETURN VARCHAR
RESULT_CACHE RELIES_ON (config_tab)
IS
  result VARCHAR(2000);
BEGIN
  SELECT value INTO result
  FROM config_tab
  WHERE Name = param_name;
END;

```

Choosing Result-Caching Granularity

PL/SQL provides the function result cache, but you choose the caching granularity. To understand the concept of granularity, consider the `Product_Descriptions` table in the Order Entry (OE) sample schema:

NAME	NULL?	TYPE
-----	-----	-----

PRODUCT_ID	NOT NULL	NUMBER(6)
LANGUAGE_ID	NOT NULL	VARCHAR2(3)
TRANSLATED_NAME	NOT NULL	NVARCHAR2(50)
TRANSLATED_DESCRIPTION	NOT NULL	NVARCHAR2(2000)

The table has the name and description of each product in several languages. The unique key for each row is PRODUCT_ID, LANGUAGE_ID.

Suppose that you want to define a function that takes a PRODUCT_ID and a LANGUAGE_ID and returns the associated TRANSLATED_NAME. You also want to cache the translated names. Some of the granularity choices for caching the names are:

- One name at a time (finer granularity)
- One language at a time (coarser granularity)

Table 8–4 Comparison of Finer and Coarser Caching Granularity

Finer Granularity	Coarser Granularity
Each function result corresponds to one logical result.	Each function result contains many logical subresults.
Stores only data that is needed at least once.	Might store data that is never used.
Each data item ages out individually.	One aged-out data item ages out the whole set.
Does not allow bulk loading optimizations.	Allows bulk loading optimizations.

In each of the following four examples, the function productName takes a PRODUCT_ID and a LANGUAGE_ID and returns the associated TRANSLATED_NAME. Each version of productName caches translated names, but at a different granularity.

In [Example 8–30](#), get_product_name is a result-cached function. Whenever get_product_name is invoked with a different PRODUCT_ID and LANGUAGE_ID, it caches the associated TRANSLATED_NAME. Each call to get_product_name adds at most one TRANSLATED_NAME to the cache.

Example 8–30 Caching One Name at a Time (Finer Granularity)

```
FUNCTION get_product_name (prod_id NUMBER, lang_id VARCHAR2)
RETURN NVARCHAR2
  RESULT_CACHE RELIES_ON (Product_Descriptions)
IS
  result VARCHAR2(50);
BEGIN
  SELECT translated_name INTO result
  FROM product_descriptions
  WHERE PRODUCT_ID = prod_id
  AND LANGUAGE_ID = lang_id;
  RETURN result;
END;
```

In [Example 8–31](#), get_product_name defines a result-cached function, all_product_names. Whenever get_product_name invokes all_product_names with a different LANGUAGE_ID, all_product_names caches every TRANSLATED_NAME associated with that LANGUAGE_ID. Each call to all_product_names adds every TRANSLATED_NAME of at most one LANGUAGE_ID to the cache.

Example 8–31 Caching Translated Names One Language at a Time (Coarser Granularity)

```

FUNCTION get_product_name (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 RELIES_ON (product_descriptions)
  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;
  END;
  RETURN all_names;
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 has a private function result cache, available only to sessions on that instance.

The access pattern and work load of an instance determine the set of results in its private cache; therefore, the private caches of different instances can have different sets of results.

If a required result is missing from the private cache of the local instance, the body of the function executes to compute the result, which is then added to the local cache. The result is not retrieved from the private cache of another instance.

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.

Managing the Result Cache

The PL/SQL function result cache shares its administrative and manageability infrastructure with the Result Cache, which is described in *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 on these parameters, see *Oracle Database Reference* and *Oracle Database Performance Tuning Guide*.
- `DBMS_RESULT_CACHE` package

The `DBMS_RESULT_CACHE` package is a PL/SQL interface to manage the Result Cache. The package is to be used with dynamic performance views.

- Dynamic performance views:
 - `[G]V$RESULT_CACHE_STATISTICS`
 - `[G]V$RESULT_CACHE_MEMORY`
 - `[G]V$RESULT_CACHE_OBJECTS`
 - `[G]V$RESULT_CACHE_DEPENDENCY`

See Also:

Oracle Database PL/SQL Packages and Types Reference for more information about the `DBMS_RESULT_CACHE` package

Oracle Database Reference for information about the dynamic performance views `V$RESULT_CACHE_STATISTICS`, `V$RESULT_CACHE_MEMORY`, `V$RESULT_CACHE_OBJECTS`, and `V$RESULT_CACHE_DEPENDENCY`

Hot-Patching PL/SQL Program Units on Which Result-Cached Functions Depend

When you hot-patch a PL/SQL program unit on which a result-cached function depends (directly or indirectly), the cached results associated with the result-cached function might not be automatically flushed in all cases.

For example, suppose that the result-cached function `P1.foo()` depends on the packaged subprogram `P2.bar()`. If a new version of the body of package `P2` is loaded, the cached results associated with `P1.foo()` are not automatically flushed.

Therefore, this is the recommended procedure for hot-patching a PL/SQL program unit:

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.

Using Triggers

A **trigger** is a named program unit that is stored in the database and executed (**fired**) in response to a specified event that occurs in the database.

Topics:

- [Overview of Triggers](#)
- [Guidelines for Designing Triggers](#)
- [Privileges Required to Use Triggers](#)
- [Creating Triggers](#)
- [Coding the Trigger Body](#)
- [Compiling Triggers](#)
- [Modifying Triggers](#)
- [Debugging Triggers](#)
- [Enabling Triggers](#)
- [Disabling Triggers](#)
- [Viewing Information About Triggers](#)
- [Examples of Trigger Applications](#)
- [Responding to Database Events Through Triggers](#)

Overview of Triggers

A trigger is a named program unit that is stored in the database and **fired** (executed) in response to a specified event. The specified **event** is associated with either a table, a view, a schema, or the database, and it is one of the following:

- A database manipulation (DML) statement (DELETE, INSERT, or UPDATE)
- A database definition (DDL) statement (CREATE, ALTER, or DROP)
- A database operation (SERVERERROR, LOGON, LOGOFF, STARTUP, or SHUTDOWN)

The trigger is said to be **defined on** the table, view, schema, or database.

Trigger Types

A **DML trigger** is fired by a DML statement, a **DDL trigger** is fired by a DDL statement, a **DELETE trigger** is fired by a DELETE statement, and so on.

An **INSTEAD OF trigger** is a DML trigger that is defined on a view (not a table). Oracle Database fires the `INSTEAD OF` trigger instead of executing the triggering DML statement. For more information, see ["Modifying Complex Views \(INSTEAD OF Triggers\)"](#) on page 9-7.

A **system trigger** is defined on a schema or the database. A trigger defined on a schema fires for each event associated with the owner of the schema (the current user). A trigger defined on a database fires for each event associated with all users.

A **simple trigger** can fire at exactly one of the following **timing points**:

- Before the triggering statement executes
- After the triggering statement executes
- Before each row that the triggering statement affects
- After each row that the triggering statement affects

A **compound trigger** can fire at more than one timing point. Compound triggers make it easier to program an approach where you want the actions you implement for the various timing points to share common data. For more information, see ["Compound Triggers"](#) on page 9-12.

Trigger States

A trigger can be in either of two states:

Enabled. An enabled trigger executes its trigger body if a triggering statement is entered and the trigger restriction (if any) evaluates to `TRUE`.

Disabled. A disabled trigger does not execute its trigger body, even if a triggering statement is entered and the trigger restriction (if any) evaluates to `TRUE`.

By default, a trigger is created in enabled state. To create a trigger in disabled state, use the `DISABLE` clause of the `CREATE TRIGGER` statement.

See Also: *Oracle Database SQL Language Reference* for more information about the `CREATE TRIGGER` statement

Guidelines for Designing Triggers

Use the following guidelines when designing triggers:

- Use triggers to guarantee that when a specific operation is performed, related actions are performed.
- Do not define triggers that duplicate features already built into Oracle Database.
For example, do not define triggers to reject bad data if you can do the same checking through declarative constraints.

- Limit the size of triggers.

If the logic for your trigger requires much more than 60 lines of PL/SQL code, put most of the code in a stored subprogram and invoke the subprogram from the trigger.

- Use triggers only for centralized, global operations that must fire for the triggering statement, regardless of which user or database application issues the statement.
- Do not create recursive triggers.

For example, creating an AFTER UPDATE statement trigger on the emp table that itself issues an UPDATE statement on emp, causes the trigger to fire recursively until it has run out of memory.

- Use triggers on DATABASE judiciously. They are executed for every user every time the event occurs on which the trigger is created.

Note: The size of the trigger cannot be more than 32K.

Privileges Required to Use Triggers

To create a trigger in your schema:

- You must have the CREATE TRIGGER system privilege
- One of the following must be true:
 - You own the table specified in the triggering statement
 - You have the ALTER privilege for the table specified in the triggering statement
 - You have the ALTER ANY TABLE system privilege

To create a trigger in another schema, or to reference a table in another schema from a trigger in your schema:

- You must have the CREATE ANY TRIGGER system privilege.
- You must have the EXECUTE privilege on the referenced subprograms or packages.

To create a trigger on the database, you must have the ADMINISTER DATABASE TRIGGER privilege. If this privilege is later revoked, you can drop the trigger but not alter it.

The object privileges to the schema objects referenced in the trigger body must be granted to the trigger owner explicitly (not through a role). The statements in the trigger body operate under the privilege domain of the trigger owner, not the privilege domain of the user issuing the triggering statement (this is similar to the privilege model for stored subprograms).

Creating Triggers

To create a trigger, use the CREATE TRIGGER statement. By default, a trigger is created in enabled state. To create a trigger in disabled state, use the DISABLE clause of the CREATE TRIGGER statement. For information about trigger states, see "[Overview of Triggers](#)" on page 9-1.

When using the CREATE TRIGGER statement with an interactive tool, such as SQL*Plus or Enterprise Manager, put a single slash (/) on the last line, as in [Example 9-1](#), which creates a simple trigger for the emp table.

Example 9-1 CREATE TRIGGER Statement

```
CREATE OR REPLACE TRIGGER Print_salary_changes
  BEFORE DELETE OR INSERT OR UPDATE ON emp
  FOR EACH ROW
  WHEN (new.EMPNO > 0)
  DECLARE
    sal_diff number;
```

```
BEGIN
    sal_diff := :NEW.SAL - :OLD.SAL;
    dbms_output.put('Old salary: ' || :old.sal);
    dbms_output.put(' New salary: ' || :new.sal);
    dbms_output.put_line(' Difference ' || sal_diff);
END;
/
```

See Also: *Oracle Database SQL Language Reference* for more information about the CREATE TRIGGER statement

The trigger in [Example 9-1](#) fires when DML operations (INSERT, UPDATE, and DELETE statements) are performed on the table. You can choose what combination of operations must fire the trigger.

Because the trigger uses the BEFORE keyword, it can access the new values before they go into the table, and can change the values if there is an easily-corrected error by assigning to :NEW.column_name. You might use the AFTER keyword if you want the trigger to query or change the same table, because triggers can only do that after the initial changes are applied and the table is back in a consistent state.

Because the trigger uses the FOR EACH ROW clause, it might be executed multiple times, such as when updating or deleting multiple rows. You might omit this clause if you just want to record the fact that the operation occurred, but not examine the data for each row.

After the trigger is created, following SQL statement fires the trigger once for each row that is updated, in each case printing the new salary, the old salary, and the difference between them:

```
UPDATE emp SET sal = sal + 500.00 WHERE deptno = 10;
```

The CREATE (or CREATE OR REPLACE) statement fails if any errors exist in the PL/SQL block.

The following sections use [Example 9-1](#) on page 9-3 to show how parts of a trigger are specified. For additional examples of CREATE TRIGGER statements, see "[Examples of Trigger Applications](#)" on page 9-30.

Topics:

- [Naming Triggers](#)
- [When Does the Trigger Fire?](#)
- [Controlling When a Trigger Fires \(BEFORE and AFTER Options\)](#)
- [Modifying Complex Views \(INSTEAD OF Triggers\)](#)
- [Firing Triggers One or Many Times \(FOR EACH ROW Option\)](#)
- [Firing Triggers Based on Conditions \(WHEN Clause\)](#)
- [Compound Triggers](#)
- [Ordering of Triggers](#)

Naming Triggers

Trigger names must be unique with respect to other triggers in the same schema. Trigger names do not need to be unique with respect to other schema objects, such as tables, views, and subprograms. For example, a table and a trigger can have the same name (however, to avoid confusion, this is not recommended).

When Does the Trigger Fire?

A trigger fires based on a **triggering statement**, which specifies:

- The SQL statement or the database event or DDL event that fires the trigger body. The options include `DELETE`, `INSERT`, and `UPDATE`. One, two, or all three of these options can be included in the triggering statement specification.
- The table, view, `DATABASE`, or `SCHEMA` on which the trigger is defined.

Note: Exactly one table or view can be specified in the triggering statement. If the `INSTEAD OF` option is used, then the triggering statement must specify a view; conversely, if a view is specified in the triggering statement, then only the `INSTEAD OF` option can be used.

In [Example 9-1](#) on page 9-3, the `PRINT_SALARY_CHANGES` trigger fires after any `DELETE`, `INSERT`, or `UPDATE` on the `emp` table. Any of the following statements trigger the `PRINT_SALARY_CHANGES` trigger:

```
DELETE FROM emp;
INSERT INTO emp VALUES ( ... );
INSERT INTO emp SELECT ... FROM ... ;
UPDATE emp SET ... ;
```

Do Import and SQL*Loader Fire Triggers?

`INSERT` triggers fire during SQL*Loader conventional loads. (For direct loads, triggers are disabled before the load.)

The `IGNORE` parameter of the `IMP` statement determines whether triggers fire during import operations:

- If `IGNORE=N` (default) and the table already exists, then import does not change the table and no existing triggers fire.
- If the table does not exist, then import creates and loads it before any triggers are defined, so again no triggers fire.
- If `IGNORE=Y`, then import loads rows into existing tables. Any existing triggers fire, and indexes are updated to account for the imported data.

How Column Lists Affect UPDATE Triggers

An `UPDATE` statement might include a list of columns. If a triggering statement includes a column list, the trigger fires only when one of the specified columns is updated. If a triggering statement omits a column list, the trigger fires when any column of the associated table is updated. A column list cannot be specified for `INSERT` or `DELETE` triggering statements.

The previous example of the `PRINT_SALARY_CHANGES` trigger can include a column list in the triggering statement. For example:

```
... BEFORE DELETE OR INSERT OR UPDATE OF ename ON emp ...
```

Note:

- You cannot specify a column list for UPDATE with INSTEAD OF triggers.
 - If the column specified in the UPDATE OF clause is an object column, then the trigger also fires if any of the attributes of the object are modified.
 - You cannot specify UPDATE OF clauses on collection columns.
-

Controlling When a Trigger Fires (BEFORE and AFTER Options)

Note: This topic applies only to simple triggers. For the options of compound triggers, see "[Compound Triggers](#)" on page 9-12.

The BEFORE or AFTER option in the CREATE TRIGGER statement specifies exactly when to fire the trigger body in relation to the triggering statement that is being run. In a CREATE TRIGGER statement, the BEFORE or AFTER option is specified just before the triggering statement. For example, the PRINT_SALARY_CHANGES trigger in the previous example is a BEFORE trigger.

In general, you use BEFORE or AFTER triggers to achieve the following results:

- Use BEFORE row triggers to modify the row before the row data is written to disk.
- Use AFTER row triggers to obtain, and perform operations, using the row ID.

An AFTER row trigger fires when the triggering statement results in ORA-2292 Child Record Found.

Note: BEFORE row triggers are slightly more efficient than AFTER row triggers. With AFTER row triggers, affected data blocks must be read (logical read, not physical read) once for the trigger and then again for the triggering statement. Alternatively, with BEFORE row triggers, the data blocks must be read only once for both the triggering statement and the trigger.

If an UPDATE or DELETE statement detects a conflict with a concurrent UPDATE, then Oracle Database performs a transparent ROLLBACK to SAVEPOINT and restarts the update. This can occur many times before the statement completes successfully. Each time the statement is restarted, the BEFORE statement trigger fires again. The rollback to savepoint does not undo changes to any package variables referenced in the trigger. Include a counter variable in your package to detect this situation.

Ordering of Triggers

A relational database does not guarantee the order of rows processed by a SQL statement. Therefore, do not create triggers that depend on the order in which rows are processed. For example, do not assign a value to a global package variable in a row trigger if the current value of the global variable is dependent on the row being processed by the row trigger. Also, if global package variables are updated within a trigger, then it is best to initialize those variables in a BEFORE statement trigger.

When a statement in a trigger body causes another trigger to fire, the triggers are said to be **cascading**. Oracle Database allows up to 32 triggers to cascade at simultaneously. You can limit the number of trigger cascades by using the initialization parameter `OPEN_CURSORS`, because a cursor must be opened for every execution of a trigger.

Although any trigger can run a sequence of operations either in-line or by invoking subprograms, using multiple triggers of the same type allows the modular installation of applications that have triggers on the same tables.

Each subsequent trigger sees the changes made by the previously fired triggers. Each trigger can see the old and new values. The old values are the original values, and the new values are the current values, as set by the most recently fired `UPDATE` or `INSERT` trigger.

Oracle Database executes all triggers of the same type before executing triggers of a different type. If you have multiple triggers of the same type on the same table, and the order in which they execute is important, use the `FOLLOWS` clause. Without the `FOLLOWS` clause, Oracle Database chooses an arbitrary, unpredictable order.

See Also: *Oracle Database SQL Language Reference* for more information about the `FOLLOWS` clause of the `CREATE TRIGGER` statement

Modifying Complex Views (INSTEAD OF Triggers)

Note: `INSTEAD OF` triggers can be defined only on views, not on tables.

An **updatable** view is one that lets you perform DML on the underlying table. Some views are inherently updatable, but others are not because they were created with one or more of the constructs listed in "[Views that Require INSTEAD OF Triggers](#)" on page 9-8.

Any view that contains one of those constructs can be made updatable by using an `INSTEAD OF` trigger. `INSTEAD OF` triggers provide a transparent way of modifying views that cannot be modified directly through `UPDATE`, `INSERT`, and `DELETE` statements. These triggers are invoked `INSTEAD OF` triggers because, unlike other types of triggers, Oracle Database fires the trigger instead of executing the triggering statement. The trigger must determine what operation was intended and perform `UPDATE`, `INSERT`, or `DELETE` operations directly on the underlying tables.

With an `INSTEAD OF` trigger, you can write normal `UPDATE`, `INSERT`, and `DELETE` statements against the view, and the `INSTEAD OF` trigger works invisibly in the background to make the right actions take place.

`INSTEAD OF` triggers can only be activated for each row.

See Also: "[Firing Triggers One or Many Times \(FOR EACH ROW Option\)](#)" on page 9-11

Note:

- The `INSTEAD OF` option can be used only for triggers defined on views.
 - The `BEFORE` and `AFTER` options cannot be used for triggers defined on views.
 - The `CHECK` option for views is not enforced when inserts or updates to the view are done using `INSTEAD OF` triggers. The `INSTEAD OF` trigger body must enforce the check.
-

Views that Require INSTEAD OF Triggers

A view cannot be modified by `UPDATE`, `INSERT`, or `DELETE` statements if the view query contains any of the following constructs:

- A set operator
- A `DISTINCT` operator
- An aggregate or analytic function
- A `GROUP BY`, `ORDER BY`, `MODEL`, `CONNECT BY`, or `START WITH` clause
- A collection expression in a `SELECT` list
- A subquery in a `SELECT` list
- A subquery designated `WITH READ ONLY`
- Joins, with some exceptions, as documented in *Oracle Database Administrator's Guide*

If a view contains pseudocolumns or expressions, then you can only update the view with an `UPDATE` statement that does not refer to any of the pseudocolumns or expressions.

`INSTEAD OF` triggers provide the means to modify object view instances on the client-side through OCI calls.

See Also: *Oracle Call Interface Programmer's Guide*

To modify an object materialized by an object view in the client-side object cache and flush it back to the persistent store, you must specify `INSTEAD OF` triggers, unless the object view is modifiable. If the object is read only, then it is not necessary to define triggers to pin it.

Triggers on Nested Table View Columns

`INSTEAD OF` triggers can also be created over nested table view columns. These triggers provide a way of updating elements of the nested table. They fire for each nested table element being modified. The row correlation variables inside the trigger correspond to the nested table element. This type of trigger also provides an additional correlation name for accessing the parent row that contains the nested table being modified.

Note: These triggers:

- Can only be defined over nested table columns in views.
 - Fire only when the nested table elements are modified using the TABLE clause. They do not fire when a DML statement is performed on the view.
-
-

For example, consider a department view that contains a nested table of employees.

```
CREATE OR REPLACE VIEW Dept_view AS
  SELECT d.Deptno, d.Dept_type, d.Dname,
         CAST (MULTISET ( SELECT e.Empno, e.Empname, e.Salary)
              FROM emp e
              WHERE e.Deptno = d.Deptno) AS Emp_list_ Emplist
  FROM dept d;
```

The CAST (MULTISET) operator creates a multiset of employees for each department. To modify the emplist column, which is the nested table of employees, define an INSTEAD OF trigger over the column to handle the operation.

The following example shows how an insert trigger might be written:

```
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 emp VALUES (:Employee.Empno,
                          :Employee.Ename, :Employee.Sal, :Department.Deptno);
END;
```

Any INSERT into the nested table fires the trigger, and the emp table is filled with the correct values. For example:

```
INSERT INTO TABLE (SELECT d.Emplist FROM Dept_view d WHERE Deptno = 10)
VALUES (1001, 'John Glenn', 10000);
```

The :department.deptno correlation variable in this example has the value 10.

Example: INSTEAD OF Trigger

Note: You might need to set up the following data structures for this example to work:

```
CREATE TABLE Project_tab (
  Prj_level NUMBER,
  Projno     NUMBER,
  Resp_dept NUMBER);
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);
```

The following example shows an INSTEAD OF trigger for inserting rows into the MANAGER_INFO view.

```
CREATE OR REPLACE VIEW manager_info AS
  SELECT e.ename, e.empno, d.dept_type, d.deptno, p.prj_level, p.projno
  FROM emp e, dept d, Project_tab p
  WHERE e.empno = d.mgr_no
  AND d.deptno = p.resp_dept;

CREATE OR REPLACE TRIGGER manager_info_insert
  INSTEAD OF INSERT ON manager_info
  REFERENCING NEW AS n -- new manager information
  FOR EACH ROW
DECLARE
  rowcnt number;
BEGIN
  SELECT COUNT(*) INTO rowcnt FROM emp WHERE empno = :n.empno;
  IF rowcnt = 0 THEN
    INSERT INTO emp (empno,ename) VALUES (:n.empno, :n.ename);
  ELSE
    UPDATE emp SET emp.ename = :n.ename WHERE emp.empno = :n.empno;
  END IF;
  SELECT COUNT(*) INTO rowcnt FROM dept WHERE deptno = :n.deptno;
  IF rowcnt = 0 THEN
    INSERT INTO dept (deptno, dept_type)
    VALUES (:n.deptno, :n.dept_type);
  ELSE
    UPDATE dept SET dept.dept_type = :n.dept_type
    WHERE dept.deptno = :n.deptno;
  END IF;
  SELECT COUNT(*) INTO rowcnt FROM Project_tab
  WHERE Project_tab.projno = :n.projno;
  IF rowcnt = 0 THEN
```

```

INSERT INTO Project_tab (projno, prj_level)
VALUES (:n.projno, :n.prj_level);
ELSE
UPDATE Project_tab SET Project_tab.prj_level = :n.prj_level
WHERE Project_tab.projno = :n.projno;
END IF;
END;

```

The actions shown for rows being inserted into the MANAGER_INFO view first test to see if appropriate rows already exist in the base tables from which MANAGER_INFO is derived. The actions then insert new rows or update existing rows, as appropriate. Similar triggers can specify appropriate actions for UPDATE and DELETE.

Firing Triggers One or Many Times (FOR EACH ROW Option)

Note: This topic applies only to simple triggers. For the options of compound triggers, see "[Compound Triggers](#)" on page 9-12.

The FOR EACH ROW option determines whether the trigger is a row trigger or a statement trigger. If you specify FOR EACH ROW, then the trigger fires once for each row of the table that is affected by the triggering statement. The absence of the FOR EACH ROW option indicates that the trigger fires only once for each applicable statement, but not separately for each row affected by the statement.

For example, you define the following trigger:

Note: You might need to set up the following data structures for certain examples to work:

```

CREATE TABLE Emp_log (
Emp_id    NUMBER,
Log_date  DATE,
New_salary NUMBER,
Action    VARCHAR2(20));

```

```

CREATE OR REPLACE TRIGGER Log_salary_increase
AFTER UPDATE ON emp
FOR EACH ROW
WHEN (new.Sal > 1000)
BEGIN
INSERT INTO Emp_log (Emp_id, Log_date, New_salary, Action)
VALUES (:new.Empno, SYSDATE, :new.SAL, 'NEW SAL');
END;

```

Then, you enter the following SQL statement:

```

UPDATE emp SET Sal = Sal + 1000.0
WHERE Deptno = 20;

```

If there are five employees in department 20, then the trigger fires five times when this statement is entered, because five rows are affected.

The following trigger fires only once for each UPDATE of the emp table:

```

CREATE OR REPLACE TRIGGER Log_emp_update
AFTER UPDATE ON emp
BEGIN

```

```
INSERT INTO Emp_log (Log_date, Action)
VALUES (SYSDATE, 'emp COMMISSIONS CHANGED');
END;
```

The statement level triggers are useful for performing validation checks for the entire statement.

Firing Triggers Based on Conditions (WHEN Clause)

Optionally, a trigger restriction can be included in the definition of a row trigger by specifying a Boolean SQL expression in a `WHEN` clause.

Note: A `WHEN` clause cannot be included in the definition of a statement trigger.

If included, then the expression in the `WHEN` clause is evaluated for each row that the trigger affects.

If the expression evaluates to `TRUE` for a row, then the trigger body executes on behalf of that row. However, if the expression evaluates to `FALSE` or `NOT TRUE` for a row (unknown, as with nulls), then the trigger body does not execute for that row. The evaluation of the `WHEN` clause does not have an effect on the execution of the triggering SQL statement (in other words, the triggering statement is not rolled back if the expression in a `WHEN` clause evaluates to `FALSE`).

For example, in the `PRINT_SALARY_CHANGES` trigger, the trigger body is not run if the new value of `Empno` is zero, `NULL`, or negative. In more realistic examples, you might test if one column value is less than another.

The expression in a `WHEN` clause of a row trigger can include correlation names, which are explained later. The expression in a `WHEN` clause must be a SQL expression, and it cannot include a subquery. You cannot use a PL/SQL expression (including user-defined functions) in the `WHEN` clause.

Note: You cannot specify the `WHEN` clause for `INSTEAD OF` triggers.

Compound Triggers

A compound trigger can fire at more than one timing point.

Topics:

- [Why Use Compound Triggers?](#)
- [Compound Trigger Sections](#)
- [Triggering Statements of Compound Triggers](#)
- [Compound Trigger Restrictions](#)
- [Compound Trigger Example](#)
- [Using Compound Triggers to Avoid Mutating-Table Error](#)

Why Use Compound Triggers?

The compound trigger makes it easier to program an approach where you want the actions you implement for the various timing points to share common data. To achieve the same effect with simple triggers, you had to model the common state with an

ancillary package. This approach was both cumbersome to program and subject to memory leak when the triggering statement caused an error and the after-statement trigger did not fire.

A compound trigger has a declaration section and a section for each of its timing points (see [Example 9-2](#)). All of these sections can access a common PL/SQL state. The common state is established when the triggering statement starts and is destroyed when the triggering statement completes, even when the triggering statement causes an error.

Example 9-2 Compound Trigger

```
CREATE TRIGGER compound_trigger
  FOR UPDATE OF sal ON emp
  COMPOUND TRIGGER

  -- Declaration Section
  -- Variables declared here have firing-statement duration.
  threshold CONSTANT SIMPLE_INTEGER := 200;

  BEFORE STATEMENT IS
  BEGIN
    ...
  END BEFORE STATEMENT;

  BEFORE EACH ROW IS
  BEGIN
    ...
  END BEFORE EACH ROW;

  AFTER EACH ROW IS
  BEGIN
    ...
  END AFTER EACH ROW;
END compound_trigger;
/
```

Two common reasons to use compound triggers are:

- To accumulate rows destined for a second table so that you can periodically bulk-insert them (as in ["Compound Trigger Example"](#) on page 9-15)
- To avoid the mutating-table error (ORA-04091) (as in ["Using Compound Triggers to Avoid Mutating-Table Error"](#) on page 9-16)

Compound Trigger Sections

A compound trigger has a declaration section and at least one timing-point section.

The **declaration section** (the first section) declares variables and subprograms that timing-point sections can use. When the trigger fires, the declaration section executes before any timing-point sections execute. Variables and subprograms declared in this section have firing-statement duration.

A compound trigger defined on a view has an `INSTEAD OF EACH ROW` timing-point section, and no other timing-point section.

A compound trigger defined on a table has one or more of the timing-point sections described in [Table 9-1](#). Timing-point sections must appear in the order shown in [Table 9-1](#). If a timing-point section is absent, nothing happens at its timing point.

A timing-point section cannot be enclosed in a PL/SQL block.

Table 9–1 summarizes the timing point sections of a compound trigger that can be defined on a table.

Table 9–1 Timing-Point Sections of a Compound Trigger Defined

Timing Point	Section
Before the triggering statement executes	BEFORE STATEMENT
After the triggering statement executes	AFTER STATEMENT
Before each row that the triggering statement affects	BEFORE EACH ROW
After each row that the triggering statement affects	AFTER EACH ROW

Any section can include the functions `Inserting`, `Updating`, `Deleting`, and `Applying`.

Triggering Statements of Compound Triggers

The triggering statement of a compound trigger must be a DML statement.

If the triggering statement affects no rows, and the compound trigger has neither a `BEFORE STATEMENT` section nor an `AFTER STATEMENT` section, the trigger never fires.

It is when the triggering statement affects many rows that a compound trigger has a performance benefit. This is why it is important to use the `BULK COLLECT` clause with the `FORALL` statement. For example, without the `BULK COLLECT` clause, a `FORALL` statement that contains an `INSERT` statement simply performs a single-row insertion operation many times, and you get no benefit from using a compound trigger. For more information about using the `BULK COLLECT` clause with the `FORALL` statement, see ["Using FORALL and BULK COLLECT Together"](#) on page 12-23.

If the triggering statement of a compound trigger is an `INSERT` statement that includes a subquery, the compound trigger retains some of its performance benefit. Its `BEFORE EACH ROW` and `AFTER EACH ROW` sections execute once for each affected row, but its `BEFORE STATEMENT` and `AFTER STATEMENT` sections each execute only once (before and after the `INSERT` statement executes, respectively).

Compound Trigger Restrictions

- The body of a compound trigger must be a compound trigger block.
- A compound trigger must be a DML trigger.
- A compound trigger must be defined on either a table or a view.
- The declaration section cannot include `PRAGMA AUTONOMOUS_TRANSACTION`.
- A compound trigger body cannot have an initialization block; therefore, it cannot have an exception section.

This is not a problem, because the `BEFORE STATEMENT` section always executes exactly once before any other timing-point section executes.

- An exception that occurs in one section must be handled in that section. It cannot transfer control to another section.
- If a section includes a `GOTO` statement, the target of the `GOTO` statement must be in the same section.
- `:OLD`, `:NEW`, and `:PARENT` cannot appear in the declaration section, the `BEFORE STATEMENT` section, or the `AFTER STATEMENT` section.

- Only the BEFORE EACH ROW section can change the value of :NEW.
- If, after the compound trigger fires, the triggering statement rolls back due to a DML exception:
 - Local variables declared in the compound trigger sections are re-initialized, and any values computed thus far are lost.
 - Side effects from firing the compound trigger are not rolled back.
- The firing order of compound triggers is not guaranteed. Their firing can be interleaved with the firing of simple triggers.
- If compound triggers are ordered using the FOLLOWS option, and if the target of FOLLOWS does not contain the corresponding section as source code, the ordering is ignored.

Compound Trigger Example

Scenario: You want to record every change to `hr.employees.salary` in a new table, `employee_salaries`. A single `UPDATE` statement will update many rows of the table `hr.employees`; therefore, bulk-inserting rows into `employee_salaries` is more efficient than inserting them individually.

Solution: Define a compound trigger on updates of the table `hr.employees`, as in [Example 9-3](#). 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).

Example 9-3 Compound Trigger Records Changes to One Table in Another Table

```
CREATE TABLE employee_salaries (
  employee_id NUMBER NOT NULL,
  change_date DATE NOT NULL,
  salary NUMBER(8,2) NOT NULL,
  CONSTRAINT pk_employee_salaries PRIMARY KEY (employee_id, change_date),
  CONSTRAINT fk_employee_salaries FOREIGN KEY (employee_id)
    REFERENCES employees (employee_id)
    ON DELETE CASCADE)
/
CREATE OR REPLACE TRIGGER maintain_employee_salaries
  FOR UPDATE OF salary ON employees
  COMPOUND TRIGGER

-- Declaration Section:
-- Choose small threshold value to show how example works:
threshold 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 >= threshold THEN
        flush_array();
    END IF;
END AFTER EACH ROW;

-- AFTER STATEMENT Section:

AFTER STATEMENT IS
BEGIN
    flush_array();
END AFTER STATEMENT;
END maintain_employee_salaries;
/
/* Increase salary of every employee in department 50 by 10%: */

UPDATE employees
    SET salary = salary * 1.1
    WHERE department_id = 50
/

/* Wait two seconds: */

BEGIN
    DBMS_LOCK.SLEEP(2);
END;
/

/* Increase salary of every employee in department 50 by 5%: */

UPDATE employees
    SET salary = salary * 1.05
    WHERE department_id = 50
/

```

Using Compound Triggers to Avoid Mutating-Table Error

You can use compound triggers to avoid the mutating-table error (ORA-04091) described in ["Trigger Restrictions on Mutating Tables"](#) on page 9-24.

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 9-4](#). The state variables are initialized each time the trigger fires (even when the triggering statement is interrupted and restarted).

Example 9-4 Compound Trigger that 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;

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;

```

Coding the Trigger Body

Note: This topic applies primarily to simple triggers. The body of a compound trigger has a different format (see "[Compound Triggers](#)" on page 9-12).

The trigger body is a CALL subprogram or a PL/SQL block that can include SQL and PL/SQL statements. The CALL subprogram can be either a PL/SQL or a Java subprogram that is encapsulated in a PL/SQL wrapper. These statements are run if the triggering statement is entered and if the trigger restriction (if included) evaluates to TRUE.

The trigger body for row triggers has some special constructs that can be included in the code of the PL/SQL block: correlation names and the REFERENCEING option, and the conditional predicates INSERTING, DELETING, and UPDATING.

Note: The INSERTING, DELETING, and UPDATING conditional predicates cannot be used for the CALL subprograms; they can only be used in a PL/SQL block.

Example: Monitoring Logons with a Trigger

Note: You might need to set up data structures similar to the following for certain examples to work:

```
CONNECT SYSTEM/password
GRANT ADMINISTER DATABASE TRIGGER TO hr;
CONNECT hr/password
CREATE TABLE audit_table (
    seq number,
    user_at VARCHAR2(10),
    time_now DATE,
    term VARCHAR2(10),
    job VARCHAR2(10),
    proc VARCHAR2(10),
    enum NUMBER);
```

```
CREATE OR REPLACE PROCEDURE foo (c VARCHAR2) AS
BEGIN
    INSERT INTO Audit_table (user_at) VALUES(c);
END;
```

```
CREATE OR REPLACE TRIGGER logontrig AFTER LOGON ON DATABASE
-- Just invoke an existing procedure. The ORA_LOGIN_USER is a function
-- that returns information about the event that fired the trigger.
CALL foo (ora_login_user)
/
```

Example: Invoking a Java Subprogram from a Trigger Although triggers are declared using PL/SQL, they can call subprograms in other languages, such as Java:

```
CREATE OR REPLACE PROCEDURE Before_delete (Id IN NUMBER, Ename VARCHAR2)
IS language Java
name 'thjvTriggers.beforeDelete (oracle.sql.NUMBER, oracle.sql.CHAR)';

CREATE OR REPLACE TRIGGER Pre_del_trigger BEFORE DELETE ON Tab
FOR EACH ROW
CALL Before_delete (:old.Id, :old.Ename)
/
```

The corresponding Java file is thjvTriggers.java:

```
import java.sql.*
import java.io.*
import oracle.sql.*
import oracle.oracore.*
public class thjvTriggers
{
public state void
beforeDelete (NUMBER old_id, CHAR old_name)
Throws SQLException, CoreException
    {
        Connection conn = JDBCConnection.defaultConnection();
        Statement stmt = conn.createStatement();
        String sql = "insert into logtab values
        (" + old_id.intValue() + ", '" + old_ename.toString() + "', BEFORE DELETE)";
        stmt.executeUpdate (sql);
        stmt.close();
        return;
    }
}
```

Topics:

- [Accessing Column Values in Row Triggers](#)
- [Triggers on Object Tables](#)
- [Triggers and Handling Remote Exceptions](#)
- [Restrictions on Creating Triggers](#)
- [Who Uses the Trigger?](#)

Accessing Column Values in Row Triggers

Within a trigger body of a row trigger, the PL/SQL code and SQL statements have access to the old and new column values of the current row affected by the triggering statement. Two correlation names exist for every column of the table being modified: one for the old column value, and one for the new column value. Depending on the type of triggering statement, certain correlation names might not have any meaning.

- A trigger fired by an `INSERT` statement has meaningful access to new column values only. Because the row is being created by the `INSERT`, the old values are null.
- A trigger fired by an `UPDATE` statement has access to both old and new column values for both `BEFORE` and `AFTER` row triggers.
- A trigger fired by a `DELETE` statement has meaningful access to `:old` column values only. Because the row no longer exists after the row is deleted, the `:new` values are `NULL`. However, you cannot modify `:new` values because `ORA-4084` is raised if you try to modify `:new` values.

The new column values are referenced using the new qualifier before the column name, while the old column values are referenced using the `old` qualifier before the column name. For example, if the triggering statement is associated with the `emp` table (with the columns `SAL`, `COMM`, and so on), then you can include statements in the trigger body. For example:

```
IF :new.Sal > 10000 ...
IF :new.Sal < :old.Sal ...
```

Old and new values are available in both `BEFORE` and `AFTER` row triggers. A new column value can be assigned in a `BEFORE` row trigger, but not in an `AFTER` row trigger (because the triggering statement takes effect before an `AFTER` row trigger fires). If a `BEFORE` row trigger changes the value of `new.column`, then an `AFTER` row trigger fired by the same statement sees the change assigned by the `BEFORE` row trigger.

Correlation names can also be used in the Boolean expression of a `WHEN` clause. A colon (`:`) must precede the `old` and `new` qualifiers when they are used in a trigger body, but a colon is not allowed when using the qualifiers in the `WHEN` clause or the `REFERENCING` option.

Example: Modifying LOB Columns with a Trigger

You can treat LOB columns the same as other columns, using regular SQL and PL/SQL functions with `CLOB` columns, and calls to the `DBMS_LOB` package with `BLOB` columns:

```
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);

  -- Previously, we couldn't change the new value for a LOB.
  -- Now, we can replace it, or construct a new value using SUBSTR, INSTR...
  -- operations for a CLOB, or DBMS_LOB calls for a BLOB.
  :NEW.c1 := :NEW.c1 || to_clob('<hr><p>Standard footer paragraph.');
```

```
  dbms_output.put_line('Final value of CLOB column: '||:NEW.c1);
end;
/

set serveroutput on;
update tab1 set c1 = '<h1>Different Document Fragment</h1><p>Different text.';

select * from tab1;
```

INSTEAD OF Triggers on Nested Table View Columns

In the case of INSTEAD OF triggers on nested table view columns, the new and old qualifiers correspond to the new and old nested table elements. The parent row corresponding to this nested table element can be accessed using the parent qualifier. The parent correlation name is meaningful and valid only inside a nested table trigger.

Avoiding Trigger Name Conflicts (REFERENCING Option)

The REFERENCING option can be specified in a trigger body of a row trigger to avoid name conflicts among the correlation names and tables that might be named old or new. Because this is rare, this option is infrequently used.

For example, assume you have a table named new with columns field1 (number) and field2 (character). The following CREATE TRIGGER example shows a trigger defined on the new table that can use correlation names and avoid naming conflicts between the correlation names and the table name:

Note: You might need to set up the following data structures for certain examples to work:

```
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;
```

Notice that the new qualifier is renamed to newest using the REFERENCING option, and it is then used in the trigger body.

Detecting the DML Operation that Fired a Trigger

If more than one type of DML operation can fire a trigger (for example, ON INSERT OR DELETE OR UPDATE OF emp), the trigger body can use the conditional predicates INSERTING, DELETING, and UPDATING to check which type of statement fire the trigger.

Within the code of the trigger body, you can execute blocks of code depending on the kind of DML operation that fired the trigger:

```
IF INSERTING THEN ... END IF;
IF UPDATING THEN ... END IF;
```

The first condition evaluates to TRUE only if the statement that fired the trigger is an INSERT statement; the second condition evaluates to TRUE only if the statement that fired the trigger is an UPDATE statement.

In an UPDATE trigger, a column name can be specified with an UPDATING conditional predicate to determine if the named column is being updated. For example, assume a trigger is defined as the following:

```
CREATE OR REPLACE TRIGGER ...
... UPDATE OF Sal, Comm ON emp ...
BEGIN

... IF UPDATING ('SAL') THEN ... END IF;

END;
```

The code in the THEN clause runs only if the triggering UPDATE statement updates the SAL column. This way, the trigger can minimize its overhead when the column of interest is not being changed.

Error Conditions and Exceptions in the Trigger Body

If a predefined or user-defined error condition or exception is raised during the execution of a trigger body, then all effects of the trigger body, as well as the triggering statement, are rolled back (unless the error is trapped by an exception handler). Therefore, a trigger body can prevent the execution of the triggering statement by raising an exception. User-defined exceptions are commonly used in triggers that enforce complex security authorizations or constraints.

If the logon trigger raises an exception, logon fails except in the following cases:

- Database startup and shutdown operations do not fail even if the system triggers for these events raise exceptions. Only the trigger action is rolled back. The error is logged in trace files and the alert log.
- If the system trigger is a database logon trigger and the user has ADMINISTER DATABASE TRIGGER privilege, then the user is able to log on successfully even if the trigger raises an exception. For schema logon triggers, if the user logging on is the trigger owner or has ALTER ANY TRIGGER privileges then logon is permitted. Only the trigger action is rolled back and an error is logged in the trace files and alert log.

Triggers on Object Tables

You can use the OBJECT_VALUE pseudocolumn in a trigger on an object table because 10g Release 1 (10.1). OBJECT_VALUE means the object as a whole. This is one example of its use. You can also invoke a PL/SQL function with OBJECT_VALUE as the datatype of an IN formal parameter.

Here is an example of the use of `OBJECT_VALUE` in a trigger. To keep track of updates to values in an object table `tbl`, a history table, `tbl_history`, is also created in the following example. For `tbl`, the values 1 through 5 are inserted into `n`, while `m` is kept at 0. The trigger is a row-level trigger that executes once for each row affected by a DML statement. The trigger causes the old and new values of the object `t` in `tbl` to be written in `tbl_history` when `tbl` is updated. These old and new values are `:OLD.OBJECT_VALUE` and `:NEW.OBJECT_VALUE`. An update of the table `tbl` is done (each value of `n` is increased by 1). A select from the history table to check that the trigger works is then shown at the end of the example:

```
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;
/
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 tbl SET tbl.n = tbl.n+1
/
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;
/
```

The result of the select shows that all values of column `n` were increased by 1. The value of `m` remains 0. The output of the select is:

```
23-MAY-05 -- old: 1 0 -- new: 2 0
23-MAY-05 -- old: 2 0 -- new: 3 0
23-MAY-05 -- old: 3 0 -- new: 4 0
23-MAY-05 -- old: 4 0 -- new: 5 0
23-MAY-05 -- old: 5 0 -- new: 6 0
```

Triggers and Handling Remote Exceptions

A trigger that accesses a remote site cannot do remote exception handling if the network link is unavailable. For example:

```
CREATE OR REPLACE TRIGGER Example
AFTER INSERT ON emp
FOR EACH ROW
```

```

BEGIN
  When dblink is inaccessible, compilation fails here:
  INSERT INTO emp@Remote VALUES ('x');
EXCEPTION
  WHEN OTHERS THEN
    INSERT INTO Emp_log VALUES ('x');
END;

```

A trigger is compiled when it is created. Thus, if a remote site is unavailable when the trigger must compile, then Oracle Database cannot validate the statement accessing the remote database, and the compilation fails. The previous example exception statement cannot run, because the trigger does not complete compilation.

Because stored subprograms are stored in a compiled form, the work-around for the previous example is as follows:

```

CREATE OR REPLACE TRIGGER Example
  AFTER INSERT ON emp
  FOR EACH ROW
BEGIN
  Insert_row_proc;
END;

CREATE OR REPLACE PROCEDURE Insert_row_proc AS
BEGIN
  INSERT INTO emp@Remote VALUES ('x');
EXCEPTION
  WHEN OTHERS THEN
    INSERT INTO Emp_log VALUES ('x');
END;

```

The trigger in this example compiles successfully and invokes the stored subprogram, which already has a validated statement for accessing the remote database; thus, when the remote INSERT statement fails because the link is down, the exception is caught.

Restrictions on Creating Triggers

Coding triggers requires some restrictions that are not required for standard PL/SQL blocks.

Topics:

- [Maximum Trigger Size](#)
- [SQL Statements Allowed in Trigger Bodies](#)
- [Trigger Restrictions on LONG and LONG RAW Datatypes](#)
- [Trigger Restrictions on Mutating Tables](#)
- [Restrictions on Mutating Tables Relaxed](#)
- [System Trigger Restrictions](#)
- [Foreign Function Callouts](#)

Maximum Trigger Size

The size of a trigger cannot be more than 32K.

SQL Statements Allowed in Trigger Bodies

A trigger body can contain `SELECT INTO` statements, `SELECT` statements in cursor definitions, and all other DML statements.

A system trigger body can contain the DDL statements `CREATETABLE`, `ALTERTABLE`, `DROP TABLE` and `ALTER COMPILER`. A nonsystem trigger body cannot contain DDL or transaction control statements.

Note: A subprogram invoked by a trigger cannot run the previous transaction control statements, because the subprogram runs within the context of the trigger body.

Statements inside a trigger can reference remote schema objects. However, pay special attention when invoking remote subprograms from within a local trigger. If a timestamp or signature mismatch is found during execution of the trigger, then the remote subprogram is not run, and the trigger is invalidated.

Trigger Restrictions on LONG and LONG RAW Datatypes

`LONG` and `LONG RAW` datatypes in triggers are subject to the following restrictions:

- A SQL statement within a trigger can insert data into a column of `LONG` or `LONG RAW` datatype.
- If data from a `LONG` or `LONG RAW` column can be converted to a constrained datatype (such as `CHAR` and `VARCHAR2`), then a `LONG` or `LONG RAW` column can be referenced in a SQL statement within a trigger. The maximum length for these datatypes is 32000 bytes.
- Variables cannot be declared using the `LONG` or `LONG RAW` datatypes.
- `:NEW` and `:PARENT` cannot be used with `LONG` or `LONG RAW` columns.

Trigger Restrictions on Mutating Tables

A **mutating table** is a table that is being modified by an `UPDATE`, `DELETE`, or `INSERT` statement, or a table that might be updated by the effects of a `DELETE CASCADE` constraint.

The session that issued the triggering statement cannot query or modify a mutating table. This restriction prevents a trigger from seeing an inconsistent set of data.

This restriction applies to all triggers that use the `FOR EACH ROW` clause. Views being modified in `INSTEAD OF` triggers are not considered mutating.

When a trigger encounters a mutating table, a run-time error occurs, the effects of the trigger body and triggering statement are rolled back, and control is returned to the user or application. (You can use compound triggers to avoid the mutating-table error. For more information, see ["Using Compound Triggers to Avoid Mutating-Table Error"](#) on page 9-16.)

Consider the following trigger:

```
CREATE OR REPLACE TRIGGER Emp_count
  AFTER DELETE ON emp
  FOR EACH ROW
DECLARE
  n INTEGER;
BEGIN
  SELECT COUNT(*) INTO n FROM emp;
```

```
DBMS_OUTPUT.PUT_LINE('There are now ' || n || ' employees.');
```

```
END;
```

If the following SQL statement is entered:

```
DELETE FROM emp WHERE empno = 7499;
```

An error is returned because the table is mutating when the row is deleted:

```
ORA-04091: table HR.emp is mutating, trigger/function might not see it
```

If you delete the line "FOR EACH ROW" from the trigger, it becomes a statement trigger that is not subject to this restriction, and the trigger.

If you need to update a mutating table, you can bypass these restrictions by using a temporary table, a PL/SQL table, or a package variable. For example, in place of a single AFTER row trigger that updates the original table, resulting in a mutating table error, you might use two triggers—an AFTER row trigger that updates a temporary table, and an AFTER statement trigger that updates the original table with the values from the temporary table.

Declarative constraints are checked at various times with respect to row triggers.

See Also: *Oracle Database Concepts* for information about the interaction of triggers and constraints

Because declarative referential constraints are not supported between tables on different nodes of a distributed database, the mutating table restrictions do not apply to triggers that access remote nodes. These restrictions are also not enforced among tables in the same database that are connected by loop-back database links. A loop-back database link makes a local table appear remote by defining an Oracle Net path back to the database that contains the link.

Restrictions on Mutating Tables Relaxed

The mutating error described in "[Trigger Restrictions on Mutating Tables](#)" on page 9-24 prevents the trigger from reading or modifying the table that the parent statement is modifying. However, as of Oracle Database Release 8.1, a deletion from the parent table causes BEFORE and AFTER triggers to fire once. Therefore, you can create triggers (just not row triggers) to read and modify the parent and child tables.

This allows most foreign key constraint actions to be implemented through their obvious after-row trigger, providing the constraint is not self-referential. Update cascade, update set null, update set default, delete set default, inserting a missing parent, and maintaining a count of children can all be implemented easily. For example, this is an implementation of update cascade:

```
CREATE TABLE p (p1 NUMBER CONSTRAINT pk_p_p1 PRIMARY KEY);
CREATE TABLE f (f1 NUMBER CONSTRAINT fk_f_f1 REFERENCES p);
CREATE TRIGGER pt AFTER UPDATE ON p FOR EACH ROW BEGIN
    UPDATE f SET f1 = :NEW.p1 WHERE f1 = :OLD.p1;
END;
/
```

This implementation requires care for multiple-row updates. For example, if table *p* has three rows with the values (1), (2), (3), and table *f* also has three rows with the values (1), (2), (3), then the following statement updates *p* correctly but causes problems when the trigger updates *f*:

```
UPDATE p SET p1 = p1+1;
```

The statement first updates (1) to (2) in *p*, and the trigger updates (1) to (2) in *f*, leaving two rows of value (2) in *f*. Then the statement updates (2) to (3) in *p*, and the trigger updates both rows of value (2) to (3) in *f*. Finally, the statement updates (3) to (4) in *p*, and the trigger updates all three rows in *f* from (3) to (4). The relationship of the data in *p* and *f* is lost.

To avoid this problem, either forbid multiple-row updates to *p* that change the primary key and re-use existing primary key values, or track updates to foreign key values and modify the trigger to ensure that no row is updated twice.

That is the only problem with this technique for foreign key updates. The trigger cannot miss rows that were changed but not committed by another transaction, because the foreign key constraint guarantees that no matching foreign key rows are locked before the after-row trigger is invoked.

System Trigger Restrictions

Depending on the event, different event attribute functions are available. For example, certain DDL operations might not be allowed on DDL events. Check "[Event Attribute Functions](#)" on page 9-46 before using an event attribute function, because its effects might be undefined rather than producing an error condition.

Only committed triggers fire. For example, if you create a trigger that fires after all CREATE events, then the trigger itself does not fire after the creation, because the correct information about this trigger was not committed at the time when the trigger on CREATE events fired.

For example, if you execute the following SQL statement:

```
CREATE OR REPLACE TRIGGER my_trigger AFTER CREATE ON DATABASE
BEGIN null;
END;
```

Then, trigger `my_trigger` does not fire after the creation of `my_trigger`. Oracle Database does not fire a trigger that is not committed.

Foreign Function Callouts

All restrictions on foreign function callouts also apply.

Who Uses the Trigger?

The following statement, inside a trigger, returns the owner of the trigger, not the name of user who is updating the table:

```
SELECT Username FROM USER_USERS;
```

Compiling Triggers

An important difference between triggers and PL/SQL anonymous blocks is their compilation. An anonymous block is compiled each time it is loaded into memory, and its compilation has three stages:

1. Syntax checking: PL/SQL syntax is checked, and a parse tree is generated.
2. Semantic checking: Type checking and further processing on the parse tree.
3. Code generation

A trigger is fully compiled when the `CREATE TRIGGER` statement executes. The trigger code is stored in the data dictionary. Therefore, it is unnecessary to open a shared cursor in order to execute the trigger; the trigger executes directly.

If an error occurs during the compilation of a trigger, the trigger is still created. Therefore, if a DML statement fires the trigger, the DML statement fails (unless the trigger was created in the disabled state). To see trigger compilation errors, either use the `SHOW ERRORS` statement in SQL*Plus or Enterprise Manager, or `SELECT` the errors from the `USER_ERRORS` view.

Topics:

- [Dependencies for Triggers](#)
- [Recompiling Triggers](#)

Dependencies for Triggers

Compiled triggers have dependencies. They become invalid if a depended-on object, such as a stored subprogram invoked from the trigger body, is modified. Triggers that are invalidated for dependency reasons are recompiled when next invoked.

You can examine the `ALL_DEPENDENCIES` view to see the dependencies for a trigger. For example, the following statement shows the dependencies for the triggers in the HR schema:

```
SELECT NAME, REFERENCED_OWNER, REFERENCED_NAME, REFERENCED_TYPE
       FROM ALL_DEPENDENCIES
       WHERE OWNER = 'HR' and TYPE = 'TRIGGER';
```

Triggers might depend on other functions or packages. If the function or package specified in the trigger is dropped, then the trigger is marked invalid. An attempt is made to validate the trigger on occurrence of the event. If the trigger cannot be validated successfully, then it is marked `VALID WITH ERRORS`, and the event fails. For more information about dependencies between schema objects, see *Oracle Database Concepts*.

Note:

- There is an exception for `STARTUP` events: `STARTUP` events succeed even if the trigger fails. There are also exceptions for `SHUTDOWN` events and for `LOGON` events if you login as `SYSTEM`.
 - Because the `DBMS_AQ` package is used to enqueue a message, dependency between triggers and queues cannot be maintained.
-
-

Recompiling Triggers

Use the `ALTER TRIGGER` statement to recompile a trigger manually. For example, the following statement recompiles the `PRINT_SALARY_CHANGES` trigger:

```
ALTER TRIGGER Print_salary_changes COMPILE;
```

To recompile a trigger, you must own the trigger or have the `ALTER ANY TRIGGER` system privilege.

Modifying Triggers

Like a stored subprogram, a trigger cannot be explicitly altered: It must be replaced with a new definition. (The `ALTER TRIGGER` statement is used only to recompile, enable, or disable a trigger.)

When replacing a trigger, you must include the `OR REPLACE` option in the `CREATE TRIGGER` statement. The `OR REPLACE` option is provided to allow a new version of an existing trigger to replace the older version, without affecting any grants made for the original version of the trigger.

Alternatively, the trigger can be dropped using the `DROP TRIGGER` statement, and you can rerun the `CREATE TRIGGER` statement.

To drop a trigger, the trigger must be in your schema, or you must have the `DROP ANY TRIGGER` system privilege.

Debugging Triggers

You can debug a trigger using the same facilities available for stored subprograms. See *Oracle Database Advanced Application Developer's Guide*.

Enabling Triggers

To enable a disabled trigger, use the `ALTER TRIGGER` statement with the `ENABLE` clause. For example, to enable the disabled trigger named `Reorder`, enter the following statement:

```
ALTER TRIGGER Reorder ENABLE;
```

To enable all triggers defined for a specific table, use the `ALTER TABLE` statement with the `ENABLE` clause and the `ALL TRIGGERS` option. For example, to enable all triggers defined for the `Inventory` table, enter the following statement:

```
ALTER TABLE Inventory ENABLE ALL TRIGGERS;
```

Disabling Triggers

You might temporarily disable a trigger if:

- An object it references is not available.
- You need to perform a large data load, and you want it to proceed quickly without firing triggers.
- You are reloading data.

To disable a trigger, use the `ALTER TRIGGER` statement with the `DISABLE` option. For example, to disable the trigger named `Reorder`, enter the following statement:

```
ALTER TRIGGER Reorder DISABLE;
```

To disable all triggers defined for a specific table, use the `ALTER TABLE` statement with the `DISABLE` clause and the `ALL TRIGGERS` option. For example, to disable all triggers defined for the `Inventory` table, enter the following statement:

```
ALTER TABLE Inventory DISABLE ALL TRIGGERS;
```


Viewing Information About Triggers

The *_TRIGGERS static data dictionary views reveal information about triggers.

The column BASE_OBJECT_TYPE specifies whether the trigger is based on DATABASE, SCHEMA, table, or view. The column TABLE_NAME is null if the base object is not table or view.

The column ACTION_TYPE specifies whether the trigger is a call type trigger or a PL/SQL trigger.

The column TRIGGER_TYPE specifies the type of the trigger; for example COMPOUND, BEFORE EVENT, or AFTER EVENT (the last two apply only to database events).

Each of the columns BEFORE_STATEMENT, BEFORE_ROW, AFTER_ROW, AFTER_STATEMENT, and INSTEAD_OF_ROW has the value YES or NO.

The column TRIGGERING_EVENT includes all system and DML events.

See Also: *Oracle Database Reference* for a complete description of the *_TRIGGERS static data dictionary views

For example, assume the following statement was used to create the Reorder trigger:

Caution: You might need to set up data structures for certain examples to work:

```
CREATE OR REPLACE TRIGGER Reorder
AFTER UPDATE OF Parts_on_hand ON Inventory
FOR EACH ROW
WHEN(new.Parts_on_hand < new.Reorder_point)
DECLARE
    x NUMBER;
BEGIN
    SELECT COUNT(*) INTO x
    FROM Pending_orders
    WHERE Part_no = :new.Part_no;
    IF x = 0 THEN
        INSERT INTO Pending_orders
        VALUES (:new.Part_no, :new.Reorder_quantity,
        sysdate);
    END IF;
END;
```

The following two queries return information about the REORDER trigger:

```
SELECT Trigger_type, Triggering_event, Table_name
FROM USER_TRIGGERS
WHERE Trigger_name = 'REORDER';
```

TYPE	TRIGGERING_STATEMENT	TABLE_NAME
AFTER EACH ROW	UPDATE	INVENTORY

```
SELECT Trigger_body
FROM USER_TRIGGERS
WHERE Trigger_name = 'REORDER';
```

```
TRIGGER_BODY
-----
```

```
DECLARE
  x NUMBER;
BEGIN
  SELECT COUNT(*) INTO x
  FROM Pending_orders
  WHERE Part_no = :new.Part_no;
  IF x = 0
  THEN INSERT INTO Pending_orders
  VALUES (:new.Part_no, :new.Reorder_quantity,
  sysdate);
  END IF;
END;
```

Examples of Trigger Applications

You can use triggers in a number of ways to customize information management in Oracle Database. For example, triggers are commonly used to:

- Provide sophisticated auditing
- Prevent invalid transactions
- Enforce referential integrity (either those actions not supported by declarative constraints or across nodes in a distributed database)
- Enforce complex business rules
- Enforce complex security authorizations
- Provide transparent event logging
- Automatically generate derived column values
- Enable building complex views that are updatable
- Track database events

This section provides an example of each of these trigger applications. These examples are not meant to be used exactly as written: They are provided to assist you in designing your own triggers.

Auditing with Triggers

Triggers are commonly used to supplement the built-in auditing features of Oracle Database. Although triggers can be written to record information similar to that recorded by the `AUDIT` statement, use triggers only when more detailed audit information is required. For example, use triggers to provide value-based auditing for each row.

Sometimes, the `AUDIT` statement is considered a *security* audit facility, while triggers can provide *financial* audit facility.

When deciding whether to create a trigger to audit database activity, consider what Oracle Database's auditing features provide, compared to auditing defined by triggers, as shown in [Table 9–2](#).

Table 9–2 Comparison of Built-in Auditing and Trigger-Based Auditing

Audit Feature	Description
DML and DDL Auditing	Standard auditing options permit auditing of DML and DDL statements regarding all types of schema objects and structures. Comparatively, <i>triggers</i> permit auditing of DML statements entered against tables, and DDL auditing at SCHEMA or DATABASE level.
Centralized Audit Trail	All database audit information is recorded centrally and automatically using the auditing features of Oracle Database.
Declarative Method	Auditing features enabled using the standard Oracle Database features are easier to declare and maintain, and less prone to errors, when compared to auditing functions defined by triggers.
Auditing Options can be Audited	Any changes to existing auditing options can also be audited to guard against malicious database activity.
Session and Execution time Auditing	Using the database auditing features, records can be generated once every time an audited statement is entered (BY ACCESS) or once for every session that enters an audited statement (BY SESSION). Triggers cannot audit by session; an audit record is generated each time a trigger-audited table is referenced.
Auditing of Unsuccessful Data Access	Database auditing can be set to audit when unsuccessful data access occurs. However, unless autonomous transactions are used, any audit information generated by a trigger is rolled back if the triggering statement is rolled back. For more information on autonomous transactions, see <i>Oracle Database Concepts</i> .
Sessions can be Audited	Connections and disconnections, as well as session activity (physical I/Os, logical I/Os, deadlocks, and so on), can be recorded using standard database auditing.

When using triggers to provide sophisticated auditing, AFTER triggers are normally used. The triggering statement is subjected to any applicable constraints. If no records are found, then the AFTER trigger does not fire, and audit processing is not carried out unnecessarily.

Choosing between AFTER row and AFTER statement triggers depends on the information being audited. For example, row triggers provide value-based auditing for each table row. Triggers can also require the user to supply a "reason code" for issuing the audited SQL statement, which can be useful in both row and statement-level auditing situations.

The following example demonstrates a trigger that audits modifications to the emp table for each row. It requires that a "reason code" be stored in a global package variable before the update. This shows how triggers can be used to provide value-based auditing and how to use public package variables.

Note: You might need to set up the following data structures for the examples to work:

```
CREATE OR REPLACE PACKAGE Auditpackage AS
    Reason VARCHAR2(10);
PROCEDURE Set_reason(Reason VARCHAR2);
END;
CREATE TABLE Emp99 (
    Empno          NOT NULL   NUMBER(4),
    Ename          VARCHAR2(10),
    Job            VARCHAR2(9),
    Mgr            NUMBER(4),
    Hiredate       DATE,
    Sal            NUMBER(7,2),
    Comm           NUMBER(7,2),
    Deptno         NUMBER(2),
    Bonus          NUMBER,
    Ssn            NUMBER,
    Job_classification NUMBER);
```

```
CREATE TABLE Audit_employee (
    Oldssn         NUMBER,
    Oldname        VARCHAR2(10),
    Oldjob         VARCHAR2(2),
    Oldsal         NUMBER,
    Newssn         NUMBER,
    Newname        VARCHAR2(10),
    Newjob         VARCHAR2(2),
    Newsal         NUMBER,
    Reason         VARCHAR2(10),
    User1          VARCHAR2(10),
    Systemdate     DATE);
```

```
CREATE OR REPLACE TRIGGER Audit_employee
AFTER INSERT OR DELETE OR UPDATE ON Emp99
FOR EACH ROW
BEGIN
    /* AUDITPACKAGE is a package with a public package
    variable REASON. REASON can be set by the
    application by a statement such as EXECUTE
    AUDITPACKAGE.SET_REASON(reason_string).
    A package variable has state for the duration of a
    session and that each session has a separate copy of
    all package variables. */

    IF Auditpackage.Reason IS NULL THEN
        Raise_application_error(-20201, 'Must specify reason'
            || ' with AUDITPACKAGE.SET_REASON(Reason_string)');
    END IF;

    /* If preceding condition evaluates to TRUE,
    user-specified error number & message is raised,
    trigger stops execution, & effects of triggering statement are rolled back.
    Otherwise, new row is inserted
    into predefined auditing table named AUDIT_EMPLOYEE
    containing existing & new values of the emp table
    & reason code defined by REASON variable of AUDITPACKAGE.
    "Old" values are NULL if triggering statement is INSERT
    & "new" values are NULL if triggering statement is DELETE. */
```

```

INSERT INTO Audit_employee VALUES (
    :old.Ssn, :old.Ename, :old.Job_classification, :old.Sal,
    :new.Ssn, :new.Ename, :new.Job_classification, :new.Sal,
    auditpackage.Reason, User, Sysdate
);
END;

```

Optionally, you can also set the reason code back to NULL if you wanted to force the reason code to be set for every update. The following simple AFTER statement trigger sets the reason code back to NULL after the triggering statement is run:

```

CREATE OR REPLACE TRIGGER Audit_employee_reset
    AFTER INSERT OR DELETE OR UPDATE ON emp
BEGIN
    auditpackage.set_reason(NULL);
END;

```

Notice that the previous two triggers are fired by the same type of SQL statement. However, the AFTER row trigger fires once for each row of the table affected by the triggering statement, while the AFTER statement trigger fires only once after the triggering statement execution is completed.

This next trigger also uses triggers to do auditing. It tracks changes made to the emp table and stores this information in AUDIT_TABLE and AUDIT_TABLE_VALUES.

Note: You might need to set up the following data structures for the example to work:

```

CREATE TABLE Audit_table (
    Seq      NUMBER,
    User_at  VARCHAR2(10),
    Time_now DATE,
    Term     VARCHAR2(10),
    Job      VARCHAR2(10),
    Proc     VARCHAR2(10),
    enum     NUMBER);
CREATE SEQUENCE Audit_seq;
CREATE TABLE Audit_table_values (
    Seq      NUMBER,
    Dept     NUMBER,
    Dept1    NUMBER,
    Dept2    NUMBER);

```

```

CREATE OR REPLACE TRIGGER Audit_emp
    AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW
DECLARE
    Time_now DATE;
    Terminal CHAR(10);
BEGIN
    -- Get current time, & terminal of user:
    Time_now := SYSDATE;
    Terminal := USERENV('TERMINAL');

    -- Record new employee primary key:
    IF INSERTING THEN
        INSERT INTO Audit_table VALUES (
            Audit_seq.NEXTVAL, User, Time_now,

```

```
        Terminal, 'emp', 'INSERT', :new.Empno
    );

    -- Record primary key of deleted row:
    ELSIF DELETING THEN
        INSERT INTO Audit_table VALUES (
            Audit_seq.NEXTVAL, User, Time_now,
            Terminal, 'emp', 'DELETE', :old.Empno
        );

    -- For updates, record primary key of row being updated:
    ELSE
        INSERT INTO Audit_table VALUES (
            audit_seq.NEXTVAL, User, Time_now,
            Terminal, 'emp', 'UPDATE', :old.Empno
        );

        -- For SAL & DEPTNO, record old & new values:
        IF UPDATING ('SAL') THEN
            INSERT INTO Audit_table_values VALUES (
                Audit_seq.CURRVAL, 'SAL',
                :old.Sal, :new.Sal
            );

            ELSIF UPDATING ('DEPTNO') THEN
                INSERT INTO Audit_table_values VALUES (
                    Audit_seq.CURRVAL, 'DEPTNO',
                    :old.Deptno, :new.DEPTNO
                );
            END IF;
        END IF;
    END IF;
END;
```

Constraints and Triggers

Triggers and declarative constraints can both be used to constrain data input. However, triggers and constraints have significant differences.

Declarative constraints are statements about the database that are always true. A constraint applies to existing data in the table and any statement that manipulates the table.

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

Triggers constrain what a transaction can do. A trigger does not apply to data loaded before the definition of the trigger; therefore, it is not known if all data in a table conforms to the rules established by an associated trigger.

Although triggers can be written to enforce many of the same rules supported by Oracle Database's declarative constraint features, use triggers only to enforce complex business rules that cannot be defined using standard constraints. The declarative constraint features provided with Oracle Database offer the following advantages when compared to constraints defined by triggers:

- Centralized integrity checks
 - All points of data access must adhere to the global set of rules defined by the constraints corresponding to each schema object.
- Declarative method

Constraints defined using the standard constraint features are much easier to write and are less prone to errors, when compared with comparable constraints defined by triggers.

While most aspects of data integrity can be defined and enforced using declarative constraints, triggers can be used to enforce complex business constraints not definable using declarative constraints. For example, triggers can be used to enforce:

- UPDATE SET NULL, and UPDATE and DELETE SET DEFAULT referential actions.
- Referential integrity when the parent and child tables are on different nodes of a distributed database.
- Complex check constraints not definable using the expressions allowed in a CHECK constraint.

Referential Integrity Using Triggers

Use triggers only when performing an action for which there is no declarative support.

When using triggers to maintain referential integrity, declare the PRIMARY (or UNIQUE) KEY constraint in the parent table. If referential integrity is being maintained between a parent and child table in the same database, then you can also declare the foreign key in the child table, but disable it; this prevents the corresponding PRIMARY KEY constraint from being dropped (unless the PRIMARY KEY constraint is explicitly dropped with the CASCADE option).

To maintain referential integrity using triggers:

- A trigger must be defined for the child table that guarantees values inserted or updated in the foreign key correspond to values in the parent key.
- One or more triggers must be defined for the parent table. These triggers guarantee the desired referential action (RESTRICT, CASCADE, or SET NULL) for values in the foreign key when values are updated or deleted in the parent key. No action is required for inserts into the parent table (no dependent foreign keys exist).

The following sections provide examples of the triggers necessary to enforce referential integrity. The emp and dept table relationship is used in these examples.

Several of the triggers include statements that lock rows (SELECT FOR UPDATE). This operation is necessary to maintain concurrency as the rows are being processed.

Foreign Key Trigger for Child Table

The following trigger guarantees that before an INSERT or UPDATE statement affects a foreign key value, the corresponding value exists in the parent key. The mutating table exception included in the following example allows this trigger to be used with the UPDATE_SET_DEFAULT and UPDATE_CASCADE triggers. This exception can be removed if this trigger is used alone.

```
CREATE OR REPLACE TRIGGER Emp_dept_check
  BEFORE INSERT OR UPDATE OF Deptno ON emp
  FOR EACH ROW WHEN (new.Deptno IS NOT NULL)

  -- Before row is inserted or DEPTNO is updated in emp table,
  -- fire this trigger to verify that new foreign key value (DEPTNO)
  -- is present in dept table.
DECLARE
  Dummy          INTEGER; -- Use for cursor fetch
  Invalid_department EXCEPTION;
```

```

Valid_department    EXCEPTION;
Mutating_table      EXCEPTION;
PRAGMA EXCEPTION_INIT (Mutating_table, -4091);

-- Cursor used to verify parent key value exists.
-- If present, lock parent key's row so it cannot be deleted
-- by another transaction until this transaction is
-- committed or rolled back.
CURSOR Dummy_cursor (Dn NUMBER) IS
  SELECT Deptno FROM dept
     WHERE Deptno = Dn
     FOR UPDATE OF Deptno;
BEGIN
  OPEN Dummy_cursor (:new.Deptno);
  FETCH Dummy_cursor INTO Dummy;

  -- Verify parent key.
  -- If not found, raise user-specified error number & message.
  -- If found, close cursor before allowing triggering statement to complete:
  IF Dummy_cursor%NOTFOUND THEN
    RAISE Invalid_department;
  ELSE
    RAISE valid_department;
  END IF;
  CLOSE Dummy_cursor;
EXCEPTION
  WHEN Invalid_department THEN
    CLOSE Dummy_cursor;
    Raise_application_error(-20000, 'Invalid Department'
      || ' Number' || TO_CHAR(:new.deptno));
  WHEN Valid_department THEN
    CLOSE Dummy_cursor;
  WHEN Mutating_table THEN
    NULL;
END;
```

UPDATE and DELETE RESTRICT Trigger for Parent Table

The following trigger is defined on the dept table to enforce the UPDATE and DELETE RESTRICT referential action on the primary key of the dept table:

```

CREATE OR REPLACE TRIGGER Dept_restrict
  BEFORE DELETE OR UPDATE OF Deptno ON dept
  FOR EACH ROW

-- Before row is deleted from dept or primary key (DEPTNO) of dept is updated,
-- check for dependent foreign key values in emp;
-- if any are found, roll back.

DECLARE
  Dummy                INTEGER; -- Use for cursor fetch
  Employees_present    EXCEPTION;
  employees_not_present EXCEPTION;

  -- Cursor used to check for dependent foreign key values.
  CURSOR Dummy_cursor (Dn NUMBER) IS
    SELECT Deptno FROM emp WHERE Deptno = Dn;

BEGIN
  OPEN Dummy_cursor (:old.Deptno);
  FETCH Dummy_cursor INTO Dummy;
```



```

-- If dependent foreign key is found, raise user-specified
-- error number and message. If not found, close cursor
-- before allowing triggering statement to complete.

IF Dummy_cursor%FOUND THEN
    RAISE Employees_present;    -- Dependent rows exist
ELSE
    RAISE Employees_not_present; -- No dependent rows exist
END IF;
CLOSE Dummy_cursor;

EXCEPTION
WHEN Employees_present THEN
    CLOSE Dummy_cursor;
    Raise_application_error(-20001, 'Employees Present in'
        || ' Department ' || TO_CHAR(:old.DEPTNO));
WHEN Employees_not_present THEN
    CLOSE Dummy_cursor;
END;
```

Caution: This trigger 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, A fires B fires A).

UPDATE and DELETE SET NULL Triggers for Parent Table

The following trigger is defined on the dept table to enforce the UPDATE and DELETE SET NULL referential action on the primary key of the dept 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 following trigger on the dept table enforces the DELETE CASCADE referential action on the primary key of the dept 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 following trigger ensures that if a department number is updated in the dept table, then this change is propagated to dependent foreign keys in the emp 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 new 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 already 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.Updateidseq;
END;

```

Note: Because this trigger updates the emp table, the Emp_dept_check trigger, 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.

Trigger for Complex Check Constraints

Triggers can enforce integrity rules other than referential integrity. For example, this trigger performs a complex check before allowing the triggering statement to run.

Note: You might need to set up the following data structures for the example to work:

```

CREATE OR REPLACE TABLE Salgrade (
    Grade          NUMBER,
    Losal          NUMBER,
    Hisal          NUMBER,
    Job_classification NUMBER);

CREATE OR REPLACE TRIGGER Salary_check
    BEFORE INSERT OR UPDATE OF Sal, Job ON Emp99
    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 '
        ||:new.Job_classification);
END;
```

Complex Security Authorizations and Triggers

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 Oracle Database. For example, a trigger can prohibit updates to salary data of the emp table during weekends, holidays, and nonworking hours.

When using a trigger to enforce a complex security authorization, it is best to use a BEFORE statement trigger. Using a BEFORE statement trigger has these benefits:

- The security check is done before the triggering statement is allowed to run, so that no wasted work is done by an unauthorized statement.
- The security check is performed only once for the triggering statement, not for each row affected by the triggering statement.

This example shows a trigger used to enforce security.

Note: You might need to set up the following data structures for the example to work:

```
CREATE TABLE Company_holidays (Day DATE);
```

```
CREATE OR REPLACE TRIGGER Emp_permit_changes
    BEFORE INSERT OR DELETE OR UPDATE ON Emp99
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 details on database security features

Transparent Event Logging and Triggers

Triggers are very useful when you want to transparently perform a related change in the database following certain events.

The REORDER trigger example shows a trigger that reorders parts as necessary when certain conditions are met. (In other words, a triggering statement is entered, and the PARTS_ON_HAND value is less than the REORDER_POINT value.)

Derived Column Values and Triggers

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 the following 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 following example illustrates how a trigger can be used to derive new column values for a table whenever a row is inserted or updated.

Note: You might need to set up the following data structures for the example to work:

```

ALTER TABLE Emp99 ADD(
    Uppername VARCHAR2(20),
    Soundexname VARCHAR2(20));
```

```

CREATE OR REPLACE TRIGGER Derived
BEFORE INSERT OR UPDATE OF Ename ON Emp99

/* 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;
```

Building Complex Updatable Views Using Triggers

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 under their respective titles. The library consists of a collection of book type objects. The following example explains the schema.

```
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;
```

Assume that the following tables exist in the relational schema:

Table `Book_table` (Booknum, Section, Title, Author, Available)

Booknum	Section	Title	Author	Available
121001	Classic	Iliad	Homer	Y
121002	Novel	Gone with the Wind	Mitchell M	N

Library consists of `library_table(section)`.

Section

Geography

Classic

You can define a complex view over these tables 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;
```

Make this view updatable by defining an `INSTEAD OF` trigger over the view.

```
CREATE OR REPLACE TRIGGER Library_trigger INSTEAD OF INSERT ON Library_view FOR
EACH ROW
  Bookvar BOOK_T;
  i      INTEGER;
BEGIN
  INSERT INTO Library_table VALUES (:NEW.Section);
  FOR i IN 1..:NEW.Booklist.COUNT LOOP
    Bookvar := Booklist(i);
    INSERT INTO book_table
      VALUES ( Bookvar.booknum, :NEW.Section, Bookvar.Title, Bookvar.Author,
bookvar.Available);
```

```

        END LOOP;
    END;
/

```

The `library_view` is an updatable view, and any INSERTs on the view are handled by the trigger that fires automatically. For example:

```

INSERT INTO Library_view VALUES ('History', book_list_t(book_t(121330,
'Alexander', 'Mirth', 'Y')));

```

Similarly, you can also define triggers on the nested table `booklist` to handle modification of the nested table element.

Fine-Grained Access Control Using Triggers

System triggers can be used to set application context. Application context is a relatively new feature that enhances your ability to implement fine-grained access control. Application context is a secure session cache, and it can be used to store session-specific attributes.

In the example that follows, procedure `set_ctx` sets the application context based on the user profile. The trigger `setexpensectx` ensures that the context is set for every user.

```

CONNECT secdemo/password

CREATE OR REPLACE CONTEXT Expenses_reporting USING Secdemo.Exprep_ctx;

REM =====
REM Creation of the package that implements the context:
REM =====

CREATE OR REPLACE PACKAGE Exprep_ctx AS
    PROCEDURE Set_ctx;
END;

SHOW ERRORS

CREATE OR REPLACE PACKAGE BODY Exprep_ctx IS
    PROCEDURE Set_ctx IS
        Empnum    NUMBER;
        Countrec  NUMBER;
        Cc        NUMBER;
        Role      VARCHAR2(20);
    BEGIN

        -- SET emp_number:
        SELECT Employee_id INTO Empnum FROM Employee
            WHERE Last_name = SYS_CONTEXT('userenv', 'session_user');

        DBMS_SESSION.SET_CONTEXT('expenses_reporting', 'emp_number', Empnum);

        -- SET ROLE:
        SELECT COUNT (*) INTO Countrec FROM Cost_center WHERE Manager_id=Empnum;
        IF (countrec > 0) THEN
            DBMS_SESSION.SET_CONTEXT('expenses_reporting', 'exp_role', 'MANAGER');
        ELSE
            DBMS_SESSION.SET_CONTEXT('expenses_reporting', 'exp_role', 'EMPLOYEE');
        END IF;

        -- SET cc_number:

```

```

SELECT Cost_center_id INTO Cc FROM Employee
      WHERE Last_name = SYS_CONTEXT('userenv','session_user');
DBMS_SESSION.SET_CONTEXT(expenses_reporting','cc_number',Cc);
END;
END;

```

Call syntax:

```

CREATE OR REPLACE TRIGGER Secdemo.Setexpseetx
AFTER LOGON ON DATABASE
CALL Secdemo.Exprep_etx.Set_otx

```

Responding to Database Events Through Triggers

Note: This topic applies only to simple triggers.

Database event publication lets applications subscribe to database events, just like they subscribe to messages from other applications. The database events publication framework includes the following features:

- Infrastructure for publish/subscribe, by making the database an active publisher of events.
- Integration of data cartridges in the server. The database events publication can be used to notify cartridges of state changes in the server.
- Integration of fine-grained access control in the server.

By creating a trigger, you can specify a subprogram that runs when an event occurs. DML events are supported on tables, and database events are supported on `DATABASE` and `SCHEMA`. You can turn notification on and off by enabling and disabling the trigger using the `ALTER TRIGGER` statement.

This feature is integrated with the Advanced Queuing engine. Publish/subscribe applications use the `DBMS_AQ.ENQUEUE` procedure, and other applications such as cartridges use callouts.

See Also:

- *Oracle Database SQL Language Reference* for more information about the `ALTER TRIGGER` statement
- *Oracle Streams Advanced Queuing User's Guide* for details on how to subscribe to published events

Topics:

- [How Events Are Published Through Triggers](#)
- [Publication Context](#)
- [Error Handling](#)
- [Execution Model](#)
- [Event Attribute Functions](#)
- [Database Events](#)

- [Client Events](#)

How Events Are Published Through Triggers

When the database detects an event, the trigger mechanism executes the action specified in the trigger. The action can include publishing the event to a queue so that subscribers receive notifications. To publish events, use the `DBMS_AQ` package.

Note: The database can detect only system-defined events. You cannot define your own events.

When it detects an event, the database fires all triggers that are enabled on that event, except the following:

- 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, within the same transaction as the triggering event.

For example, recursive DDL within a system trigger might modify a trigger, which prevents the modified trigger from being fired by events within the same transaction.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_AQ` package

Publication Context

When an event is published, certain run-time context and attributes, as specified in the parameter list, are passed to the callout subprogram. A set of functions called event attribute functions are provided.

See Also: "[Event Attribute Functions](#)" on page 9-46 for information on event-specific attributes

For each supported database event, you can identify and predefine event-specific attributes for the event. You can choose the parameter list to be any of these attributes, along with other simple expressions. For callouts, these are passed as `IN` arguments.

Error Handling

Return status from publication callout functions for all events are ignored. For example, with `SHUTDOWN` events, the database cannot do anything with the return status.

Execution Model

Traditionally, triggers execute as the definer of the trigger. The trigger action of an event is executed as the definer of the action (as the definer of the package or function in callouts, or as owner of the trigger in queues). Because the owner of the trigger must have `EXECUTE` privileges on the underlying queues, packages, or subprograms, this action is consistent.

Event Attribute Functions

When the database fires a trigger, you can retrieve certain attributes about the event that fired the trigger. You can retrieve each attribute with a function call. [Table 9–3](#) describes the system-defined event attributes.

Note:

- These attributes are available only if the CATPROC.SQL script was run. To verify that it was run, a DBA can use the following query to check that the database component CATPROC has the status VALID and the same version as the current server version:


```
SELECT COMP_ID, STATUS, VERSION FROM DBA_REGISTRY;
```
- The trigger dictionary object maintains metadata about events that will be published and their corresponding attributes.
- In earlier releases, these functions were accessed through the SYS package. We recommend you use these public synonyms whose names begin with ora_.
- ora_name_list_t is defined in package DBMS_STANDARD as


```
TYPE ora_name_list_t IS TABLE OF VARCHAR2(64);
```

Table 9–3 System-Defined Event Attributes

Attribute	Type	Description	Example
ora_client_ip_address	VARCHAR2	Returns IP address of the client in a LOGON event when the underlying protocol is TCP/IP	<pre>DECLARE v_addr VARCHAR2(11); BEGIN IF (ora_sysevent = 'LOGON') THEN v_addr := ora_client_ip_address; END IF; END;</pre>
ora_database_name	VARCHAR2(50)	Database name.	<pre>DECLARE v_db_name VARCHAR2(50); BEGIN v_db_name := ora_database_name; END;</pre>
ora_des_encrypted_password	VARCHAR2	The DES-encrypted password of the user being created or altered.	<pre>IF (ora_dict_obj_type = 'USER') THEN INSERT INTO event_table VALUES (ora_des_encrypted_password); END IF;</pre>
ora_dict_obj_name	VARCHAR(30)	Name of the dictionary object on which the DDL operation occurred.	<pre>INSERT INTO event_table VALUES ('Changed object is ' ora_dict_obj_name);</pre>
ora_dict_obj_name_list (name_list OUT ora_name_list_t)	PLS_INTEGER	Return the list of object names of objects being modified in the event.	<pre>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;</pre>

Table 9-3 (Cont.) System-Defined Event Attributes

Attribute	Type	Description	Example
ora_dict_obj_owner	VARCHAR(30)	Owner of the dictionary object on which the 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	Returns the list of object owners of objects being modified in the event.	DECLARE owner_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(owner_list); END IF; END;
ora_dict_obj_type	VARCHAR(20)	Type of the dictionary object on which the 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	Returns the grantees of a grant event in the OUT parameter; returns the number of grantees in the return value.	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;
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	Returns true if the specified column is altered.	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	Returns true if the current event is creating a nested table	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	Returns true if the specified column is dropped.	IF (ora_sysevent = 'ALTER' AND ora_dict_obj_type = 'TABLE') THEN drop_column := ora_is_drop_column('C'); END IF;
ora_is_servererror	BOOLEAN	Returns 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 an INSTEAD OF trigger for CREATE TABLE, the position within the SQL text where you can insert a PARTITION clause.	-- Retrieve ora_sql_txt into -- sql_text variable first. v_n := ora_partition_pos; v_new_stmt := SUBSTR(sql_text,1,v_n - 1) ' ' my_partition_clause ' ' SUBSTR(sql_text, v_n);

Table 9–3 (Cont.) System-Defined Event Attributes

Attribute	Type	Description	Example
ora_privilege_list (privilege_list OUT ora_name_list_t)	PLS_INTEGER	Returns the list of privileges being granted by the grantee or the list of privileges revoked from the revokees in the OUT parameter; returns the number of privileges in the return value.	<pre> DECLARE privelege_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;</pre>
ora_revokee (user_list OUT ora_name_list_t)	PLS_INTEGER	Returns the revokees of a revoke event in the OUT parameter; returns the number of revokees in the return value.	<pre> 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;</pre>
ora_server_error	NUMBER	Given a position (1 for top of stack), it returns the error number at that position on error stack	<pre> INSERT INTO event_table VALUES ('top stack error ' ora_server_error(1));</pre>
ora_server_error_depth	PLS_INTEGER	Returns the total number of error messages on the error stack.	<pre> n := ora_server_error_depth; -- This value is used with other functions -- such as ora_server_error</pre>
ora_server_error_msg (position in pls_integer)	VARCHAR2	Given a position (1 for top of stack), it returns the error message at that position on error stack	<pre> INSERT INTO event_table VALUES ('top stack error message' ora_server_error_msg(1));</pre>
ora_server_error_num_params (position in pls_integer)	PLS_INTEGER	Given a position (1 for top of stack), it returns the number of strings that were substituted into the error message using a format like %s.	<pre> n := ora_server_error_num_params(1);</pre>
ora_server_error_param (position in pls_integer, param in pls_integer)	VARCHAR2	Given a position (1 for top of stack) and a parameter number, returns the matching substitution value (%s, %d, and so on) in the error message.	<pre> -- For example, the second %s in a -- message: "Expected %s, found %s" param := ora_server_error_param(1,2);</pre>

Table 9–3 (Cont.) System-Defined Event Attributes

Attribute	Type	Description	Example
ora_sql_txt (sql_text out ora_name_list_t)	PLS_INTEGER	Returns the SQL text of the triggering statement in the OUT parameter. If the statement is long, it is broken into multiple PL/SQL table elements. The function return value shows the number of elements are in the PL/SQL table.	--... -- Create table event_table 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)	Database event firing the trigger: Event name is same as that in the syntax.	INSERT INTO event_table VALUES (ora_sysevent);
ora_with_grant_option	BOOLEAN	Returns true if the privileges are granted with grant option.	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	Returns true if the error is related to an out-of-space condition, and fills in the OUT parameters with information about the object that caused the error.	IF (space_error_info(eno, typ, owner, ts, obj, subobj) = TRUE) THEN DBMS_OUTPUT.PUT_LINE('The object ' obj ' owned by ' owner ' has run out of space.');

Database Events

Database events are related to entire instances or schemas, not individual tables or rows. Triggers associated with startup and shutdown events must be defined on the database instance. Triggers associated with on-error and suspend events can be defined on either the database instance or a particular schema.

Table 9–4 Database Events

Event	When Trigger Fires	Conditions	Restrictions	Transaction	Attribute Functions
STARTUP	When the database is opened.	None allowed	No database operations allowed in the trigger. Return status ignored.	Starts a separate transaction and commits it after firing the triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name
SHUTDOWN	Just before the server starts the shutdown of an instance. This lets the cartridge shutdown completely. For abnormal instance shutdown, this trigger might not fire.	None allowed	No database operations allowed in the trigger. Return status ignored.	Starts a separate transaction and commits it after firing the triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name
DB_ROLE_CHANGE	When the database is opened for the first time after a role change.	None allowed	Return status ignored.	Starts a separate transaction and commits it after firing the triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name
SERVERERROR	When the error eno occurs. If no condition is given, then this trigger fires whenever an error occurs. The trigger does not fire on ORA-1034, ORA-1403, ORA-1422, ORA-1423, and ORA-4030 because they are not true errors or are too serious to continue processing. It also fails to fire on ORA-18 and ORA-20 because a process is not available to connect to the database to record the error.	ERRNO = eno	Depends on the error. Return status ignored.	Starts a separate transaction and commits it after firing the triggers.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror space_error_info

Client Events

Client events are the events related to user logon/logoff, DML, and DDL operations. For example:

```
CREATE OR REPLACE TRIGGER On_Logon
  AFTER LOGON
  ON The_user.Schema
BEGIN
  Do_Something;
END;
```

The LOGON and LOGOFF events allow simple conditions on UID and USER. All other events allow simple conditions on the type and name of the object, as well as functions like UID and USER.

The LOGON event starts a separate transaction and commits it after firing the triggers. All other events fire the triggers in the existing user transaction.

The LOGON and LOGOFF events can operate on any objects. For all other events, the corresponding trigger cannot perform any DDL operations, such as DROP and ALTER, on the object that caused the event to be generated.

The DDL allowed inside these triggers is altering, creating, or dropping a table, creating a trigger, and compile operations.

If an event trigger becomes the target of a DDL operation (such as CREATE TRIGGER), it cannot fire later during the same transaction

Table 9–5 Client Events

Event	When Trigger Fires	Attribute Functions
BEFORE ALTER AFTER ALTER	When a catalog object is altered.	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 a catalog object is dropped.	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 an analyze statement is issued	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 an associate statistics statement is issued	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 AUDIT AFTER AUDIT BEFORE NOAUDIT AFTER NOAUDIT	When an audit or noaudit statement is issued	ora_sysevent ora_login_user ora_instance_num ora_database_name
BEFORE COMMENT AFTER COMMENT	When an object is commented	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 a catalog object is created.	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)

Table 9–5 (Cont.) Client Events

Event	When Trigger Fires	Attribute Functions
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.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner
BEFORE DISASSOCIATE STATISTICS AFTER DISASSOCIATE STATISTICS	When a disassociate statistics statement is issued	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 a grant statement is issued	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 the start of a user logoff	ora_sysevent ora_login_user ora_instance_num ora_database_name
AFTER LOGON	After a successful logon of a user.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_client_ip_address
BEFORE RENAME AFTER RENAME	When a rename statement is issued.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_owner ora_dict_obj_type

Table 9–5 (Cont.) Client Events

Event	When Trigger Fires	Attribute Functions
BEFORE REVOKE AFTER REVOKE	When a revoke statement is issued	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 a SQL statement is suspended because of an out-of-space condition. The trigger must correct the condition so the statement can be resumed.	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_server_error ora_is_servererror space_error_info
BEFORE TRUNCATE AFTER TRUNCATE	When an object is truncated	ora_sysevent ora_login_user ora_instance_num ora_database_name ora_dict_obj_name ora_dict_obj_type ora_dict_obj_owner

Using PL/SQL Packages

This chapter shows how to bundle related PL/SQL code and data into a package. The package might include a set of subprograms that forms an API, or a pool of type definitions and variable declarations. The package is compiled and stored in the database, where many applications can share its contents.

Topics:

- [What is a PL/SQL Package?](#)
- [What Goes in a PL/SQL Package?](#)
- [Advantages of PL/SQL Packages](#)
- [Understanding the PL/SQL Package Specification](#)
- [Referencing PL/SQL Package Contents](#)
- [Understanding the PL/SQL Package Body](#)
- [Examples of PL/SQL Package Features](#)
- [Private and Public Items in PL/SQL Packages](#)
- [How STANDARD Package Defines the PL/SQL Environment](#)
- [Overview of Product-Specific PL/SQL Packages](#)
- [Guidelines for Writing PL/SQL Packages](#)
- [Separating Cursor Specifications and Bodies with PL/SQL Packages](#)

What is a PL/SQL Package?

A package is a schema object that groups logically related PL/SQL types, variables, and subprograms. Packages usually have two parts, a specification (spec) and a body; sometimes the body is unnecessary. The specification is the interface to the package. It declares the types, variables, constants, exceptions, cursors, and subprograms that can be referenced from outside the package. The body defines the queries for the cursors and the code for the subprograms.

You can think of the spec as an interface and of the body as a *black box*. You can debug, enhance, or replace a package body without changing the package spec.

To create package specs, use the SQL statement `CREATE PACKAGE`. A `CREATE PACKAGE BODY` statement defines the package body. For information on the `CREATE PACKAGE SQL` statement, see *Oracle Database SQL Language Reference*. For information on the `CREATE PACKAGE BODY SQL` statement, see *Oracle Database SQL Language Reference*.

The spec holds public declarations, which are visible to stored subprograms and other code outside the package. You must declare subprograms at the end of the spec after all other items (except pragmas that name a specific function; such pragmas must follow the function spec).

The body holds implementation details and private declarations, which are hidden from code outside the package. Following the declarative part of the package body is the optional initialization part, which holds statements that initialize package variables and do any other one-time setup steps.

The `AUTHID` clause determines whether all the packaged subprograms execute 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 ["Using Invoker's Rights or Definer's Rights \(AUTHID Clause\)"](#) on page 8-17.

A call specification lets you map a package subprogram to a Java method or external C function. The call specification maps the Java or C name, parameter types, and return type to their SQL counterparts.

See Also:

- *Oracle Database Java Developer's Guide* to learn how to write Java call specifications
- *Oracle Database Advanced Application Developer's Guide* to learn how to write C call specifications
- *Oracle Database PL/SQL Packages and Types Reference* for information about PL/SQL packages provided with the Oracle database

What Goes in a PL/SQL Package?

The following is contained in a PL/SQL package:

- `Get` and `Set` methods for the package variables, if you want to avoid letting other subprograms read and write them directly.
- Cursor declarations with the text of SQL queries. Reusing exactly the same query text in multiple locations is faster than retyping the same query each time with slight differences. It is also easier to maintain if you need to change a query that is used in many places.
- Declarations for exceptions. Typically, you need to be able to reference these from different subprograms, so that you can handle exceptions within invoked subprograms.
- Declarations for subprograms that invoke each other. You do not need to worry about compilation order for packaged subprograms, making them more convenient than standalone stored subprograms when they invoke back and forth to each other.
- Declarations for overloaded subprograms. You can create multiple variations of a subprogram, using the same names but different sets of parameters.
- Variables that you want to remain available between subprogram calls in the same session. You can treat variables in a package like global variables.
- Type declarations for PL/SQL collection types. To pass a collection as a parameter between stored subprograms, you must declare the type in a package so that both the invoking and invoked subprogram can refer to it.

For additional information, see "[Package Declaration](#)" on page 13-107. For an examples of a PL/SQL packages, see [Example 1–14](#) on page 1-15 and [Example 10–3](#) on page 10-6. Only the declarations in the package spec are visible and accessible to applications. Implementation details in the package body are hidden and inaccessible. You can change the body (implementation) without having to recompile invoking programs.

Advantages of PL/SQL Packages

Packages have a long history in software engineering, offering important features for reliable, maintainable, reusable code, often in team development efforts for large systems.

Modularity

Packages let you encapsulate logically related types, items, and subprograms in a named PL/SQL module. Each package is easy to understand, and the interfaces between packages are simple, clear, and well defined. This aids application development.

Easier Application Design

When designing an application, all you need initially is the interface information in the package specs. You can code and compile a spec without its body. Then, stored subprograms that reference the package can be compiled as well. You need not define the package bodies fully until you are ready to complete the application.

Information Hiding

With packages, you can specify which types, items, and subprograms are public (visible and accessible) or private (hidden and inaccessible). For example, if a package contains four subprograms, three might be public and one private. The package hides the implementation of the private subprogram so that only the package (not your application) is affected if the implementation changes. This simplifies maintenance and enhancement. Also, by hiding implementation details from users, you protect the integrity of the package.

Added Functionality

Packaged public variables and cursors persist for the duration of a session. They can be shared by all subprograms that execute in the environment. They let you maintain data across transactions without storing it in the database.

Better Performance

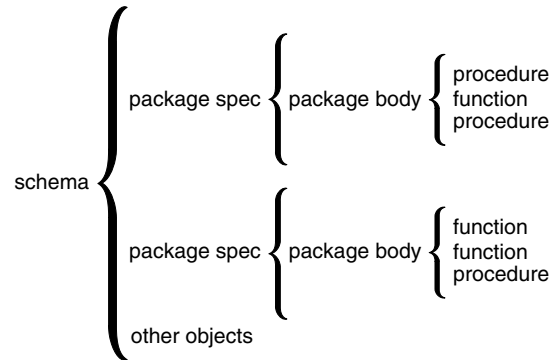
When you invoke a packaged subprogram for the first time, the whole package is loaded into memory. Later calls to related subprograms in the package require no disk I/O.

Packages stop cascading dependencies and avoid unnecessary recompiling. For example, if you change the body of a packaged function, Oracle does not recompile other subprograms that invoke the function; these subprograms only depend on the parameters and return value that are declared in the spec, so they are only recompiled if the spec changes.

Understanding the PL/SQL Package Specification

The package specification contains public declarations. The declared items are accessible from anywhere in the package and to any other subprograms in the same schema. [Figure 10–1](#) illustrates the scoping.

Figure 10–1 Package Scope



The spec lists the package resources available to applications. All the information your application needs to use the resources is in the spec. For example, the following declaration shows that the function named `factorial` takes one argument of type `INTEGER` and returns a value of type `INTEGER`:

```
FUNCTION factorial (n INTEGER) RETURN INTEGER; -- returns n!
```

That is all the information you need to invoke the function. You need not consider its underlying implementation (whether it is iterative or recursive for example).

If a spec declares only types, constants, variables, exceptions, and call specifications, the package body is unnecessary. Only subprograms and cursors have an underlying implementation. In [Example 10–1](#), the package needs no body because it declares types, exceptions, and variables, but no subprograms or cursors. Such packages let you define global variables, usable by stored subprograms and triggers, that persist throughout a session.

Example 10–1 A Simple Package Specification Without a Body

```
CREATE PACKAGE trans_data AS -- bodiless package
  TYPE TimeRec IS RECORD (
    minutes SMALLINT,
    hours   SMALLINT);
  TYPE TransRec IS RECORD (
    category VARCHAR2(10),
    account  INT,
    amount   REAL,
    time_of  TimeRec);
  minimum_balance  CONSTANT REAL := 10.00;
  number_processed INT;
  insufficient_funds EXCEPTION;
END trans_data;
/
```

Referencing PL/SQL Package Contents

To reference the types, items, subprograms, and call specifications declared within a package spec, use dot notation:

```
package_name.type_name
package_name.item_name
package_name.subprogram_name
package_name.call_spec_name
```

You can reference package contents from database triggers, stored subprograms, 3GL application programs, and various Oracle tools. For example, you can invoke package subprograms as shown in [Example 1–15](#) on page 1-16 or [Example 10–3](#) on page 10-6.

The following example invokes the `hire_employee` procedure from an anonymous block in a Pro*C program. The actual parameters `emp_id`, `emp_lname`, and `emp_fname` are host variables.

```
EXEC SQL EXECUTE
  BEGIN
    emp_actions.hire_employee(:emp_id,:emp_lname,:emp_fname, ...);
```

Restrictions

You cannot reference remote packaged variables, either directly or indirectly. For example, you cannot invoke the a subprogram through a database link if the subprogram refers to a packaged variable.

Inside a package, you cannot reference host variables.

Understanding the PL/SQL Package Body

The package body contains the implementation of every cursor and subprogram declared in the package spec. Subprograms defined in a package body are accessible outside the package only if their specs also appear in the package spec. If a subprogram spec is not included in the package spec, that subprogram can only be invoked by other subprograms in the same package. A package body must be in the same schema as the package spec.

To match subprogram specs and bodies, PL/SQL does a token-by-token comparison of their headers. Except for white space, the headers must match word for word. Otherwise, PL/SQL raises an exception, as [Example 10–2](#) shows.

Example 10–2 Matching Package Specifications and Bodies

```
CREATE PACKAGE emp_bonus AS
  PROCEDURE calc_bonus (date_hired employees.hire_date%TYPE);
END emp_bonus;
/
CREATE PACKAGE BODY emp_bonus AS
-- the following parameter declaration raises an exception
-- because 'DATE' does not match employees.hire_date%TYPE
-- PROCEDURE calc_bonus (date_hired DATE) IS
-- the following is correct because there is an exact match
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;
```

/

The package body can also contain private declarations, which define types and items necessary for the internal workings of the package. The scope of these declarations is local to the package body. Therefore, the declared types and items are inaccessible except from within the package body. Unlike a package spec, the declarative part of a package body can contain subprogram bodies.

Following the declarative part of a package body is the optional initialization part, which typically holds statements that initialize some of the variables previously declared in the package.

The initialization part of a package plays a minor role because, unlike subprograms, a package cannot be invoked or passed parameters. As a result, the initialization part of a package is run only once, the first time you reference the package.

Remember, if a package specification declares only types, constants, variables, exceptions, and call specifications, the package body is unnecessary. However, the body can still be used to initialize items declared in the package spec.

Examples of PL/SQL Package Features

Consider the following package, named `emp_admin`. The package specification declares the following types, items, and subprograms:

- Type `EmpRecTyp`
- Cursor `desc_salary`
- Exception `invalid_salary`
- Functions `hire_employee` and `nth_highest_salary`
- Procedures `fire_employee` and `raise_salary`

After writing the package, you can develop applications that reference its types, invoke its subprograms, use its cursor, and raise its exception. When you create the package, it is stored in an Oracle database for use by any application that has execute privilege on the package.

Example 10-3 *Creating the emp_admin Package*

```
-- create the audit table to track changes
CREATE TABLE emp_audit(date_of_action DATE, user_id VARCHAR2(20),
                       package_name VARCHAR2(30));

CREATE OR REPLACE PACKAGE emp_admin AS
-- Declare externally visible types, cursor, exception
  TYPE EmpRecTyp IS RECORD (emp_id NUMBER, sal NUMBER);
  CURSOR desc_salary RETURN EmpRecTyp;
  invalid_salary EXCEPTION;
-- Declare externally callable subprograms
  FUNCTION hire_employee (last_name VARCHAR2,
                          first_name VARCHAR2,
                          email VARCHAR2,
                          phone_number VARCHAR2,
                          job_id VARCHAR2,
                          salary NUMBER,
                          commission_pct NUMBER,
                          manager_id NUMBER,
                          department_id NUMBER)
    RETURN NUMBER;
```



```

PROCEDURE fire_employee
  (emp_id NUMBER); -- overloaded subprogram
PROCEDURE fire_employee
  (emp_email VARCHAR2); -- overloaded subprogram
PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
FUNCTION nth_highest_salary (n NUMBER) RETURN EmpRecTyp;
END emp_admin;
/
CREATE OR REPLACE PACKAGE BODY emp_admin AS
  number_hired NUMBER; -- visible only in this package
-- Fully define cursor specified in package
CURSOR desc_salary RETURN EmpRecTyp IS
  SELECT employee_id, salary
  FROM employees
  ORDER BY salary DESC;
-- Fully define subprograms specified in package
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 VALUES (new_emp_id,
    last_name,
    first_name,
    email,
    phone_number,
    SYSDATE,
    job_id,
    salary,
    commission_pct,
    manager_id,
    department_id);
  number_hired := number_hired + 1;
  DBMS_OUTPUT.PUT_LINE('The number of employees hired is '
    || TO_CHAR(number_hired) );
  RETURN new_emp_id;
END hire_employee;
PROCEDURE fire_employee (emp_id NUMBER) IS
BEGIN
  DELETE FROM employees WHERE employee_id = emp_id;
END fire_employee;
PROCEDURE fire_employee (emp_email VARCHAR2) IS
BEGIN
  DELETE FROM employees WHERE email = emp_email;
END fire_employee;
-- Define local function, available only inside package
FUNCTION sal_ok (jobid VARCHAR2, sal NUMBER) RETURN BOOLEAN IS
  min_sal NUMBER;
  max_sal NUMBER;
BEGIN
  SELECT MIN(salary), MAX(salary)
  INTO min_sal, max_sal
  FROM employees

```

```

        WHERE job_id = jobid;
        RETURN (sal >= min_sal) AND (sal <= max_sal);
    END sal_ok;
    PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS
        sal NUMBER(8,2);
        jobid VARCHAR2(10);
    BEGIN
        SELECT job_id, salary INTO jobid, sal
            FROM employees
            WHERE employee_id = emp_id;
        IF sal_ok(jobid, sal + amount) THEN
            UPDATE employees SET salary =
                salary + amount WHERE employee_id = emp_id;
        ELSE
            RAISE invalid_salary;
        END IF;
    EXCEPTION -- exception-handling part starts here
        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 starts here
        INSERT INTO emp_audit VALUES (SYSDATE, USER, 'EMP_ADMIN');
        number_hired := 0;
    END emp_admin;
/
-- invoking the package procedures
DECLARE
    new_emp_id NUMBER(6);
    BEGIN
        new_emp_id := emp_admin.hire_employee ('Belden',
            'Enrique',
            'EBELDEN',
            '555.111.2222',
            'ST_CLERK',
            2500,
            .1,
            101,
            110);

        DBMS_OUTPUT.PUT_LINE
            ('The new 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;
```

/

Remember, the initialization part of a package is run just once, the first time you reference the package. In the last example, only one row is inserted into the database table `emp_audit`, and the variable `number_hired` is initialized only once.

Every time the procedure `hire_employee` is invoked, the variable `number_hired` is updated. However, the count kept by `number_hired` is session specific. That is, the count reflects the number of new employees processed by one user, not the number processed by all users.

PL/SQL allows two or more packaged subprograms to have the same name. This option is useful when you want a subprogram to accept similar sets of parameters that have different datatypes. For example, the `emp_admin` package in [Example 10-3](#) defines two procedures named `fire_employee`. The first procedure accepts a number, while the second procedure accepts string. Each procedure handles the data appropriately. For the rules that apply to overloaded subprograms, see "[Overloading PL/SQL Subprogram Names](#)" on page 8-12.

Private and Public Items in PL/SQL Packages

In the package `emp_admin`, the package body declares a variable named `number_hired`, which is initialized to zero. Items declared in the body are restricted to use within the package. PL/SQL code outside the package cannot reference the variable `number_hired`. Such items are called private.

Items declared in the spec of `emp_admin`, such as the exception `invalid_salary`, are visible outside the package. Any PL/SQL code can reference the exception `invalid_salary`. Such items are called public.

To maintain items throughout a session or across transactions, place them in the declarative part of the package body. For example, the value of `number_hired` is kept between calls to `hire_employee` within the same session. The value is lost when the session ends.

To make the items public, place them in the package specification. For example, `emp_rec` declared in the spec of the package is available for general use.

How STANDARD Package Defines the PL/SQL Environment

A package named `STANDARD` defines the PL/SQL environment. The package spec globally declares types, exceptions, and subprograms, which are available automatically to PL/SQL programs. For example, package `STANDARD` declares function `ABS`, which returns the absolute value of its argument, as follows:

```
FUNCTION ABS (n NUMBER) RETURN NUMBER;
```

The contents of package `STANDARD` are directly visible to applications. You do not need to qualify references to its contents by prefixing the package name. For example, you might invoke `ABS` from a database trigger, stored subprogram, Oracle tool, or 3GL application, as follows:

```
abs_diff := ABS(x - y);
```

If you declare your own version of `ABS`, your local declaration overrides the global declaration. You can still invoke the built-in function by specifying its full name:

```
abs_diff := STANDARD.ABS(x - y);
```

Most built-in functions are overloaded. For example, package `STANDARD` contains the following declarations:

```
FUNCTION TO_CHAR (right DATE) RETURN VARCHAR2;  
FUNCTION TO_CHAR (left NUMBER) RETURN VARCHAR2;  
FUNCTION TO_CHAR (left DATE, right VARCHAR2) RETURN VARCHAR2;  
FUNCTION TO_CHAR (left NUMBER, right VARCHAR2) RETURN VARCHAR2;
```

PL/SQL resolves a call to `TO_CHAR` by matching the number and datatypes of the formal and actual parameters.

Overview of Product-Specific PL/SQL Packages

Oracle Database and various Oracle tools are supplied with product-specific packages that define application programming interfaces (APIs) that you can invoke from PL/SQL, SQL, Java, and other programming environments. This section briefly describes the following widely used product-specific packages:

- [DBMS_ALERT Package](#)
- [DBMS_OUTPUT Package](#)
- [DBMS_PIPE Package](#)
- [DBMS_CONNECTION_POOL Package](#)
- [HTF and HTP Packages](#)
- [UTL_FILE Package](#)
- [UTL_HTTP Package](#)
- [UTL_SMTP Package](#)

For more information on these and other product-specific packages, see *Oracle Database PL/SQL Packages and Types Reference*.

DBMS_ALERT Package

`DBMS_ALERT` package lets you use database triggers to alert an application when specific database values change. The alerts are transaction based and asynchronous (that is, they operate independently of any timing mechanism). For example, a company might use this package to update the value of its investment portfolio as new stock and bond quotes arrive.

DBMS_OUTPUT Package

`DBMS_OUTPUT` package enables you to display output from PL/SQL blocks, subprograms, packages, and triggers. The package is especially useful for displaying PL/SQL debugging information. The procedure `PUT_LINE` outputs information to a buffer that can be read by another trigger, subprogram, or package. You display the information by invoking the procedure `GET_LINE` or by setting `SERVEROUTPUT ON` in SQL*Plus. [Example 10-4](#) shows how to display output from a PL/SQL block.

Example 10-4 Using PUT_LINE in the DBMS_OUTPUT Package

```
REM set server output to ON to display output from DBMS_OUTPUT  
SET SERVEROUTPUT ON  
BEGIN  
    DBMS_OUTPUT.PUT_LINE  
        ('These are the tables that ' || USER || ' owns:');
```

```

FOR item IN (SELECT table_name FROM user_tables)
LOOP
    DBMS_OUTPUT.PUT_LINE(item.table_name);
END LOOP;
END;
/

```

DBMS_PIPE Package

DBMS_PIPE package allows different sessions to communicate over named pipes. (A *pipe* is an area of memory used by one process to pass information to another.) You can use the procedures `PACK_MESSAGE` and `SEND_MESSAGE` to pack a message into a pipe, then send it to another session in the same instance or to a waiting application such as a Linux or UNIX program.

At the other end of the pipe, you can use the procedures `RECEIVE_MESSAGE` and `UNPACK_MESSAGE` to receive and unpack (read) the message. Named pipes are useful in many ways. For example, you can write a C program to collect data, then send it through pipes to stored subprograms in an Oracle database.

DBMS_CONNECTION_POOL Package

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

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for a detailed description of the DBMS_CONNECTION_POOL package
- *Oracle Database Administrator's Guide* for information about managing the Database Resident Connection Pool

HTF and HTP Packages

HTF and HTP packages enable your PL/SQL programs to generate HTML tags.

UTL_FILE Package

UTL_FILE package lets PL/SQL programs read and write operating system text files. It provides a restricted version of standard operating system stream file I/O, including open, put, get, and close operations.

When you want to read or write a text file, you invoke the function `FOPEN`, which returns a file handle for use in subsequent subprogram calls. For example, the procedure `PUT_LINE` writes a text string and line terminator to an open file, and the procedure `GET_LINE` reads a line of text from an open file into an output buffer.

UTL_HTTP Package

UTL_HTTP package enables your PL/SQL programs to make hypertext transfer protocol (HTTP) callouts. It can retrieve data from the Internet or invoke Oracle Web Server cartridges. The package has two entry points, each of which accepts a URL (uniform resource locator) string, contacts the specified site, and returns the requested data, which is usually in hypertext markup language (HTML) format.

UTL_SMTP Package

UTL_SMTP package enables your PL/SQL programs to send electronic mails (emails) over Simple Mail Transfer Protocol (SMTP). The package provides interfaces to the SMTP commands for an email client to dispatch emails to a SMTP server.

Guidelines for Writing PL/SQL Packages

When writing packages, keep them general so they can be reused in future applications. Become familiar with the Oracle-supplied packages, and avoid writing packages that duplicate features already provided by Oracle.

Design and define package specs before the package bodies. Place in a spec only those things that must be visible to invoking programs. That way, other developers cannot build unsafe dependencies on your implementation details.

To reduce the need for recompiling when code is changed, place as few items as possible in a package spec. Changes to a package body do not require recompiling invoking subprograms. Changes to a package spec require Oracle to recompile every stored subprogram that references the package.

Separating Cursor Specifications and Bodies with PL/SQL Packages

You can separate a cursor specification (spec for short) from its body for placement in a package. That way, you can change the cursor body without having to change the cursor spec. For information on the cursor syntax, see "[Cursor Declaration](#)" on page 13-41.

In [Example 10-5](#), you use the %ROWTYPE attribute to provide a record type that represents a row in the database table employees:

Example 10-5 Separating Cursor Specifications with Packages

```
CREATE PACKAGE emp_stuff AS
  -- Declare cursor spec
  CURSOR c1 RETURN employees%ROWTYPE;
END emp_stuff;
/
CREATE PACKAGE BODY emp_stuff AS
  CURSOR c1 RETURN employees%ROWTYPE IS
    -- Define cursor body
    SELECT * FROM employees WHERE salary > 2500;
END emp_stuff;
/
```

The cursor spec has no SELECT statement because the RETURN clause specifies the datatype of the return value. However, the cursor body must have a SELECT statement and the same RETURN clause as the cursor spec. Also, the number and datatypes of items in the SELECT list and the RETURN clause must match.

Packaged cursors increase flexibility. For example, you can change the cursor body in the last example, without having to change the cursor spec.

From a PL/SQL block or subprogram, you use dot notation to reference a packaged cursor, as the following example shows:

```
DECLARE
  emp_rec employees%ROWTYPE;
BEGIN
  OPEN emp_stuff.c1;
```

```
    LOOP
      FETCH emp_stuff.c1 INTO emp_rec;
    -- do processing here ...
      EXIT WHEN emp_stuff.c1%NOTFOUND;
    END LOOP;
    CLOSE emp_stuff.c1;
  END;
/
```

The scope of a packaged cursor is not limited to a PL/SQL block. When you open a packaged cursor, it remains open until you close it or you disconnect from the session.

Handling PL/SQL Errors

Run-time errors arise from design faults, coding mistakes, hardware failures, and many other sources. Although you cannot anticipate all possible errors, you can plan to handle certain kinds of errors meaningful to your PL/SQL program.

With many programming languages, unless you disable error checking, a run-time error such as stack overflow or division by zero stops normal processing and returns control to the operating system. With PL/SQL, a mechanism called exception handling lets you bulletproof your program so that it can continue operating in the presence of errors.

Topics:

- [Overview of PL/SQL Run-Time Error Handling](#)
- [Guidelines for Avoiding and Handling PL/SQL Errors and Exceptions](#)
- [Advantages of PL/SQL Exceptions](#)
- [Summary of Predefined PL/SQL Exceptions](#)
- [Defining Your Own PL/SQL Exceptions](#)
- [How PL/SQL Exceptions Are Raised](#)
- [How PL/SQL Exceptions Propagate](#)
- [Reraising a PL/SQL Exception](#)
- [Handling Raised PL/SQL Exceptions](#)
- [Overview of PL/SQL Compile-Time Warnings](#)

Overview of PL/SQL Run-Time Error Handling

In PL/SQL, an error condition is called an exception. Exceptions can be internally defined (by the run-time system) or user defined. Examples of internally defined exceptions include `division by zero` and `out of memory`. Some common internal exceptions have predefined names, such as `ZERO_DIVIDE` and `STORAGE_ERROR`. The other internal exceptions can be given names.

You can define exceptions of your own in the declarative part of any PL/SQL block, subprogram, or package. For example, you might define an exception named `insufficient_funds` to flag overdrawn bank accounts. Unlike internal exceptions, user-defined exceptions must be given names.

When an error occurs, an exception is raised. That is, normal execution stops and control transfers to the exception-handling part of your PL/SQL block or subprogram. Internal exceptions are raised implicitly (automatically) by the run-time system.

User-defined exceptions must be raised explicitly by RAISE statements, which can also raise predefined exceptions.

To handle raised exceptions, you write separate routines called exception handlers. After an exception handler runs, the current block stops executing and the enclosing block resumes with the next statement. If there is no enclosing block, control returns to the host environment. For information on managing errors when using BULK COLLECT, see ["Handling FORALL Exceptions \(%BULK_EXCEPTIONS Attribute\)"](#) on page 12-17.

Example 11-1 calculates a price-to-earnings ratio for a company. If the company has zero earnings, the division operation raises the predefined exception ZERO_DIVIDE, the execution of the block is interrupted, and control is transferred to the exception handlers. The optional OTHERS handler catches all exceptions that the block does not name specifically.

Example 11-1 Run-Time Error Handling

```
DECLARE
    stock_price NUMBER := 9.73;
    net_earnings NUMBER := 0;
    pe_ratio NUMBER;
BEGIN
    -- Calculation might cause division-by-zero error.
    pe_ratio := stock_price / net_earnings;
    DBMS_OUTPUT.PUT_LINE('Price/earnings ratio = ' || pe_ratio);
EXCEPTION -- exception handlers begin
    -- Only one of the WHEN blocks is executed.
    WHEN ZERO_DIVIDE THEN -- handles 'division by zero' error
        DBMS_OUTPUT.PUT_LINE('Company must have had zero earnings. ');
        pe_ratio := NULL;
    WHEN OTHERS THEN -- handles all other errors
        DBMS_OUTPUT.PUT_LINE('Some other kind of error occurred. ');
        pe_ratio := NULL;
END; -- exception handlers and block end here
/
```

The last example illustrates exception handling. With better error checking, you can avoid the exception entirely, by substituting a null for the answer if the denominator was zero, as shown in the following example.

```
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;
/
```

Guidelines for Avoiding and Handling PL/SQL Errors and Exceptions

Because reliability is crucial for database programs, use both error checking and exception handling to ensure your program can handle all possibilities:

- Add exception handlers whenever errors can occur.

Errors are especially likely during arithmetic calculations, string manipulation, and database operations. Errors can also occur at other times, for example if a hardware failure with disk storage or memory causes a problem that has nothing to do with your code; but your code still needs to take corrective action.

- Add error-checking code whenever bad input data can cause an error.

Expect that at some time, your code will be passed incorrect or null parameters, that your queries will return no rows or more rows than you expect.

Test your code with different combinations of bad data to see what potential errors arise.

- Make your programs robust enough to work even if the database is not in the state you expect.

For example, perhaps a table you query will have columns added or deleted, or their types changed. You can avoid such problems by declaring individual variables with %TYPE qualifiers, and declaring records to hold query results with %ROWTYPE qualifiers.

- Handle named exceptions whenever possible, instead of using WHEN OTHERS in exception handlers.

Learn the names and causes of the predefined exceptions. If your database operations might cause particular ORA- errors, associate names with these errors so you can write handlers for them. (You will learn how to do that later in this chapter.)

- Write out debugging information in your exception handlers.

You might store such information in a separate table. If so, do it by invoking a subprogram declared with the PRAGMA AUTONOMOUS_TRANSACTION, so that you can commit your debugging information, even if you roll back the work that the main subprogram was doing.

- Carefully consider whether each exception handler should commit the transaction, roll it back, or let it continue.

No matter how severe the error is, you want to leave the database in a consistent state and avoid storing any bad data.

Advantages of PL/SQL Exceptions

Using exceptions for error handling has several advantages. With exceptions, you can reliably handle potential errors from many statements with a single exception handler:

Example 11–2 *Managing Multiple Errors with a Single Exception Handler*

```
DECLARE
    emp_column      VARCHAR2(30) := 'last_name';
    table_name      VARCHAR2(30) := 'emp';
    temp_var        VARCHAR2(30);
BEGIN
    temp_var := emp_column;
    SELECT COLUMN_NAME INTO temp_var FROM USER_TAB_COLS
        WHERE TABLE_NAME = 'EMPLOYEES'
        AND COLUMN_NAME = UPPER(emp_column);
    -- processing here
    temp_var := table_name;
    SELECT OBJECT_NAME INTO temp_var FROM USER_OBJECTS
        WHERE OBJECT_NAME = UPPER(table_name)
```

```

        AND OBJECT_TYPE = 'TABLE';
    -- processing here
EXCEPTION
    -- Catches all 'no data found' errors
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE
            ('No Data found for SELECT on ' || temp_var);
END;
/

```

Instead of checking for an error at every point it might occur, just add an exception handler to your PL/SQL block. If the exception is ever raised in that block (or any sub-block), you can be sure it will be handled.

Sometimes the error is not immediately obvious, and cannot be detected until later when you perform calculations using bad data. Again, a single exception handler can trap all division-by-zero errors, bad array subscripts, and so on.

If you need to check for errors at a specific spot, you can enclose a single statement or a group of statements inside its own BEGIN-END block with its own exception handler. You can make the checking as general or as precise as you like.

Isolating error-handling routines makes the rest of the program easier to read and understand.

Summary of Predefined PL/SQL Exceptions

An internal exception is raised automatically if your PL/SQL program violates an Oracle rule or exceeds a system-dependent limit. PL/SQL predefines some common Oracle errors as exceptions. For example, PL/SQL raises the predefined exception `NO_DATA_FOUND` if a `SELECT INTO` statement returns no rows.

You can use the pragma `EXCEPTION_INIT` to associate exception names with other Oracle error codes that you can anticipate. To handle unexpected Oracle errors, you can use the `OTHERS` handler. Within this handler, you can invoke the functions `SQLCODE` and `SQLERRM` to return the Oracle error code and message text. Once you know the error code, you can use it with pragma `EXCEPTION_INIT` and write a handler specifically for that error.

PL/SQL declares predefined exceptions globally in package `STANDARD`. You need not declare them yourself. You can write handlers for predefined exceptions using the names in the following table:

Exception	ORA Error	SQLCODE	Raise When ...
<code>ACCESS_INTO_NULL</code>	06530	-6530	A program attempts to assign values to the attributes of an uninitialized object
<code>CASE_NOT_FOUND</code>	06592	-6592	None of the choices in the <code>WHEN</code> clauses of a <code>CASE</code> statement is selected, and there is no <code>ELSE</code> clause.
<code>COLLECTION_IS_NULL</code>	06531	-6531	A program attempts to apply collection methods other than <code>EXISTS</code> to an uninitialized nested table or varray, or the program attempts to assign values to the elements of an uninitialized nested table or varray.
<code>CURSOR_ALREADY_OPEN</code>	06511	-6511	A program attempts to open an already open cursor. A cursor must be closed before it can be reopened. A cursor <code>FOR</code> loop automatically opens the cursor to which it refers, so your program cannot open that cursor inside the loop.

Exception	ORA Error	SQLCODE	Raise When ...
DUP_VAL_ON_INDEX	00001	-1	A program attempts to store duplicate values in a column that is constrained by a unique index.
INVALID_CURSOR	01001	-1001	A program attempts a cursor operation that is not allowed, such as closing an unopened cursor.
INVALID_NUMBER	01722	-1722	In a SQL statement, the conversion of a character string into a number fails because the string does not represent a valid number. (In procedural statements, <code>VALUE_ERROR</code> is raised.) This exception is also raised when the <code>LIMIT</code> -clause expression in a bulk <code>FETCH</code> statement does not evaluate to a positive number.
LOGIN_DENIED	01017	-1017	A program attempts to log on to Oracle with an invalid username or password.
NO_DATA_FOUND	01403	+100	A <code>SELECT INTO</code> statement returns no rows, or your program references a deleted element in a nested table or an uninitialized element in an index-by table. Because this exception is used internally by some SQL functions to signal completion, you must not rely on this exception being propagated if you raise it within a function that is invoked as part of a query.
NOT_LOGGED_ON	01012	-1012	A program issues a database call without being connected to Oracle.
PROGRAM_ERROR	06501	-6501	PL/SQL has an internal problem.
ROWTYPE_MISMATCH	06504	-6504	The host cursor variable and PL/SQL cursor variable involved in an assignment have incompatible return types. When an open host cursor variable is passed to a stored subprogram, the return types of the actual and formal parameters must be compatible.
SELF_IS_NULL	30625	-30625	A program attempts to invoke a <code>MEMBER</code> method, but the instance of the object type was not initialized. The built-in parameter <code>SELF</code> points to the object, and is always the first parameter passed to a <code>MEMBER</code> method.
STORAGE_ERROR	06500	-6500	PL/SQL ran out of memory or memory was corrupted.
SUBSCRIPT_BEYOND_COUNT	06533	-6533	A program references a nested table or varray element using an index number larger than the number of elements in the collection.
SUBSCRIPT_OUTSIDE_LIMIT	06532	-6532	A program references a nested table or varray element using an index number (-1 for example) that is outside the legal range.
SYS_INVALID_ROWID	01410	-1410	The conversion of a character string into a universal rowid fails because the character string does not represent a valid rowid.
TIMEOUT_ON_RESOURCE	00051	-51	A time out occurs while Oracle is waiting for a resource.
TOO_MANY_ROWS	01422	-1422	A <code>SELECT INTO</code> statement returns more than one row.
VALUE_ERROR	06502	-6502	An arithmetic, conversion, truncation, or size-constraint error occurs. For example, when your program selects a column value into a character variable, if the value is longer than the declared length of the variable, PL/SQL stops the assignment and raises <code>VALUE_ERROR</code> . In procedural statements, <code>VALUE_ERROR</code> is raised if the conversion of a character string into a number fails. (In SQL statements, <code>INVALID_NUMBER</code> is raised.)

Exception	ORA Error	SQLCODE	Raise When ...
ZERO_DIVIDE	01476	-1476	A program attempts to divide a number by zero.

Defining Your Own PL/SQL Exceptions

PL/SQL lets you define exceptions of your own. Unlike predefined exceptions, user-defined exceptions must be declared and must be raised explicitly by `RAISE` statements.

Topics:

- [Declaring PL/SQL Exceptions](#)
- [Scope Rules for PL/SQL Exceptions](#)
- [Associating a PL/SQL Exception with a Number \(Pragma `EXCEPTION_INIT`\)](#)
- [Defining Your Own Error Messages \(`RAISE_APPLICATION_ERROR` Procedure\)](#)
- [Redeclaring Predefined Exceptions](#)

Declaring PL/SQL Exceptions

Exceptions can be declared only in the declarative part of a PL/SQL block, subprogram, or package. You declare an exception by introducing its name, followed by the keyword `EXCEPTION`. In the following example, you declare an exception named `past_due`:

```
DECLARE
    past_due EXCEPTION;
```

Exception and variable declarations are similar. But remember, an exception is an error condition, not a data item. Unlike variables, exceptions cannot appear in assignment statements or SQL statements. However, the same scope rules apply to variables and exceptions.

Scope Rules for PL/SQL Exceptions

You cannot declare an exception twice in the same block. You can, however, declare the same exception in two different blocks.

Exceptions declared in a block are considered local to that block and global to all its sub-blocks. Because a block can reference only local or global exceptions, enclosing blocks cannot reference exceptions declared in a sub-block.

If you redeclare a global exception in a sub-block, the local declaration prevails. The sub-block cannot reference the global exception, unless the exception is declared in a labeled block and you qualify its name with the block label `block_label.exception_name`.

[Example 11-3](#) illustrates the scope rules:

Example 11-3 Scope of PL/SQL Exceptions

```
DECLARE
    past_due EXCEPTION;
    acct_num NUMBER;
BEGIN
    DECLARE ----- sub-block begins
        past_due EXCEPTION; -- this declaration prevails
```

```

        acct_num NUMBER;
        due_date DATE := SYSDATE - 1;
        todays_date DATE := SYSDATE;
    BEGIN
        IF due_date < todays_date THEN
            RAISE past_due; -- this is not handled
        END IF;
    END; ----- sub-block ends
EXCEPTION
-- Does not handle raised exception
WHEN past_due THEN
    DBMS_OUTPUT.PUT_LINE
        ('Handling PAST_DUE exception.');
```

```

WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE
        ('Could not recognize PAST_DUE_EXCEPTION in this scope.');
```

```

END;
/
```

The enclosing block does not handle the raised exception because the declaration of `past_due` in the sub-block prevails. Though they share the same name, the two `past_due` exceptions are different, just as the two `acct_num` variables share the same name but are different variables. Thus, the `RAISE` statement and the `WHEN` clause refer to different exceptions. To have the enclosing block handle the raised exception, you must remove its declaration from the sub-block or define an `OTHERS` handler.

Associating a PL/SQL Exception with a Number (Pragma `EXCEPTION_INIT`)

To handle error conditions (typically `ORA-` messages) that have no predefined name, you must use the `OTHERS` handler or the pragma `EXCEPTION_INIT`. A pragma is a compiler directive that is processed at compile time, not at run time.

In PL/SQL, the pragma `EXCEPTION_INIT` tells the compiler to associate an exception name with an Oracle error number. That lets you refer to any internal exception by name and to write a specific handler for it. When you see an error stack, or sequence of error messages, the one on top is the one that you can trap and handle.

You code the pragma `EXCEPTION_INIT` in the declarative part of a PL/SQL block, subprogram, or package using the following syntax:

```
PRAGMA EXCEPTION_INIT(exception_name, -Oracle_error_number);
```

where *exception_name* is the name of a previously declared exception and the number is a negative value corresponding to an `ORA-` error number. The pragma must appear somewhere after the exception declaration in the same declarative section, as shown in [Example 11-4](#).

Example 11-4 Using `PRAGMA EXCEPTION_INIT`

```

DECLARE
    deadlock_detected EXCEPTION;
    PRAGMA EXCEPTION_INIT(deadlock_detected, -60);
BEGIN
    NULL; -- Some operation that causes an ORA-00060 error
EXCEPTION
    WHEN deadlock_detected THEN
        NULL; -- handle the error
END;
/
```

Defining Your Own Error Messages (RAISE_APPLICATION_ERROR Procedure)

The `RAISE_APPLICATION_ERROR` procedure lets you issue user-defined ORA- error messages from stored subprograms. That way, you can report errors to your application and avoid returning unhandled exceptions.

To invoke `RAISE_APPLICATION_ERROR`, use the following syntax:

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

where *error_number* is a negative integer in the range -20000..-20999 and *message* is a character string up to 2048 bytes long. If the optional third parameter is `TRUE`, the error is placed on the stack of previous errors. If the parameter is `FALSE` (the default), the error replaces all previous errors. `RAISE_APPLICATION_ERROR` is part of package `DBMS_STANDARD`, and as with package `STANDARD`, you do not need to qualify references to it.

An application can invoke `raise_application_error` only from an executing stored subprogram (or method). When invoked, `raise_application_error` ends the subprogram and returns a user-defined error number and message to the application. The error number and message can be trapped like any Oracle error.

In [Example 11-5](#), you invoke `raise_application_error` if an error condition of your choosing happens (in this case, if the current schema owns less than 1000 tables):

Example 11-5 Raising an Application Error with `raise_application_error`

```
DECLARE
    num_tables NUMBER;
BEGIN
    SELECT COUNT(*) INTO num_tables FROM USER_TABLES;
    IF num_tables < 1000 THEN
        /* Issue your own error code (ORA-20101)
           with your own error message. You do not need to
           qualify raise_application_error with
           DBMS_STANDARD */
        raise_application_error
            (-20101, 'Expecting at least 1000 tables');
    ELSE
        -- Do rest of processing (for nonerror case)
        NULL;
    END IF;
END;
```

The invoking application gets a PL/SQL exception, which it can process using the error-reporting functions `SQLCODE` and `SQLERRM` in an `OTHERS` handler. Also, it can use the pragma `EXCEPTION_INIT` to map specific error numbers returned by `raise_application_error` to exceptions of its own, as the following Pro*C example shows:

```
EXEC SQL EXECUTE
    /* Execute embedded PL/SQL block using host
       variables v_emp_id and v_amount, which were
       assigned values in the host environment. */
DECLARE
    null_salary EXCEPTION;
    /* Map error number returned by raise_application_error
       to user-defined exception. */
    PRAGMA EXCEPTION_INIT(null_salary, -20101);
```



```

BEGIN
    raise_salary(:v_emp_id, :v_amount);
EXCEPTION
    WHEN null_salary THEN
        INSERT INTO emp_audit VALUES (:v_emp_id, ...);
END;
END-EXEC;

```

This technique allows the invoking application to handle error conditions in specific exception handlers.

Redeclaring Predefined Exceptions

Remember, PL/SQL declares predefined exceptions globally in package `STANDARD`, so you need not declare them yourself. Redeclaring predefined exceptions is error prone because your local declaration overrides the global declaration. For example, if you declare an exception named `invalid_number` and then PL/SQL raises the predefined exception `INVALID_NUMBER` internally, a handler written for `INVALID_NUMBER` will not catch the internal exception. In such cases, you must use dot notation to specify the predefined exception, as follows:

```

EXCEPTION
    WHEN invalid_number OR STANDARD.INVALID_NUMBER THEN
        -- handle the error
END;

```

How PL/SQL Exceptions Are Raised

Internal exceptions are raised implicitly by the run-time system, as are user-defined exceptions that you have associated with an Oracle error number using `EXCEPTION_INIT`. However, other user-defined exceptions must be raised explicitly by `RAISE` statements.

Raise an exception in a PL/SQL block or subprogram only when an error makes it undesirable or impossible to finish processing. You can place `RAISE` statements for a given exception anywhere within the scope of that exception. In [Example 11-6](#), you alert your PL/SQL block to a user-defined exception named `out_of_stock`.

Example 11-6 Using RAISE to Force a User-Defined Exception

```

DECLARE
    out_of_stock EXCEPTION;
    number_on_hand NUMBER := 0;
BEGIN
    IF number_on_hand < 1 THEN
        RAISE out_of_stock; -- raise an exception that we defined
    END IF;
EXCEPTION
    WHEN out_of_stock THEN
        -- handle the error
        DBMS_OUTPUT.PUT_LINE('Encountered out-of-stock error.');
```

END;
/

You can also raise a predefined exception explicitly. That way, an exception handler written for the predefined exception can process other errors, as [Example 11-7](#) shows:

Example 11–7 Using RAISE to Force a Pre-Defined Exception

```

DECLARE
    acct_type INTEGER := 7;
BEGIN
    IF acct_type NOT IN (1, 2, 3) THEN
        RAISE INVALID_NUMBER; -- raise predefined exception
    END IF;
EXCEPTION
    WHEN INVALID_NUMBER THEN
        DBMS_OUTPUT.PUT_LINE
            ('HANDLING INVALID INPUT BY ROLLING BACK. ');
        ROLLBACK;
END;
/

```

How PL/SQL Exceptions Propagate

When an exception is raised, if PL/SQL cannot find a handler for it in the current block or subprogram, the exception propagates. That is, the exception reproduces itself in successive enclosing blocks until a handler is found or there are no more blocks to search. If no handler is found, PL/SQL returns an unhandled exception error to the host environment.

Exceptions cannot propagate across remote subprogram calls done through database links. A PL/SQL block cannot catch an exception raised by a remote subprogram. For a workaround, see ["Defining Your Own Error Messages \(RAISE_APPLICATION_ERROR Procedure\)"](#) on page 11-8.

Figure 11–1, Figure 11–2, and Figure 11–3 illustrate the basic propagation rules.

Figure 11–1 Propagation Rules: Example 1

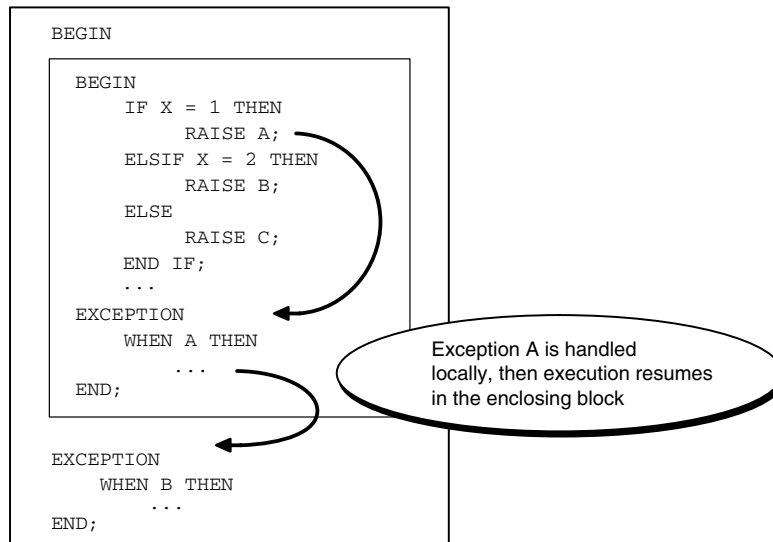
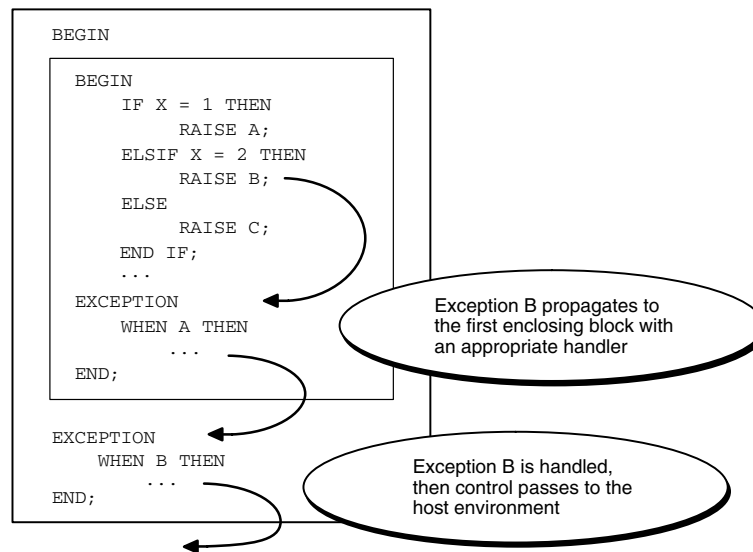
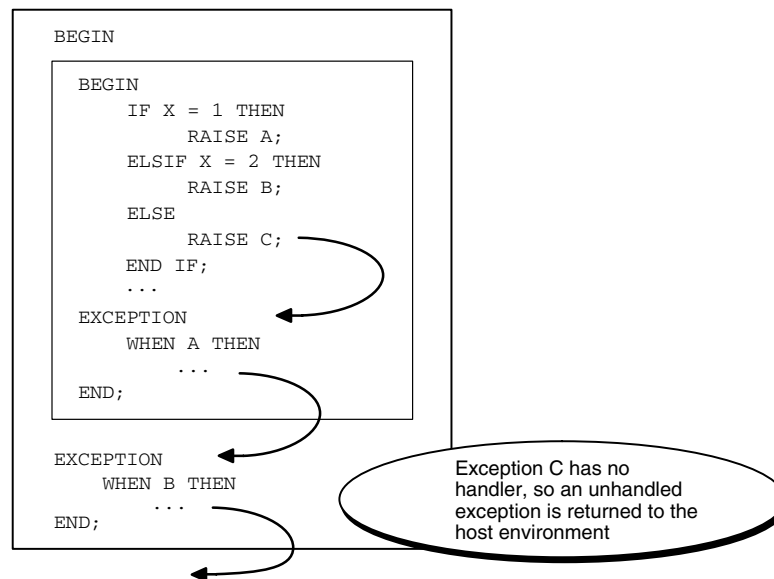


Figure 11–2 Propagation Rules: Example 2**Figure 11–3 Propagation Rules: Example 3**

An exception can propagate beyond its scope, that is, beyond the block in which it was declared, as shown in [Example 11–8](#).

Example 11–8 Scope of an Exception

```

BEGIN
  DECLARE ----- sub-block begins
    past_due EXCEPTION;
    due_date DATE := trunc(SYSDATE) - 1;
    todays_date DATE := trunc(SYSDATE);
  BEGIN
    IF due_date < todays_date THEN
      RAISE past_due;
    END IF;
  END; ----- sub-block ends
END;

```

```

EXCEPTION
  WHEN OTHERS THEN
    ROLLBACK;
END;
/

```

Because the block that declares the exception `past_due` has no handler for it, the exception propagates to the enclosing block. But the enclosing block cannot reference the name `PAST_DUE`, because the scope where it was declared no longer exists. Once the exception name is lost, only an `OTHERS` handler can catch the exception. If there is no handler for a user-defined exception, the invoking application gets this error:

```
ORA-06510: PL/SQL: unhandled user-defined exception
```

Reraising a PL/SQL Exception

Sometimes, you want to reraise an exception, that is, handle it locally, then pass it to an enclosing block. For example, you might want to roll back a transaction in the current block, then log the error in an enclosing block.

To reraise an exception, use a `RAISE` statement without an exception name, which is allowed only in an exception handler:

Example 11–9 Reraising a PL/SQL Exception

```

DECLARE
  salary_too_high EXCEPTION;
  current_salary NUMBER := 20000;
  max_salary NUMBER := 10000;
  erroneous_salary NUMBER;
BEGIN
  BEGIN ----- sub-block begins
    IF current_salary > max_salary THEN
      RAISE salary_too_high; -- raise the exception
    END IF;
  EXCEPTION
    WHEN salary_too_high THEN
      -- first step in handling the error
      DBMS_OUTPUT.PUT_LINE('Salary ' || erroneous_salary ||
        ' is out of range. ');
      DBMS_OUTPUT.PUT_LINE
        ('Maximum salary is ' || max_salary || '. ');
      RAISE; -- reraise the current exception
    END; ----- sub-block ends
  EXCEPTION
    WHEN salary_too_high THEN
      -- handle the error more thoroughly
      erroneous_salary := current_salary;
      current_salary := max_salary;
      DBMS_OUTPUT.PUT_LINE('Revising salary from ' || erroneous_salary ||
        ' to ' || current_salary || '. ');
  END;
/

```

Handling Raised PL/SQL Exceptions

When an exception is raised, normal execution of your PL/SQL block or subprogram stops and control transfers to its exception-handling part, which is formatted as follows:

```

EXCEPTION
  WHEN exception1 THEN -- handler for exception1
    sequence_of_statements1
  WHEN exception2 THEN -- another handler for exception2
    sequence_of_statements2
  ...
  WHEN OTHERS THEN -- optional handler for all other errors
    sequence_of_statements3
END;

```

To catch raised exceptions, you write exception handlers. Each handler consists of a `WHEN` clause, which specifies an exception, followed by a sequence of statements to be executed when that exception is raised. These statements complete execution of the block or subprogram; control does not return to where the exception was raised. In other words, you cannot resume processing where you left off.

The optional `OTHERS` exception handler, which is always the last handler in a block or subprogram, acts as the handler for all exceptions not named specifically. Thus, a block or subprogram can have only one `OTHERS` handler. Use of the `OTHERS` handler guarantees that no exception will go unhandled.

If you want two or more exceptions to execute the same sequence of statements, list the exception names in the `WHEN` clause, separating them by the keyword `OR`, as follows:

```

EXCEPTION
  WHEN over_limit OR under_limit OR VALUE_ERROR THEN
    -- handle the error

```

If any of the exceptions in the list is raised, the associated sequence of statements is executed. The keyword `OTHERS` cannot appear in the list of exception names; it must appear by itself. You can have any number of exception handlers, and each handler can associate a list of exceptions with a sequence of statements. However, an exception name can appear only once in the exception-handling part of a PL/SQL block or subprogram.

The usual scoping rules for PL/SQL variables apply, so you can reference local and global variables in an exception handler. However, when an exception is raised inside a cursor `FOR` loop, the cursor is closed implicitly before the handler is invoked. Therefore, the values of explicit cursor attributes are *not* available in the handler.

Topics:

- [Exceptions Raised in Declarations](#)
- [Handling Exceptions Raised in Exception Handlers](#)
- [Branching To or from an Exception Handler](#)
- [Retrieving the Error Code and Error Message \(SQLCODE and SQLERRM Functions\)](#)
- [Catching Unhandled Exceptions](#)
- [Guidelines for Handling PL/SQL Errors](#)

Exceptions Raised in Declarations

Exceptions can be raised in declarations by faulty initialization expressions. For example, the following declaration raises an exception because the constant `credit_limit` cannot store numbers larger than 999:

Example 11–10 Raising an Exception in a Declaration

```
DECLARE
  -- Raises an error:
  credit_limit CONSTANT NUMBER(3) := 5000;
BEGIN
  NULL;
EXCEPTION
  WHEN OTHERS THEN
    -- Cannot catch exception. This handler is never invoked.
    DBMS_OUTPUT.PUT_LINE
      ('Can't handle an exception in a declaration.');
```

END;
/

Handlers in the current block cannot catch the raised exception because an exception raised in a declaration propagates immediately to the enclosing block.

Handling Exceptions Raised in Exception Handlers

When an exception occurs within an exception handler, that same handler cannot catch the exception. An exception raised inside a handler propagates immediately to the enclosing block, which is searched to find a handler for this new exception. From there on, the exception propagates normally. For example:

```
EXCEPTION
  WHEN INVALID_NUMBER THEN
    INSERT INTO ... -- might raise DUP_VAL_ON_INDEX
  WHEN DUP_VAL_ON_INDEX THEN -- cannot catch exception
END;
```

Branching To or from an Exception Handler

A GOTO statement can branch from an exception handler into an enclosing block.

A GOTO statement cannot branch into an exception handler, or from an exception handler into the current block.

Retrieving the Error Code and Error Message (SQLCODE and SQLERRM Functions)

In an exception handler, you can use the built-in functions `SQLCODE` and `SQLERRM` to find out which error occurred and to get the associated error message. For internal exceptions, `SQLCODE` returns the number of the Oracle error. The number that `SQLCODE` returns is negative unless the Oracle error is no data found, in which case `SQLCODE` returns +100. `SQLERRM` returns the corresponding error message. The message begins with the Oracle error code.

For user-defined exceptions, `SQLCODE` returns +1 and `SQLERRM` returns the message `User-Defined Exception` unless you used the pragma `EXCEPTION_INIT` to associate the exception name with an Oracle error number, in which case `SQLCODE` returns that error number and `SQLERRM` returns the corresponding error message. The maximum length of an Oracle error message is 512 characters including the error code, nested messages, and message inserts such as table and column names.

If no exception was raised, `SQLCODE` returns zero and `SQLERRM` returns the following message:

```
ORA-0000: normal, successful completion
```

You can pass an error number to `SQLERRM`, in which case `SQLERRM` returns the message associated with that error number. Make sure you pass negative error numbers to `SQLERRM`.

Passing a positive number to `SQLERRM` always returns the message user-defined exception unless you pass `+100`, in which case `SQLERRM` returns the message no data found. Passing a zero to `SQLERRM` always returns the following message:

```
ORA-0000: normal, successful completion
```

You cannot use `SQLCODE` or `SQLERRM` directly in a SQL statement. Instead, you must assign their values to local variables, then use the variables in the SQL statement, as shown in [Example 11-11](#).

Example 11-11 Displaying SQLCODE and SQLERRM

```
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 VALUES (v_code, v_errm, SYSTIMESTAMP);
END;
/
```

The string function `SUBSTR` ensures that a `VALUE_ERROR` exception (for truncation) is not raised when you assign the value of `SQLERRM` to `err_msg`. The functions `SQLCODE` and `SQLERRM` are especially useful in the `OTHERS` exception handler because they tell you which internal exception was raised.

When using pragma `RESTRICT_REFERENCES` to assert the purity of a stored function, you cannot specify the constraints `WNPS` and `RNPS` if the function invokes `SQLCODE` or `SQLERRM`.

Catching Unhandled Exceptions

Remember, if it cannot find a handler for a raised exception, PL/SQL returns an unhandled exception error to the host environment, which determines the outcome. For example, in the Oracle Precompilers environment, any database changes made by a failed SQL statement or PL/SQL block are rolled back.

Unhandled exceptions can also affect subprograms. If you exit a subprogram successfully, PL/SQL assigns values to `OUT` parameters. However, if you exit with an unhandled exception, PL/SQL does not assign values to `OUT` parameters (unless they are `NOCOPY` parameters). Also, if a stored subprogram fails with an unhandled exception, PL/SQL does not roll back database work done by the subprogram.

You can avoid unhandled exceptions by coding an OTHERS handler at the topmost level of every PL/SQL program.

Guidelines for Handling PL/SQL Errors

Topics:

- [Continuing Execution After an Exception Is Raised](#)
- [Retrying a Transaction](#)
- [Using Locator Variables to Identify Exception Locations](#)

Continuing Execution After an Exception Is Raised

An exception handler lets you recover from an otherwise fatal error before exiting a block. But when the handler completes, the block is terminated. You cannot return to the current block from an exception handler. In the following example, if the `SELECT INTO` statement raises `ZERO_DIVIDE`, you cannot resume with the `INSERT` statement:

```
CREATE TABLE employees_temp AS
  SELECT employee_id, salary,
         commission_pct FROM employees;

DECLARE
  sal_calc NUMBER(8,2);
BEGIN
  INSERT INTO employees_temp VALUES (301, 2500, 0);
  SELECT salary / commission_pct INTO sal_calc
    FROM employees_temp
   WHERE employee_id = 301;
  INSERT INTO employees_temp VALUES (302, sal_calc/100, .1);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
    NULL;
END;
/
```

You can still handle an exception for a statement, then continue with the next statement. Place the statement in its own sub-block with its own exception handlers. If an error occurs in the sub-block, a local handler can catch the exception. When the sub-block ends, the enclosing block continues to execute at the point where the sub-block ends, as shown in [Example 11–12](#).

Example 11–12 Continuing After an Exception

```
DECLARE
  sal_calc NUMBER(8,2);
BEGIN
  INSERT INTO employees_temp VALUES (303, 2500, 0);
  BEGIN -- sub-block begins
    SELECT salary / commission_pct INTO sal_calc
      FROM employees_temp
     WHERE employee_id = 301;
  EXCEPTION
    WHEN ZERO_DIVIDE THEN
      sal_calc := 2500;
  END; -- sub-block ends
  INSERT INTO employees_temp VALUES (304, sal_calc/100, .1);
EXCEPTION
  WHEN ZERO_DIVIDE THEN
```



```

        NULL;
    END;
/

```

In this example, if the `SELECT INTO` statement raises a `ZERO_DIVIDE` exception, the local handler catches it and sets `sal_calc` to 2500. Execution of the handler is complete, so the sub-block terminates, and execution continues with the `INSERT` statement.

See Also: [Example 5–38, "Collection Exceptions"](#) on page 5-28

You can also perform a sequence of DML operations where some might fail, and process the exceptions only after the entire operation is complete, as described in ["Handling FORALL Exceptions \(%BULK_EXCEPTIONS Attribute\)"](#) on page 12-17.

Retrying a Transaction

After an exception is raised, rather than abandon your transaction, you might want to retry it. The technique is:

1. Encase the transaction in a sub-block.
2. Place the sub-block inside a loop that repeats the transaction.
3. Before starting the transaction, mark a savepoint. If the transaction succeeds, commit, then exit from the loop. If the transaction fails, control transfers to the exception handler, where you roll back to the savepoint undoing any changes, then try to fix the problem.

In [Example 11–13](#), the `INSERT` statement might raise an exception because of a duplicate value in a unique column. In that case, we change the value that needs to be unique and continue with the next loop iteration. If the `INSERT` succeeds, we exit from the loop immediately. With this technique, use a `FOR` or `WHILE` loop to limit the number of attempts.

Example 11–13 Retrying a Transaction After an Exception

```

CREATE TABLE results (res_name VARCHAR(20), res_answer VARCHAR2(3));
CREATE UNIQUE INDEX res_name_ix ON results (res_name);
INSERT INTO results VALUES ('SMYTHE', 'YES');
INSERT INTO results VALUES ('JONES', 'NO');

DECLARE
    name      VARCHAR2(20) := 'SMYTHE';
    answer    VARCHAR2(3)  := 'NO';
    suffix    NUMBER := 1;
BEGIN
    FOR i IN 1..5 LOOP -- try 5 times
        BEGIN -- sub-block begins
            SAVEPOINT start_transaction; -- mark a savepoint
            /* Remove rows from a table of survey results. */
            DELETE FROM results WHERE res_answer = 'NO';
            /* Add a survey respondent's name and answers. */
            INSERT INTO results VALUES (name, answer);
            -- raises DUP_VAL_ON_INDEX
            -- if two respondents have the same name
            COMMIT;
            EXIT;
        EXCEPTION
            WHEN DUP_VAL_ON_INDEX THEN

```

```
        ROLLBACK TO start_transaction; -- undo changes
        suffix := suffix + 1;         -- try to fix problem
        name := name || TO_CHAR(suffix);
    END; -- sub-block ends
END LOOP;
END;
/
```

Using Locator Variables to Identify Exception Locations

Using one exception handler for a sequence of statements, such as INSERT, DELETE, or UPDATE statements, can mask the statement that caused an error. If you need to know which statement failed, you can use a locator variable, as in [Example 11-14](#).

Example 11-14 Using a Locator Variable to Identify the Location of an Exception

```
CREATE OR REPLACE PROCEDURE loc_var AS
    stmt_no NUMBER;
    name     VARCHAR2(100);
BEGIN
    stmt_no := 1; -- designates 1st SELECT statement
    SELECT table_name INTO name
        FROM user_tables
        WHERE table_name LIKE 'ABC%';
    stmt_no := 2; -- designates 2nd SELECT statement
    SELECT table_name INTO name
        FROM user_tables
        WHERE table_name LIKE 'XYZ%';
EXCEPTION
    WHEN NO_DATA_FOUND THEN
        DBMS_OUTPUT.PUT_LINE
            ('Table name not found in query ' || stmt_no);
END;
/
CALL loc_var();
```

Overview of PL/SQL Compile-Time Warnings

To make your programs more robust and avoid problems at run time, you can turn on checking for certain warning conditions. These conditions are not serious enough to produce an error and keep you from compiling a subprogram. They might point out something in the subprogram that produces an undefined result or might create a performance problem.

To work with PL/SQL warning messages, you use the `PLSQL_WARNINGS` initialization parameter, the `DBMS_WARNING` package, and the static data dictionary views `*_PLSQL_OBJECT_SETTINGS`.

Topics:

- [PL/SQL Warning Categories](#)
- [Controlling PL/SQL Warning Messages](#)
- [Using DBMS_WARNING Package](#)

PL/SQL Warning Categories

PL/SQL warning messages are divided into the categories listed and described in [Table 11-1](#). You can suppress or display groups of similar warnings during compilation. To refer to all warning messages, use the keyword `ALL`.

Table 11-1 *PL/SQL 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 be executed

You can also treat particular messages as errors instead of warnings. For example, if you know that the warning message `PLW-05003` represents a serious problem in your code, including `'ERROR:05003'` in the `PLSQL_WARNINGS` setting makes that condition trigger an error message (`PLS_05003`) instead of a warning message. An error message causes the compilation to fail.

Controlling PL/SQL Warning Messages

To let the database issue warning messages during PL/SQL compilation, you set the initialization parameter `PLSQL_WARNINGS`. You can enable and disable entire categories of warnings (`ALL`, `SEVERE`, `INFORMATIONAL`, `PERFORMANCE`), enable and disable specific message numbers, and make the database treat certain warnings as compilation errors so that those conditions must be corrected.

This parameter can be set at the system level or the session level. You can also set it for a single compilation by including it as part of an `ALTER COMPILER` statement. You might turn on all warnings during development, turn off all warnings when deploying for production, or turn on some warnings when working on a particular subprogram where you are concerned with some aspect, such as unnecessary code or performance.

Example 11-15 *Controlling the Display of PL/SQL Warnings*

```
-- Focus on one aspect:
ALTER SESSION
  SET PLSQL_WARNINGS='ENABLE:PERFORMANCE';
-- Recompile with extra checking:
ALTER PROCEDURE loc_var
  COMPILE PLSQL_WARNINGS='ENABLE:PERFORMANCE'
  REUSE SETTINGS;
-- Turn off warnings:
ALTER SESSION
  SET PLSQL_WARNINGS='DISABLE:ALL';
-- Display 'severe' warnings but not 'performance' warnings,
-- display PLW-06002 warnings to produce errors that halt compilation:
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:SEVERE',
  'DISABLE:PERFORMANCE', 'ERROR:06002';
-- For debugging during development
ALTER SESSION SET PLSQL_WARNINGS='ENABLE:ALL';
```

Warning messages can be issued during compilation of PL/SQL subprograms; anonymous blocks do not produce any warnings.

To see any warnings generated during compilation, use the SQL*Plus `SHOW ERRORS` statement or query the static data dictionary view `USER_ERRORS`. PL/SQL warning messages use the prefix `PLW`.

For general information about PL/SQL initialization parameters, see "[Initialization Parameters for PL/SQL Compilation](#)" on page 12-1.

Using `DBMS_WARNING` Package

If you are writing PL/SQL subprograms in a development environment that compiles them, you can control PL/SQL warning messages by invoking subprograms in the `DBMS_WARNING` package. You can also use this package when compiling a complex application, made up of several nested SQL*Plus scripts, where different warning settings apply to different subprograms. You can save the current state of the `PLSQL_WARNINGS` parameter with one call to the package, change the parameter to compile a particular set of subprograms, then restore the original parameter value.

The procedure in [Example 11-16](#) has unnecessary code that can be removed. It could represent a mistake, or it could be intentionally hidden by a debug flag, so you might or might not want a warning message for it.

Example 11-16 Using the `DBMS_WARNING` Package to Display Warnings

```
-- When warnings disabled,
-- the following procedure compiles with no warnings
CREATE OR REPLACE PROCEDURE unreachable_code AS
  x CONSTANT BOOLEAN := TRUE;
BEGIN
  IF x THEN
    DBMS_OUTPUT.PUT_LINE('TRUE');
  ELSE
    DBMS_OUTPUT.PUT_LINE('FALSE');
  END IF;
END unreachable_code;
/
-- enable all warning messages for this session
CALL DBMS_WARNING.set_warning_setting_string
  ('ENABLE:ALL' , 'SESSION');
-- Check the current warning setting
SELECT DBMS_WARNING.get_warning_setting_string() FROM DUAL;

-- Recompile procedure
-- and warning about unreachable code displays
ALTER PROCEDURE unreachable_code COMPILE;
SHOW ERRORS;
```

In [Example 11-16](#), you could have used the following `ALTER PROCEDURE` without invoking `DBMS_WARNINGS.set_warning_setting_string`:

```
ALTER PROCEDURE unreachable_code COMPILE
  PLSQL_WARNINGS = 'ENABLE:ALL' REUSE SETTINGS;
```

For more information, see `ALTER PROCEDURE` in *Oracle Database SQL Language Reference*, `DBMS_WARNING` package in *Oracle Database PL/SQL Packages and Types Reference*, and `PLW-` messages in *Oracle Database Error Messages*

Tuning PL/SQL Applications for Performance

This chapter explains how to write efficient new PL/SQL code and speed up existing PL/SQL code.

Topics:

- Initialization Parameters for PL/SQL Compilation
- How PL/SQL Optimizes Your Programs
- When to Tune PL/SQL Code
- Guidelines for Avoiding PL/SQL Performance Problems
- Collecting Data About User-Defined Identifiers
- Profiling and Tracing PL/SQL Programs
- Reducing Loop Overhead for DML Statements and Queries with Bulk SQL
- Writing Computation-Intensive PL/SQL Programs
- Tuning Dynamic SQL with EXECUTE IMMEDIATE Statement and Cursor Variables
- Tuning PL/SQL Subprogram Calls with NOCOPY Hint
- Compiling PL/SQL Program Units for Native Execution
- Performing Multiple Transformations with Pipelined Table Functions

Initialization Parameters for PL/SQL Compilation

Table 12-1 lists and briefly describes the initialization parameters that are used to compile PL/SQL units. The static data dictionary view ALL_PLSQL_OBJECT_SETTINGS displays the values of these parameters. For complete descriptions of these parameters and ALL_PLSQL_OBJECT_SETTINGS, see *Oracle Database Reference*.

Table 12-1 Initialization Parameters for PL/SQL Compilation¹

Parameter	Description
PLSQL_CCFLAGS ¹	Enables you to control conditional compilation of each PL/SQL unit independently.

Table 12–1 (Cont.) Initialization Parameters for PL/SQL Compilation¹

Parameter	Description
PLSQL_CODE_TYPE ¹	<p>Specifies the compilation mode for PL/SQL program units—<code>INTERPRETED</code> (the default) or <code>NATIVE</code>.</p> <p>If the optimization level (set by <code>PLSQL_OPTIMIZE_LEVEL</code>) is less than 2:</p> <ul style="list-style-type: none"> ■ The compiler generates interpreted code, regardless of <code>PLSQL_CODE_TYPE</code>. ■ If you specify <code>NATIVE</code>, the compiler warns you that <code>NATIVE</code> was ignored.
PLSQL_DEBUG ¹	Specifies whether or not PL/SQL units will be compiled for debugging. See note following table.
PLSQL_NATIVE_LIBRARY_DIR	Specifies the name of the directory where shared objects produced by the native compiler are stored. See note following table.
PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT	Specifies the number of subdirectories in the directory specified by <code>PLSQL_NATIVE_LIBRARY_DIR</code> . See note following table.
PLSQL_OPTIMIZE_LEVEL ¹	<p>Specifies the optimization level at which to compile PL/SQL units (the higher the level, the more optimizations the compiler tries to make).</p> <p>If <code>PLSQL_OPTIMIZE_LEVEL=1</code>, PL/SQL units will be compiled for debugging.</p>
PLSQL_WARNINGS ¹	Enables or disables the reporting of warning messages by the PL/SQL compiler, and specifies which warning messages to show as errors.
NLS_LENGTH_SEMANTICS ¹	Enables you to create <code>CHAR</code> and <code>VARCHAR2</code> columns using either byte or character length semantics.

¹ The compile-time value of this parameter is stored with the metadata of the PL/SQL program unit.

Note: The following initialization parameters are deprecated and might be unavailable in future Oracle Database releases:

- `PLSQL_DEBUG`
Instead of `PLSQL_DEBUG`, Oracle recommends using `PLSQL_OPTIMIZE_LEVEL=1`.
- `PLSQL_NATIVE_LIBRARY_DIR`
- `PLSQL_NATIVE_LIBRARY_SUBDIR_COUNT`

For Release 11.1, they have the same effects as they had for Release 10.2—described in [Table 12–1](#)—but the compiler warns you that they are deprecated.

If the compile-time value of an initialization parameter is stored with the metadata of the PL/SQL program unit, then you can reuse that value when you explicitly recompile the program unit, by doing the following:

1. Use one of the following statements to recompile the program unit:

- ALTER FUNCTION COMPILE
- ALTER PACKAGE COMPILE
- ALTER PROCEDURE COMPILE

2. Include the REUSE SETTINGS clause in the statement.

This clause preserves the existing settings and uses them for the recompilation of any parameters for which values are not specified elsewhere in the statement.

If you use the SQL statement CREATE OR REPLACE to explicitly compile a PL/SQL subprogram, or if you do not include the REUSE SETTINGS clause in the ALTER COMPILE statement, then the value of the initialization parameter is its value for the session.

See Also:

- *Oracle Database SQL Language Reference* for more information about the ALTER FUNCTION statement
- *Oracle Database SQL Language Reference* for more information about the ALTER PACKAGE statement
- *Oracle Database SQL Language Reference* for more information about the ALTER PROCEDURE statement

How PL/SQL Optimizes Your Programs

Prior to Oracle Database Release 10g, the PL/SQL compiler translated your code to machine code without applying many changes to improve performance. Now, PL/SQL uses an optimizing compiler 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 take too long, you can lower the optimization by setting the initialization parameter PLSQL_OPTIMIZE_LEVEL=1 instead of its default value 2. In even rarer cases, you might see a change in exception action, either an exception that is not raised at all, or one that is raised earlier than expected. Setting PLSQL_OPTIMIZE_LEVEL=1 prevents the code from being rearranged.

One optimization that the compiler can perform is **subprogram inlining**. Subprogram inlining replaces a subprogram call (to a subprogram in the same program unit) with a copy of the called subprogram.

To allow subprogram inlining, either accept the default value of the PLSQL_OPTIMIZE_LEVEL initialization parameter (which is 2) or set it to 3. With PLSQL_OPTIMIZE_LEVEL=2, you must specify each subprogram to be inlined. With PLSQL_OPTIMIZE_LEVEL=3, the PL/SQL compiler seeks opportunities to inline subprograms beyond those that you specify.

If a particular subprogram is inlined, performance almost always improves. However, because the compiler inlines subprograms early in the optimization process, it is possible for subprogram inlining to preclude later, more powerful optimizations.

If subprogram inlining slows the performance of a particular PL/SQL program, use the PL/SQL hierarchical profiler to identify subprograms for which you want to turn

off inlining. To turn off inlining for a subprogram, use the `INLINE` pragma, described in "[INLINE Pragma](#)" on page 13-82.

See Also:

- *Oracle Database Advanced Application Developer's Guide* for information about the PL/SQL hierarchical profiler
- *Oracle Database Reference* for information about the `PLSQL_OPTIMIZE_LEVEL` initialization parameter
- *Oracle Database Reference* for information about the static dictionary view `ALL_PLSQL_OBJECT_SETTINGS`

When to Tune PL/SQL Code

The information in this chapter is especially valuable if you are responsible for:

- Programs that do a lot of mathematical calculations. You will want to investigate the datatypes `PLS_INTEGER`, `BINARY_FLOAT`, and `BINARY_DOUBLE`.
- Functions that are called from PL/SQL queries, where the functions might be executed millions of times. You will want to look at all performance features to make the function as efficient as possible, and perhaps a function-based index to precompute the results for each row and save on query time.
- Programs that spend a lot of time processing `INSERT`, `UPDATE`, or `DELETE` statements, or looping through query results. You will want to investigate the `FORALL` statement for issuing DML, and the `BULK COLLECT INTO` and `RETURNING BULK COLLECT INTO` clauses for queries.
- Older code that does not take advantage of recent PL/SQL language features. With the many performance improvements in Oracle Database 10g, any code from earlier releases is a candidate for tuning.
- Any program that spends a lot of time doing PL/SQL processing, as opposed to issuing DDL statements like `CREATE TABLE` that are just passed directly to SQL. You will want to investigate native compilation. Because many built-in database features use PL/SQL, you can apply this tuning feature to an entire database to improve performance in many areas, not just your own code.

Before starting any tuning effort, benchmark the current system and measure how long particular subprograms take. PL/SQL in Oracle Database 10g includes many automatic optimizations, so you might see performance improvements without doing any tuning.

Guidelines for Avoiding PL/SQL Performance Problems

When a PL/SQL-based application performs poorly, it is often due to badly written SQL statements, poor programming practices, inattention to PL/SQL basics, or misuse of shared memory.

Topics:

- [Avoiding CPU Overhead in PL/SQL Code](#)
- [Avoiding Memory Overhead in PL/SQL Code](#)

Avoiding CPU Overhead in PL/SQL Code

Topics:

- [Make SQL Statements as Efficient as Possible](#)
- [Make Function Calls as Efficient as Possible](#)
- [Make Loops as Efficient as Possible](#)
- [Use Built-In String Functions](#)
- [Put Least Expensive Conditional Tests First](#)
- [Minimize Datatype Conversions](#)
- [Use PLS_INTEGER or SIMPLE_INTEGER for Integer Arithmetic](#)
- [Use BINARY_FLOAT and BINARY_DOUBLE for Floating-Point Arithmetic](#)

Make SQL Statements as Efficient as Possible

PL/SQL programs look relatively simple because most of the work is done by SQL statements. Slow SQL statements are the main reason for slow execution.

If SQL statements are slowing down your program:

- Make sure you have appropriate indexes. There are different kinds of indexes for different situations. Your index strategy might be different depending on the sizes of various tables in a query, the distribution of data in each query, and the columns used in the `WHERE` clauses.
- Make sure you have up-to-date statistics on all the tables, using the subprograms in the `DBMS_STATS` package.
- Analyze the execution plans and performance of the SQL statements, using:
 - `EXPLAIN PLAN` statement
 - SQL Trace facility with `TKPROF` utility
- Rewrite the SQL statements if necessary. For example, query hints can avoid problems such as unnecessary full-table scans.

For more information about these methods, see *Oracle Database Performance Tuning Guide*.

Some PL/SQL features also help improve the performance of SQL statements:

- If you are running SQL statements inside a PL/SQL loop, look at the `FORALL` statement as a way to replace loops of `INSERT`, `UPDATE`, and `DELETE` statements.
- If you are looping through the result set of a query, look at the `BULK COLLECT` clause of the `SELECT INTO` statement as a way to bring the entire result set into memory in a single operation.

Make Function Calls as Efficient as Possible

Badly written subprograms (for example, a slow sort or search function) can harm performance. Avoid unnecessary calls to subprograms, and optimize their code:

- If a function is called within a SQL query, you can cache the function value for each row by creating a function-based index on the table in the query. The `CREATE INDEX` statement might take a while, but queries can be much faster.
- If a column is passed to a function within an SQL query, the query cannot use regular indexes on that column, and the function might be called for every row in a (potentially very large) table. Consider nesting the query so that the inner query filters the results to a small number of rows, and the outer query calls the function only a few times as shown in [Example 12-1](#).

Example 12–1 Nesting a Query to Improve Performance

```

BEGIN
-- Inefficient, calls function for every row
  FOR item IN (SELECT DISTINCT(SQRT(department_id)) col_alias FROM employees)
  LOOP
    DBMS_OUTPUT.PUT_LINE(item.col_alias);
  END LOOP;
-- Efficient, only calls function once for each distinct value.
  FOR item IN
    ( SELECT SQRT(department_id) col_alias FROM
      ( SELECT DISTINCT department_id FROM employees)
    )
  LOOP
    DBMS_OUTPUT.PUT_LINE(item.col_alias);
  END LOOP;
END;
/

```

If you use `OUT` or `IN OUT` parameters, PL/SQL adds some performance overhead to ensure correct action in case of exceptions (assigning a value to the `OUT` parameter, then exiting the subprogram because of an unhandled exception, so that the `OUT` parameter keeps its original value).

If your program does not depend on `OUT` parameters keeping their values in such situations, you can add the `NOCOPY` keyword to the parameter declarations, so the parameters are declared `OUT NOCOPY` or `IN OUT NOCOPY`.

This technique can give significant speedup if you are passing back large amounts of data in `OUT` parameters, such as collections, big `VARCHAR2` values, or LOBs.

This technique also applies to member methods of object types. If these methods modify attributes of the object type, all the attributes are copied when the method ends. To avoid this overhead, you can explicitly declare the first parameter of the member method as `SELF IN OUT NOCOPY`, instead of relying on PL/SQL's implicit declaration `SELF IN OUT`. For information about design considerations for object methods, see *Oracle Database Object-Relational Developer's Guide*.

Make Loops as Efficient as Possible

Because PL/SQL applications are often built around loops, it is important to optimize the loop itself and the code inside the loop:

- To issue a series of DML statements, replace loop constructs with `FORALL` statements.
- To loop through a result set and store the values, use the `BULK COLLECT` clause on the query to bring the query results into memory in one operation.
- If you have to loop through a result set more than once, or issue other queries as you loop through a result set, you can probably enhance the original query to give you exactly the results you want. Some query operators to explore include `UNION`, `INTERSECT`, `MINUS`, and `CONNECT BY`.
- You can also nest one query inside another (known as a subselect) to do the filtering and sorting in multiple stages. For example, instead of calling a PL/SQL function in the inner `WHERE` clause (which might call the function once for each row of the table), you can filter the result set to a small set of rows in the inner query, and call the function in the outer query.

Use Built-In String Functions

PL/SQL provides many highly optimized string functions such as `REPLACE`, `TRANSLATE`, `SUBSTR`, `INSTR`, `RPAD`, and `LTRIM`. The built-in functions use low-level code that is more efficient than regular PL/SQL.

If you use PL/SQL string functions to search for regular expressions, consider using the built-in regular expression functions, such as `REGEXP_SUBSTR`.

- You can search for regular expressions using the SQL operator `REGEXP_LIKE`. See [Example 6–10](#) on page 6-11.
- You can test or manipulate strings using the built-in functions `REGEXP_INSTR`, `REGEXP_REPLACE`, and `REGEXP_SUBSTR`.

Oracle's regular expression features use characters like '.', '*', '^', and '\$' that you might be familiar with from Linux, UNIX, or Perl programming. For multi-language programming, there are also extensions such as '[:lower:]' to match a lowercase letter, instead of '[a-z]' which does not match lowercase accented letters.

Put Least Expensive Conditional Tests First

PL/SQL stops evaluating a logical expression as soon as the result can be determined. This functionality is known as short-circuit evaluation. See ["Short-Circuit Evaluation"](#) on page 2-23.

When evaluating multiple conditions separated by `AND` or `OR`, put the least expensive ones first. For example, check the values of PL/SQL variables before testing function return values, because PL/SQL might be able to skip calling the functions.

Minimize Datatype Conversions

At run time, PL/SQL converts between different datatypes automatically. For example, assigning a `PLS_INTEGER` variable to a `NUMBER` variable results in a conversion because their internal representations are different.

Whenever possible, choose datatypes carefully to minimize implicit conversions. Use literals of the appropriate types, such as character literals in character expressions and decimal numbers in number expressions.

Minimizing conversions might mean changing the types of your variables, or even working backward and designing your tables with different datatypes. Or, you might convert data once, such as from an `INTEGER` column to a `PLS_INTEGER` variable, and use the PL/SQL type consistently after that. The conversion from `INTEGER` to `PLS_INTEGER` datatype might improve performance, because of the use of more efficient hardware arithmetic. See ["Use PLS_INTEGER or SIMPLE_INTEGER for Integer Arithmetic"](#) on page 12-7.

Use PLS_INTEGER or SIMPLE_INTEGER for Integer Arithmetic

When declaring a local integer variable:

- If the value of the variable might be `NULL`, or if the variable needs overflow checking, use the datatype `PLS_INTEGER`.
- If the value of the variable will never be `NULL`, and the variable does not need overflow checking, use the datatype `SIMPLE_INTEGER`.

`PLS_INTEGER` values use less storage space than `INTEGER` or `NUMBER` values, and `PLS_INTEGER` operations use hardware arithmetic. For more information, see ["PLS_INTEGER and BINARY_INTEGER Datatypes"](#) on page 3-2.

`SIMPLE_INTEGER` is a predefined subtype of `PLS_INTEGER`. It has the same range as `PLS_INTEGER` and has a `NOT NULL` constraint. It differs significantly from `PLS_INTEGER` in its overflow semantics—for details, see ["Overflow Semantics"](#) on page 3-3.

The datatype `NUMBER` and its subtypes are represented in a special internal format, designed for portability and arbitrary scale and precision, not performance. Even the subtype `INTEGER` is treated as a floating-point number with nothing after the decimal point. Operations on `NUMBER` or `INTEGER` variables require calls to library routines.

Avoid constrained subtypes such as `INTEGER`, `NATURAL`, `NATURALN`, `POSITIVE`, `POSITIVEN`, and `SIGNTYPE` in performance-critical code. Variables of these types require extra checking at run time, each time they are used in a calculation.

Use `BINARY_FLOAT` and `BINARY_DOUBLE` for Floating-Point Arithmetic

The datatype `NUMBER` and its subtypes are represented in a special internal format, designed for portability and arbitrary scale and precision, not performance. Operations on `NUMBER` or `INTEGER` variables require calls to library routines.

The `BINARY_FLOAT` and `BINARY_DOUBLE` types can use native hardware arithmetic instructions, and are more efficient for number-crunching applications such as scientific processing. They also require less space in the database.

These types do not always represent fractional values precisely, and handle rounding differently than the `NUMBER` types. These types are less suitable for financial code where accuracy is critical.

Avoiding Memory Overhead in PL/SQL Code

Topics:

- [Declare `VARCHAR2` Variables of 4000 or More Characters](#)
- [Group Related Subprograms into Packages](#)
- [Pin Packages in the Shared Memory Pool](#)
- [Apply Advice of Compiler Warnings](#)

Declare `VARCHAR2` Variables of 4000 or More Characters

You might need to allocate large `VARCHAR2` variables when you are not sure how big an expression result will be. You can actually conserve memory by declaring `VARCHAR2` variables with large sizes, such as 32000, rather than estimating just a little on the high side, such as by specifying 256 or 1000. PL/SQL has an optimization that makes it easy to avoid overflow problems and still conserve memory. Specify a size of more than 4000 characters for the `VARCHAR2` variable; PL/SQL waits until you assign the variable, then only allocates as much storage as needed.

Group Related Subprograms into Packages

When you call a packaged subprogram for the first time, the whole package is loaded into the shared memory pool. Subsequent calls to related subprograms in the package require no disk I/O, and your code executes faster. If the package is aged out of memory, it must be reloaded if you reference it again.

You can improve performance by sizing the shared memory pool correctly. Make sure it is large enough to hold all frequently used packages but not so large that memory is wasted.

Pin Packages in the Shared Memory Pool

You can pin frequently accessed packages in the shared memory pool, using the supplied package `DBMS_SHARED_POOL`. When a package is pinned, it is not aged out by the least recently used (LRU) algorithm that Oracle normally uses. The package remains in memory no matter how full the pool gets or how frequently you access the package.

For more information on the `DBMS_SHARED_POOL` package, see *Oracle Database PL/SQL Packages and Types Reference*.

Apply Advice of Compiler Warnings

The PL/SQL compiler issues warnings about things that do not make a program incorrect, but might lead to poor performance. If you receive such a warning, and the performance of this code is important, follow the suggestions in the warning and change the code to be more efficient.

Collecting Data About User-Defined Identifiers

PL/Scope extracts, organizes, and stores data about user-defined identifiers from PL/SQL source code. You can retrieve source code identifier data with the static data dictionary views `*_IDENTIFIERS`. For more information, see *Oracle Database Advanced Application Developer's Guide*.

Profiling and Tracing PL/SQL Programs

To help you isolate performance problems in large PL/SQL programs, PL/SQL provides the following tools, implemented as PL/SQL packages:

Tool	Package	Description
Profiler API	<code>DBMS_PROFILER</code>	<p>Computes the time that your PL/SQL program spends at each line and in each subprogram.</p> <p>You must have <code>CREATE</code> privileges on the units to be profiled.</p> <p>Saves run-time statistics in database tables, which you can query.</p>
Trace API	<code>DBMS_TRACE</code>	<p>Traces the order in which subprograms execute.</p> <p>You can specify the subprograms to trace and the tracing level.</p> <p>Saves run-time statistics in database tables, which you can query.</p>
PL/SQL hierarchical profiler	<code>DBMS_HPROF</code>	<p>Reports the dynamic execution program profile of your PL/SQL program, organized by subprogram calls. Accounts for SQL and PL/SQL execution times separately.</p> <p>Requires no special source or compile-time preparation.</p> <p>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).</p>

Topics:

- [Using the Profiler API: Package DBMS_PROFILER](#)
- [Using the Trace API: Package DBMS_TRACE](#)

For a detailed description of PL/SQL hierarchical profiler, see *Oracle Database Advanced Application Developer's Guide*.

Using the 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 executing 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, due to 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*.

Using the 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 ["Using the Profiler API: Package DBMS_PROFILER"](#) on page 12-10).

For detailed information about the DBMS_TRACE subprograms, see *Oracle Database PL/SQL Packages and Types Reference*.

Reducing Loop Overhead for DML Statements and Queries with Bulk SQL

PL/SQL sends SQL statements such as DML and queries to the SQL engine for execution, and SQL returns the result data to PL/SQL. You can minimize the performance overhead of this communication between PL/SQL and SQL by using the PL/SQL language features known collectively as bulk SQL. The FORALL statement sends INSERT, UPDATE, or DELETE statements in batches, rather than one at a time. The BULK COLLECT clause brings back batches of results from SQL. If the DML statement affects four or more database rows, the use of bulk SQL can improve performance considerably.

The assigning of values to PL/SQL variables in SQL statements is called binding. PL/SQL binding operations fall into the following categories:

Binding Category	When This Binding Occurs
in-bind	When an INSERT or UPDATE statement stores a PL/SQL variable or host variable in the database
out-bind	When the RETURNING clause of an INSERT, UPDATE, or DELETE statement assigns a database value to a PL/SQL variable or host variable
define	When a SELECT or FETCH statement assigns a database value to a PL/SQL variable or host variable

Bulk SQL uses PL/SQL collections, such as varrays or nested tables, to pass large amounts of data back and forth in a single operation. This process is known as bulk binding. If the collection has 20 elements, bulk binding lets you perform the equivalent of 20 SELECT, INSERT, UPDATE, or DELETE statements using a single operation. Queries can pass back any number of results, without requiring a FETCH statement for each row.

Note: Parallel DML is disabled with bulk binds.

To speed up INSERT, UPDATE, and DELETE statements, enclose the SQL statement within a PL/SQL FORALL statement instead of a loop construct.

To speed up SELECT statements, include the BULK COLLECT INTO clause in the SELECT statement instead of using INTO.

For syntax of, restrictions on, and usage notes for these statements, see ["FORALL Statement"](#) on page 13-71 and ["SELECT INTO Statement"](#) on page 13-128.

Topics:

- [Running One DML Statement Multiple Times \(FORALL Statement\)](#)
- [Retrieving Query Results into Collections \(BULK COLLECT Clause\)](#)

Running One DML Statement Multiple Times (FORALL Statement)

The keyword `FORALL` lets you run multiple DML statements very efficiently. It can only repeat a single DML statement, unlike a general-purpose `FOR` loop. For full syntax and restrictions, see "[FORALL Statement](#)" on page 13-71.

The SQL statement can reference more than one collection, but `FORALL` only improves performance where the index value is used as a subscript.

Usually, the bounds specify a range of consecutive index numbers. If the index numbers are not consecutive, such as after you delete collection elements, you can use the `INDICES OF` or `VALUES OF` clause to iterate over just those index values that really exist.

The `INDICES OF` clause iterates over all of the index values in the specified collection, or only those between a lower and upper bound.

The `VALUES OF` clause refers to a collection that is indexed by `PLS_INTEGER` and whose elements are of type `PLS_INTEGER`. The `FORALL` statement iterates over the index values specified by the elements of this collection.

The `FORALL` statement in [Example 12-2](#) sends all three `DELETE` statements to the SQL engine at once.

Example 12-2 Issuing DELETE Statements in a Loop

```
CREATE TABLE employees_temp AS SELECT * FROM employees;
DECLARE
    TYPE NumList IS VARRAY(20) OF NUMBER;
    depts NumList := NumList(10, 30, 70); -- department numbers
BEGIN
    FORALL i IN depts.FIRST..depts.LAST
        DELETE FROM employees_temp WHERE department_id = depts(i);
    COMMIT;
END;
/
```

[Example 12-3](#) loads some data into PL/SQL collections. Then it inserts the collection elements into a database table twice: first using a `FOR` loop, then using a `FORALL` statement. The `FORALL` version is much faster.

Example 12-3 Issuing INSERT Statements in a Loop

```
CREATE TABLE parts1 (pnum INTEGER, pname VARCHAR2(15));
CREATE TABLE parts2 (pnum INTEGER, pname VARCHAR2(15));
DECLARE
    TYPE NumTab IS TABLE OF parts1.pnum%TYPE INDEX BY PLS_INTEGER;
    TYPE NameTab IS TABLE OF parts1.pname%TYPE INDEX BY PLS_INTEGER;
    pnums NumTab;
    pnames NameTab;
    iterations CONSTANT PLS_INTEGER := 500;
    t1 INTEGER;
    t2 INTEGER;
    t3 INTEGER;
BEGIN
    FOR j IN 1..iterations LOOP -- load index-by tables
        pnums(j) := j;
        pnames(j) := 'Part No. ' || TO_CHAR(j);
    END LOOP;
    t1 := DBMS_UTILITY.get_time;
    FOR i IN 1..iterations LOOP -- use FOR loop
```



```

        INSERT INTO parts1 VALUES (pnums(i), pnames(i));
    END LOOP;
    t2 := DBMS_UTILITY.get_time;
    FORALL i IN 1..iterations -- use FORALL statement
        INSERT INTO parts2 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;
/

```

Executing this block shows that the loop using FORALL is much faster.

The bounds of the FORALL loop can apply to part of a collection, not necessarily all the elements, as shown in [Example 12-4](#).

Example 12-4 Using FORALL with Part of a Collection

```

CREATE TABLE employees_temp AS SELECT * FROM employees;
DECLARE
    TYPE NumList IS VARRAY(10) OF NUMBER;
    depts NumList := NumList(5,10,20,30,50,55,57,60,70,75);
BEGIN
    FORALL j IN 4..7 -- use only part of varray
        DELETE FROM employees_temp WHERE department_id = depts(j);
    COMMIT;
END;
/

```

You might need to delete some elements from a collection before using the collection in a FORALL statement. The INDICES OF clause processes sparse collections by iterating through only the remaining elements.

You might also want to leave the original collection alone, but process only some elements, process the elements in a different order, or process some elements more than once. Instead of copying the entire elements into new collections, which might use up substantial amounts of memory, the VALUES OF clause lets you set up simple collections whose elements serve as pointers to elements in the original collection.

[Example 12-5](#) creates a collection holding some arbitrary data, a set of table names. Deleting some of the elements makes it a sparse collection that does not work in a default FORALL statement. The program uses a FORALL statement with the INDICES OF clause to insert the data into a table. It then sets up two more collections, pointing to certain elements from the original collection. The program stores each set of names in a different database table using FORALL statements with the VALUES OF clause.

Example 12-5 Using FORALL with Nonconsecutive Index Values

```

-- Create empty tables to hold order details
CREATE TABLE valid_orders (cust_name VARCHAR2(32),
                           amount NUMBER(10,2));
CREATE TABLE big_orders AS SELECT * FROM valid_orders
    WHERE 1 = 0;
CREATE TABLE rejected_orders AS SELECT * FROM valid_orders
    WHERE 1 = 0;
DECLARE
-- Collections for set of customer names & order amounts:

```

```

SUBTYPE cust_name IS valid_orders.cust_name%TYPE;
TYPE cust_typ IS TABLE OF cust_name;
cust_tab cust_typ;
SUBTYPE order_amount IS valid_orders.amount%TYPE;
TYPE amount_typ IS TABLE OF NUMBER;
amount_tab amount_typ;
-- Collections to point into CUST_TAB collection.
TYPE index_pointer_t IS TABLE OF PLS_INTEGER;
big_order_tab index_pointer_t := index_pointer_t();
rejected_order_tab index_pointer_t := index_pointer_t();
PROCEDURE setup_data IS BEGIN
-- Set up sample order data,
-- including some invalid orders and some 'big' orders.
    cust_tab := cust_typ('Company1','Company2',
        'Company3','Company4','Company5');
    amount_tab := amount_typ(5000.01, 0,
        150.25, 4000.00, NULL);
END;
BEGIN
    setup_data();
    DBMS_OUTPUT.PUT_LINE
        ('--- Original order data ---');
    FOR i IN 1..cust_tab.LAST LOOP
        DBMS_OUTPUT.PUT_LINE
            ('Customer #' || i || ', ' || cust_tab(i) || ': $' ||
                amount_tab(i));
    END LOOP;
-- Delete invalid orders (where amount is null or 0).
    FOR i IN 1..cust_tab.LAST LOOP
        IF amount_tab(i) IS NULL OR amount_tab(i) = 0 THEN
            cust_tab.delete(i);
            amount_tab.delete(i);
        END IF;
    END LOOP;
    DBMS_OUTPUT.PUT_LINE
        ('--- Data with invalid orders deleted ---');
    FOR i IN 1..cust_tab.LAST LOOP
        IF cust_tab.EXISTS(i) THEN
            DBMS_OUTPUT.PUT_LINE('Customer #' || i || ', ' ||
                cust_tab(i) || ': $' || amount_tab(i));
        END IF;
    END LOOP;
-- Because subscripts of collections are not consecutive,
-- use FORALL...INDICES OF to iterate through actual subscripts,
-- rather than 1..COUNT
    FORALL i IN INDICES OF cust_tab
        INSERT INTO valid_orders(cust_name, amount)
            VALUES(cust_tab(i), amount_tab(i));
-- Now process the order data differently
-- Extract 2 subsets and store each subset in a different table
-- Initialize the CUST_TAB and AMOUNT_TAB collections again.
    setup_data();
    FOR i IN cust_tab.FIRST .. cust_tab.LAST LOOP
        IF amount_tab(i) IS NULL OR amount_tab(i) = 0 THEN
            -- Add a new element to this collection
            rejected_order_tab.EXTEND;
-- Record the subscript from the original collection
            rejected_order_tab(rejected_order_tab.LAST) := i;
        END IF;
        IF amount_tab(i) > 2000 THEN

```

```

        -- Add a new element to this collection
        big_order_tab.EXTEND;
    -- Record the subscript from the original collection
        big_order_tab(big_order_tab.LAST) := i;
    END IF;
    END LOOP;
    -- Now it's easy to run one DML statement
    -- on one subset of elements,
    -- and another DML statement on a different subset.
    FORALL i IN VALUES OF rejected_order_tab
        INSERT INTO rejected_orders
            VALUES (cust_tab(i), amount_tab(i));
    FORALL i IN VALUES OF big_order_tab
        INSERT INTO big_orders
            VALUES (cust_tab(i), amount_tab(i));
    COMMIT;
END;
/
-- Verify that the correct order details were stored
SELECT cust_name "Customer",
       amount "Valid order amount" FROM valid_orders;
SELECT cust_name "Customer",
       amount "Big order amount" FROM big_orders;
SELECT cust_name "Customer",
       amount "Rejected order amount" FROM rejected_orders;

```

Topics:

- [How FORALL Affects Rollbacks](#)
- [Counting Rows Affected by FORALL \(%BULK_ROWCOUNT Attribute\)](#)
- [Handling FORALL Exceptions \(%BULK_EXCEPTIONS Attribute\)](#)

How FORALL Affects Rollbacks

In a FORALL statement, if any execution of the SQL statement raises an unhandled exception, all database changes made during previous executions are rolled back. However, if a raised exception is caught and handled, changes are rolled back to an implicit savepoint marked before each execution of the SQL statement. Changes made during previous executions are not rolled back. For example, suppose you create a database table that stores department numbers and job titles, as shown in [Example 12–6](#). Then, you change the job titles so that they are longer. The second UPDATE fails because the new value is too long for the column. Because we handle the exception, the first UPDATE is not rolled back and we can commit that change.

Example 12–6 Using Rollbacks with FORALL

```

CREATE TABLE emp_temp (deptno NUMBER(2), job VARCHAR2(18));
DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    depts NumList := NumList(10, 20, 30);
BEGIN
    INSERT INTO emp_temp VALUES(10, 'Clerk');
    -- Lengthening this job title causes an exception
    INSERT INTO emp_temp VALUES(20, 'Bookkeeper');
    INSERT INTO emp_temp VALUES(30, 'Analyst');
    COMMIT;
    FORALL j IN depts.FIRST..depts.LAST -- Run 3 UPDATE statements.
        UPDATE emp_temp SET job = job || ' (Senior)'
            WHERE deptno = depts(j);

```

```

-- raises a "value too large" exception
EXCEPTION
  WHEN OTHERS THEN
    DBMS_OUTPUT.PUT_LINE
      ('Problem in the FORALL statement.');
```

COMMIT; -- Commit results of successful updates.

```

END;
/
```

Counting Rows Affected by FORALL (%BULK_ROWCOUNT Attribute)

The cursor attributes `SQL%FOUND`, `SQL%ISOPEN`, `SQL%NOTFOUND`, and `SQL%ROWCOUNT`, return useful information about the most recently executed DML statement. For additional description of cursor attributes, see ["Implicit Cursors"](#) on page 6-7.

The SQL cursor has one composite attribute, `%BULK_ROWCOUNT`, for use with the `FORALL` statement. This attribute works like an associative array: `SQL%BULK_ROWCOUNT(i)` stores the number of rows processed by the *i*th execution of an `INSERT`, `UPDATE` or `DELETE` statement, as in [Example 12-7](#).

Example 12-7 Using %BULK_ROWCOUNT with the FORALL Statement

```

CREATE TABLE emp_temp AS SELECT * FROM employees;
DECLARE
  TYPE NumList IS TABLE OF NUMBER;
  depts NumList := NumList(30, 50, 60);
BEGIN
  FORALL j IN depts.FIRST..depts.LAST
    DELETE FROM emp_temp WHERE department_id = depts(j);
  -- How many rows were affected by each DELETE statement?
  FOR i IN depts.FIRST..depts.LAST
    LOOP
      DBMS_OUTPUT.PUT_LINE('Iteration #' || i || ' deleted ' ||
        SQL%BULK_ROWCOUNT(i) || ' rows.');
```

END LOOP;

```

END;
/
```

The `FORALL` statement and `%BULK_ROWCOUNT` attribute use the same subscripts. For example, if `FORALL` uses the range `5..10`, so does `%BULK_ROWCOUNT`. If the `FORALL` statement uses the `INDICES OF` clause to process a sparse collection, `%BULK_ROWCOUNT` has corresponding sparse subscripts. If the `FORALL` statement uses the `VALUES OF` clause to process a subset of elements, `%BULK_ROWCOUNT` has subscripts corresponding to the values of the elements in the index collection. If the index collection contains duplicate elements, so that some DML statements are issued multiple times using the same subscript, then the corresponding elements of `%BULK_ROWCOUNT` represent the sum of all rows affected by the DML statement using that subscript.

`%BULK_ROWCOUNT` is usually equal to 1 for inserts, because a typical insert operation affects only a single row. For the `INSERT SELECT` construct, `%BULK_ROWCOUNT` might be greater than 1. For example, the `FORALL` statement in [Example 12-8](#) inserts an arbitrary number of rows for each iteration. After each iteration, `%BULK_ROWCOUNT` returns the number of items inserted.

Example 12-8 Counting Rows Affected by FORALL with %BULK_ROWCOUNT

```

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
            SELECT employee_id, department_id FROM employees
                WHERE department_id = deptnums(i);
    FOR i IN 1..deptnums.COUNT LOOP
-- Count how many rows were inserted for each department; that is,
-- how many employees are in each department.
        DBMS_OUTPUT.PUT_LINE('Dept ' || deptnums(i) || ': inserted ' ||
            SQL%BULK_ROWCOUNT(i) || ' records');
    END LOOP;
    DBMS_OUTPUT.PUT_LINE('Total records inserted: ' || SQL%ROWCOUNT);
END;
/

```

You can also use the scalar attributes `%FOUND`, `%NOTFOUND`, and `%ROWCOUNT` after running a `FORALL` statement. For example, `%ROWCOUNT` returns the total number of rows processed by all executions of the SQL statement.

`%FOUND` and `%NOTFOUND` refer only to the last execution of the SQL statement. You can use `%BULK_ROWCOUNT` to infer their values for individual executions. For example, when `%BULK_ROWCOUNT(i)` is zero, `%FOUND` and `%NOTFOUND` are `FALSE` and `TRUE`, respectively.

Handling FORALL Exceptions (%BULK_EXCEPTIONS Attribute)

PL/SQL provides a mechanism to handle exceptions raised during the execution of a `FORALL` statement. This mechanism enables a bulk-bind operation to save information about exceptions and continue processing.

To have a bulk bind complete despite errors, add the keywords `SAVE EXCEPTIONS` to your `FORALL` statement after the bounds, before the DML statement. Provide an exception handler to track the exceptions that occurred during the bulk operation.

[Example 12-9](#) shows how you can perform a number of DML operations, without stopping if some operations encounter errors. In the example, `EXCEPTION_INIT` is used to associate the `dml_errors` exception with the `ORA-24381` error. The `ORA-24381` error is raised if any exceptions are caught and saved after a bulk operation.

All exceptions raised during the execution are saved in the cursor attribute `%BULK_EXCEPTIONS`, which stores a collection of records. Each record has two fields:

- `%BULK_EXCEPTIONS(i).ERROR_INDEX` holds the iteration of the `FORALL` statement during which the exception was raised.
- `%BULK_EXCEPTIONS(i).ERROR_CODE` holds the corresponding Oracle error code.

The values stored by `%BULK_EXCEPTIONS` always refer to the most recently executed `FORALL` statement. The number of exceptions is saved in `%BULK_EXCEPTIONS.COUNT`. Its subscripts range from 1 to `COUNT`.

The individual error messages, or any substitution arguments, are not saved, but the error message text can be looked up using `ERROR_CODE` with `SQLERRM` as shown in [Example 12-9](#).

You might need to work backward to determine which collection element was used in the iteration that caused an exception. For example, if you use the `INDICES OF` clause to process a sparse collection, you must step through the elements one by one to find the one corresponding to `%BULK_EXCEPTIONS(i).ERROR_INDEX`. If you use the `VALUES OF` clause to process a subset of elements, you must find the element in the index collection whose subscript matches `%BULK_EXCEPTIONS(i).ERROR_INDEX`, and then use that element's value as the subscript to find the erroneous element in the original collection.

If you omit the keywords `SAVE EXCEPTIONS`, execution of the `FORALL` statement stops when an exception is raised. In that case, `SQL%BULK_EXCEPTIONS.COUNT` returns 1, and `SQL%BULK_EXCEPTIONS` contains just one record. If no exception is raised during execution, `SQL%BULK_EXCEPTIONS.COUNT` returns 0.

Example 12–9 Bulk Operation that Continues Despite Exceptions

```
-- create a temporary table for this example
CREATE TABLE emp_temp AS SELECT * FROM employees;

DECLARE
    TYPE empid_tab IS TABLE OF employees.employee_id%TYPE;
    emp_sr empid_tab;
-- create an exception handler for ORA-24381
    errors NUMBER;
    dml_errors EXCEPTION;
    PRAGMA EXCEPTION_INIT(dml_errors, -24381);
BEGIN
    SELECT employee_id
        BULK COLLECT INTO emp_sr FROM emp_temp
        WHERE hire_date < '30-DEC-94';
-- add '_SR' to the job_id of the most senior employees
    FORALL i IN emp_sr.FIRST..emp_sr.LAST SAVE EXCEPTIONS
        UPDATE emp_temp SET job_id = job_id || '_SR'
        WHERE emp_sr(i) = emp_temp.employee_id;
-- If any errors occurred during the FORALL SAVE EXCEPTIONS,
-- a single exception is raised when the statement completes.

EXCEPTION
-- Figure out what failed and why
    WHEN dml_errors THEN
        errors := SQL%BULK_EXCEPTIONS.COUNT;
        DBMS_OUTPUT.PUT_LINE
            ('Number of statements that failed: ' || errors);
        FOR i IN 1..errors LOOP
            DBMS_OUTPUT.PUT_LINE('Error #' || i || ' occurred during ' ||
                'iteration #' || SQL%BULK_EXCEPTIONS(i).ERROR_INDEX);
            DBMS_OUTPUT.PUT_LINE('Error message is ' ||
                SQLERRM(-SQL%BULK_EXCEPTIONS(i).ERROR_CODE));
        END LOOP;
END;
/
DROP TABLE emp_temp;
```

The output from the example is similar to:

```
Number of statements that failed: 2
Error #1 occurred during iteration #7
Error message is ORA-12899: value too large for column
Error #2 occurred during iteration #13
Error message is ORA-12899: value too large for column
```

In [Example 12-9](#), PL/SQL raises predefined exceptions because updated values were too large to insert into the `job_id` column. After the `FORALL` statement, `SQL%BULK_EXCEPTIONS.COUNT` returned 2, and the contents of `SQL%BULK_EXCEPTIONS` were (7,12899) and (13,12899).

To get the Oracle error message (which includes the code), the value of `SQL%BULK_EXCEPTIONS(i).ERROR_CODE` was negated and then passed to the error-reporting function `SQLERRM`, which expects a negative number.

Retrieving Query Results into Collections (BULK COLLECT Clause)

Using the keywords `BULK COLLECT` with a query is a very efficient way to retrieve the result set. Instead of looping through each row, you store the results in one or more collections, in a single operation. You can use these keywords in the `SELECT INTO` and `FETCH INTO` statements, and the `RETURNING INTO` clause.

With the `BULK COLLECT` clause, all the variables in the `INTO` list must be collections. The table columns can hold scalar or composite values, including object types.

[Example 12-10](#) loads two entire database columns into nested tables:

Example 12-10 Retrieving Query Results with BULK COLLECT

```

DECLARE
  TYPE NumTab IS TABLE OF employees.employee_id%TYPE;
  TYPE NameTab IS TABLE OF employees.last_name%TYPE;
  enums NumTab; -- No need to initialize collections
  names NameTab; -- Values will be filled by SELECT INTO
  PROCEDURE print_results IS
  BEGIN
    IF enums.COUNT = 0 THEN
      DBMS_OUTPUT.PUT_LINE('No results!');
    ELSE
      DBMS_OUTPUT.PUT_LINE('Results:');
      FOR i IN enums.FIRST .. enums.LAST
      LOOP
        DBMS_OUTPUT.PUT_LINE
          (' Employee #' || enums(i) || ': '
           names(i));
      END LOOP;
    END IF;
  END;
BEGIN
  -- Retrieve data for employees with Ids greater than 1000
  SELECT employee_id, last_name
     BULK COLLECT INTO enums, names FROM employees
    WHERE employee_id > 1000;
  -- Data was brought into memory by BULK COLLECT
  -- No need to FETCH each row from result set
  print_results();
  -- Retrieve approximately 20% of all rows
  SELECT employee_id, last_name
     BULK COLLECT INTO enums, names FROM employees SAMPLE (20);
  print_results();
END;
/

```

The collections are initialized automatically. Nested tables and associative arrays are extended to hold as many elements as needed. If you use varrays, all the return values

must fit in the varray's declared size. Elements are inserted starting at index 1, overwriting any existing elements.

Because the processing of the `BULK COLLECT INTO` clause is similar to a `FETCH` loop, it does not raise a `NO_DATA_FOUND` exception if no rows match the query. You must check whether the resulting nested table or varray is null, or if the resulting associative array has no elements, as shown in [Example 12–11](#).

To prevent the resulting collections from expanding without limit, you can use the `LIMIT` clause to or pseudocolumn `ROWNUM` to limit the number of rows processed. You can also use the `SAMPLE` clause to retrieve a random sample of rows.

Example 12–11 Using the Pseudocolumn `ROWNUM` to Limit Query Results

```
DECLARE
    TYPE SalList IS TABLE OF employees.salary%TYPE;
    sals SalList;
BEGIN
    -- Limit number of rows to 50
    SELECT salary BULK COLLECT INTO sals
    FROM employees
    WHERE ROWNUM <= 50;
    -- Retrieve ~10% rows from table
    SELECT salary BULK COLLECT INTO sals FROM employees SAMPLE (10);
END;
/
```

You can process very large result sets by fetching a specified number of rows at a time from a cursor, as shown in the following sections.

Topics:

- [Examples of Bulk Fetching from a Cursor](#)
- [Limiting Rows for a Bulk `FETCH` Operation \(`LIMIT` Clause\)](#)
- [Retrieving DML Results Into a Collection \(`RETURNING INTO` Clause\)](#)
- [Using `FORALL` and `BULK COLLECT` Together](#)
- [Using Host Arrays with Bulk Binds](#)

Examples of Bulk Fetching from a Cursor

You can fetch from a cursor into one or more collections as shown in [Example 12–12](#).

Example 12–12 Bulk-Fetching from a Cursor Into One or More Collections

```
DECLARE
    TYPE NameList IS TABLE OF employees.last_name%TYPE;
    TYPE SalList IS TABLE OF employees.salary%TYPE;
    CURSOR c1 IS SELECT last_name, salary
    FROM employees
    WHERE salary > 10000;
    names NameList;
    sals SalList;
    TYPE RecList IS TABLE OF c1%ROWTYPE;
    recs RecList;
    v_limit PLS_INTEGER := 10;
    PROCEDURE print_results IS
    BEGIN
        -- Check if collections are empty
        IF names IS NULL OR names.COUNT = 0 THEN
```



```

        DBMS_OUTPUT.PUT_LINE('No results!');
    ELSE
        DBMS_OUTPUT.PUT_LINE('Results: ');
        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 at once ---');
    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;
/

```

[Example 12–13](#) shows how you can fetch from a cursor into a collection of records.

Example 12–13 Bulk-Fetching from a Cursor Into a Collection of Records

```

DECLARE
    TYPE DeptRecTab IS TABLE OF departments%ROWTYPE;
    dept_rec Tab DeptRecTab;
    CURSOR c1 IS
        SELECT department_id, department_name, manager_id, location_id
        FROM departments
        WHERE department_id > 70;
BEGIN
    OPEN c1;
    FETCH c1 BULK COLLECT INTO dept_rec;
END;
/

```

Limiting Rows for a Bulk FETCH Operation (LIMIT Clause)

The optional `LIMIT` clause, allowed only in bulk `FETCH` statements, limits the number of rows fetched from the database. In [Example 12–14](#), with each iteration of the loop,

the `FETCH` statement fetches ten rows (or less) into index-by table `empids`. The previous values are overwritten. Note the use of `empids.COUNT` to determine when to exit the loop.

Example 12–14 Using `LIMIT` to Control the Number of Rows In a `BULK COLLECT`

```

DECLARE
    TYPE numtab IS TABLE OF NUMBER INDEX BY PLS_INTEGER;
    CURSOR c1 IS SELECT employee_id
        FROM employees
        WHERE department_id = 80;
    empids    numtab;
    rows      PLS_INTEGER := 10;
BEGIN
    OPEN c1;
    -- Fetch 10 rows or less in each iteration
    LOOP
        FETCH c1 BULK COLLECT INTO empids LIMIT rows;
        EXIT WHEN empids.COUNT = 0;
    -- EXIT WHEN c1%NOTFOUND; -- incorrect, can omit some data
        DBMS_OUTPUT.PUT_LINE
            ('----- Results from Each Bulk Fetch -----');
        FOR i IN 1..empids.COUNT LOOP
            DBMS_OUTPUT.PUT_LINE( 'Employee Id: ' || empids(i));
        END LOOP;
    END LOOP;
    CLOSE c1;
END;
/

```

Retrieving DML Results Into a Collection (`RETURNING INTO` Clause)

You can use the `BULK COLLECT` clause in the `RETURNING INTO` clause of an `INSERT`, `UPDATE`, or `DELETE` statement:

Example 12–15 Using `BULK COLLECT` with the `RETURNING INTO` Clause

```

CREATE TABLE emp_temp AS SELECT * FROM employees;
DECLARE
    TYPE NumList IS TABLE OF employees.employee_id%TYPE;
    enums NumList;
    TYPE NameList IS TABLE OF employees.last_name%TYPE;
    names NameList;
BEGIN
    DELETE FROM emp_temp WHERE department_id = 30
        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;
/

```

Using FORALL and BULK COLLECT Together

You can combine the `BULK COLLECT` clause with a `FORALL` statement. The output collections are built up as the `FORALL` statement iterates.

In [Example 12-16](#), the `employee_id` value of each deleted row is stored in the collection `e_ids`. The collection `depts` has 3 elements, so the `FORALL` statement iterates 3 times. If each `DELETE` issued by the `FORALL` statement deletes 5 rows, then the collection `e_ids`, which stores values from the deleted rows, has 15 elements when the statement completes:

Example 12-16 Using FORALL with BULK COLLECT

```
CREATE TABLE emp_temp AS SELECT * FROM employees;
DECLARE
    TYPE NumList IS TABLE OF NUMBER;
    depts NumList := NumList(10,20,30);
    TYPE enum_t IS TABLE OF employees.employee_id%TYPE;
    TYPE dept_t IS TABLE OF employees.department_id%TYPE;
    e_ids enum_t;
    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;
/
```

The column values returned by each execution are added to the values returned previously. If you use a `FOR` loop instead of the `FORALL` statement, the set of returned values is overwritten by each `DELETE` statement.

You cannot use the `SELECT BULK COLLECT` statement in a `FORALL` statement.

Using Host Arrays with Bulk Binds

Client-side programs can use anonymous PL/SQL blocks to bulk-bind input and output host arrays. This is the most efficient way to pass collections to and from the database server.

Host arrays are declared in a host environment such as an OCI or a Pro*C program and must be prefixed with a colon to distinguish them from PL/SQL collections. In the following example, an input host array is used in a `DELETE` statement. At run time, the anonymous PL/SQL block is sent to the database server for execution.

```
DECLARE
BEGIN
    -- Assume that values were assigned to host array
    -- and host variables in host environment
    FORALL i IN :lower..:upper
        DELETE FROM employees
            WHERE department_id = :depts(i);
```

```
    COMMIT;  
END;
```

Writing Computation-Intensive PL/SQL Programs

The `BINARY_FLOAT` and `BINARY_DOUBLE` datatypes make it practical to write PL/SQL programs to do number-crunching, for scientific applications involving floating-point calculations. These datatypes act much like the native floating-point types on many hardware systems, with semantics derived from the IEEE-754 floating-point standard.

The way these datatypes represent decimal data make them less suitable for financial applications, where precise representation of fractional amounts is more important than pure performance.

The `PLS_INTEGER` datatype is a PL/SQL-only datatype that is more efficient than the SQL datatypes `NUMBER` or `INTEGER` for integer arithmetic. You can use `PLS_INTEGER` to write pure PL/SQL code for integer arithmetic, or convert `NUMBER` or `INTEGER` values to `PLS_INTEGER` for manipulation by PL/SQL.

Within a package, you can write overloaded versions of subprograms that accept different numeric parameters. The math routines can be optimized for each kind of parameter (`BINARY_FLOAT`, `BINARY_DOUBLE`, `NUMBER`, `PLS_INTEGER`), avoiding unnecessary conversions.

The built-in math functions such as `SQRT`, `SIN`, `COS`, and so on already have fast overloaded versions that accept `BINARY_FLOAT` and `BINARY_DOUBLE` parameters. You can speed up math-intensive code by passing variables of these types to such functions, and by calling the `TO_BINARY_FLOAT` or `TO_BINARY_DOUBLE` functions when passing expressions to such functions.

Tuning Dynamic SQL with EXECUTE IMMEDIATE Statement and Cursor Variables

Some programs (a general-purpose report writer for example) must build and process a variety of SQL statements, where the exact text of the statement is unknown until run time. Such statements probably change from execution to execution. They are called dynamic SQL statements.

Formerly, to execute dynamic SQL statements, you had to use the supplied package `DBMS_SQL`. Now, within PL/SQL, you can execute any kind of dynamic SQL statement using an interface called native dynamic SQL. The main PL/SQL features involved are the `EXECUTE IMMEDIATE` statement and cursor variables (also known as `REF CURSORS`).

Native dynamic SQL code is more compact and much faster than calling the `DBMS_SQL` package. The following example declares a cursor variable, then associates it with a dynamic `SELECT` statement:

```
DECLARE  
    TYPE EmpCurTyp IS REF CURSOR;  
    emp_cv   EmpCurTyp;  
    v_ename  VARCHAR2(15);  
    v_sal    NUMBER := 1000;  
    table_name VARCHAR2(30) := 'employees';  
BEGIN  
    OPEN emp_cv FOR 'SELECT last_name, salary FROM ' || table_name ||  
        ' WHERE salary > :s' USING v_sal;  
    CLOSE emp_cv;
```

```
END;
/
```

For more information, see [Chapter 7, "Using Dynamic SQL"](#).

Tuning PL/SQL Subprogram Calls with NOCOPY Hint

By default, OUT and IN OUT parameters are passed by value. The values of any IN OUT parameters are copied before the subprogram is executed. During subprogram execution, temporary variables hold the output parameter values. If the subprogram exits normally, these values are copied to the actual parameters. If the subprogram exits with an unhandled exception, the original parameters are unchanged.

When the parameters represent large data structures such as collections, records, and instances of object types, this copying slows down execution and uses up memory. In particular, this overhead applies to each call to an object method: temporary copies are made of all the attributes, so that any changes made by the method are only applied if the method exits normally.

To avoid this overhead, you can specify the NOCOPY hint, which allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference. If the subprogram exits normally, the action is the same as normal. If the subprogram exits early with an exception, the values of OUT and IN OUT parameters (or object attributes) might still change. To use this technique, ensure that the subprogram handles all exceptions.

The following example asks the compiler to pass IN OUT parameter `v_staff` by reference, to avoid copying the varray on entry to and exit from the subprogram:

```
DECLARE
    TYPE Staff IS VARRAY(200) OF Employee;
    PROCEDURE reorganize (v_staff IN OUT NOCOPY Staff) IS ...
```

Example 12–17 loads 25,000 records into a local nested table, which is passed to two local procedures that do nothing. A call to the subprogram that uses NOCOPY takes much less time.

Example 12–17 Using NOCOPY with Parameters

```
DECLARE
    TYPE EmpTabTyp IS TABLE OF employees%ROWTYPE;
    emp_tab EmpTabTyp := EmpTabTyp(NULL); -- initialize
    t1 NUMBER;
    t2 NUMBER;
    t3 NUMBER;
    PROCEDURE get_time (t OUT NUMBER) IS
        BEGIN t := DBMS_UTILITY.get_time; END;
    PROCEDURE do_nothing1 (tab IN OUT EmpTabTyp) IS
        BEGIN
            NULL;
        END;
    PROCEDURE do_nothing2 (tab IN OUT NOCOPY EmpTabTyp) IS
        BEGIN
            NULL;
        END;
BEGIN
    SELECT * INTO emp_tab(1)
        FROM employees
        WHERE employee_id = 100;
    -- Copy element 1 into 2..50000
    emp_tab.EXTEND(49999, 1);
```

```

get_time(t1);
-- Pass IN OUT parameter
do_nothing1(emp_tab);
get_time(t2);
-- Pass IN OUT NOCOPY parameter
do_nothing2(emp_tab);
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;
/

```

Restrictions on NOCOPY Hint

The use of NOCOPY increases the likelihood of parameter aliasing. For more information, see ["Understanding PL/SQL Subprogram Parameter Aliasing"](#) on page 8-27.

Remember, NOCOPY is a hint, not a directive. In the following cases, the PL/SQL compiler ignores the NOCOPY hint and uses the by-value parameter-passing method; no error is generated:

- The actual parameter is an element of an associative array. This restriction does not apply if the parameter is an entire associative array.
- The actual parameter is constrained, such as by scale or NOT NULL. This restriction does not apply to size-constrained character strings. This restriction does not extend to constrained elements or attributes of composite types.
- The actual and formal parameters are records, one or both records were declared using %ROWTYPE or %TYPE, and constraints on corresponding fields in the records differ.
- The actual and formal parameters are records, the actual parameter was declared (implicitly) as the index of a cursor FOR loop, and constraints on corresponding fields in the records differ.
- Passing the actual parameter requires an implicit datatype conversion.
- The subprogram is called through a database link or as an external subprogram.

Compiling PL/SQL Program Units for Native Execution

You can usually speed up PL/SQL program units by compiling them into native code (processor-dependent machine code), which is stored in the SYSTEM tablespace.

You can compile any PL/SQL program unit into native code—subprogram, package spec, package body, anonymous block, type spec, or type body. The program unit can be one that you wrote or one that Oracle supplied.

Natively compiled program units work in all server environments, including shared server configuration (formerly called "multithreaded server") and Oracle Real Application Clusters (Oracle RAC).

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 will speed up both your own code and calls to all of the built-in PL/SQL packages.

Topics:

- [Determining Whether to Use PL/SQL Native Compilation](#)
- [How PL/SQL Native Compilation Works](#)
- [Dependencies, Invalidation, and Revalidation](#)
- [Setting PLSQL_CODE_TYPE Initialization Parameter](#)
- [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 program 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, consider the following in determining whether to compile a PL/SQL program unit for native mode:

- 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 executing SQL.
- When many program units (typically over 15,000) are compiled for native execution, and are simultaneously active, the large amount of shared memory required might affect system performance.

How PL/SQL Native Compilation Works

Without native compilation, the PL/SQL statements in a PL/SQL program unit are compiled into an intermediate form, machine-readable code, which is stored in the database dictionary and interpreted at run time.

With PL/SQL native compilation, the PL/SQL statements in a PL/SQL program unit are compiled into native code and stored in the SYSTEM tablespace. The native code does not have to be interpreted at run time, so it runs faster.

Because native compilation applies only to PL/SQL statements, a PL/SQL program unit that only calls SQL statements might not run faster when natively compiled, but it will run at least as fast as the corresponding interpreted code. The compiled code and the interpreted code make the same library calls, so their action is exactly the same.

The first time a natively compiled PL/SQL program unit is executed, it is fetched from the SYSTEM tablespace into shared memory. Regardless of how many sessions call the program unit, shared memory has only one copy of it. If a program unit is not being used, the shared memory it is using might be freed, to reduce memory load.

Natively compiled subprograms and interpreted subprograms can call each other.

PLSQL native compilation works transparently in a Oracle Real Application Clusters (Oracle RAC) environment.

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 ["Initialization Parameters for PL/SQL Compilation"](#) on page 12-1.

Setting `PLSQL_CODE_TYPE` Initialization Parameter

The `PLSQL_CODE_TYPE` initialization parameter determines whether PL/SQL code is natively compiled or interpreted. The default setting is `INTERPRETED`, which is recommended during development. To enable PL/SQL native compilation, set the value of `PLSQL_CODE_TYPE` to `NATIVE`.

Note: If the optimization level (set by the `PLSQL_OPTIMIZE_LEVEL` initialization parameter) is less than 2, and you set `PLSQL_CODE_TYPE` to `NATIVE`, the compiler ignores your setting and warns you of this.

A package specification and its body can have different `PLSQL_CODE_TYPE` settings.

If you compile the whole database as `NATIVE`, Oracle recommends that you set `PLSQL_CODE_TYPE` at the system level.

See Also: *Oracle Database Reference* for information about the `PLSQL_CODE_TYPE` initialization parameter and the initialization parameter file.

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 initialization parameter `PLSQL_CODE_TYPE` to `NATIVE`. The performance benefits apply to all the built-in PL/SQL packages, which are used for many database operations.

Compiling the Entire Database for PL/SQL Native or Interpreted Compilation

If you have DBA privileges, you can recompile all PL/SQL modules in an existing database to `NATIVE` or `INTERPRETED`, using the `dbmsupgnv.sql` and `dbmsupgin.sql` scripts respectively during the process described in this section.

Before making the conversion, review "[Determining Whether to Use PL/SQL Native Compilation](#)" on page 12-27.

During the conversion to native compilation, `TYPE` specifications are not recompiled by `dbmsupgnav.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 `dbmsupgnav.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 need to recompile all PL/SQL modules to interpreted compilation, make these changes in the steps.

- Skip the first step.
 - Set the `PLSQL_CODE_TYPE` initialization parameter to `INTERPRETED` rather than `NATIVE`.
 - Substitute `dbmsupgin.sql` for the `dbmsupgnav.sql` script.
-
-

1. Ensure that a test PL/SQL unit can be compiled. For example:

```
ALTER PROCEDURE my_proc COMPILE PLSQL_CODE_TYPE=NATIVE REUSE SETTINGS;
```

2. Shut down application services, the listener, and the database.

- Shut down all of the Application services including the Forms Processes, Web Servers, Reports Servers, and Concurrent Manager Servers. After shutting down all of the Application services, ensure that all of the connections to the database were terminated.
- Shut down the TNS listener of the database to ensure that no new connections are made.
- Shut down the database in normal or immediate mode as the user `SYS`. See the *Oracle Database Administrator's Guide*.

3. Set `PLSQL_CODE_TYPE` to `NATIVE` in the initialization parameter file. If the database is using a server parameter file, then set this after the database has started. See "[Setting PLSQL_CODE_TYPE Initialization Parameter](#)" on page 12-28.

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 compilation type that you want.

4. Start up the database in upgrade mode, using the `UPGRADE` option. For information on `SQL*Plus STARTUP`, see the *SQL*Plus User's Guide and Reference*.
5. Execute the following code to list the invalid PL/SQL units. You can save the output of the query for future reference with the `SQL SPOOL` statement:

```
REM 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';
```

REM 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. Execute the following 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/dbmsupgmv.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 `utlrvp.sql` script, Oracle recommends that no other sessions are connected to avoid possible problems. You can ensure this with the following statement:

```
ALTER SYSTEM ENABLE RESTRICTED SESSION;
```

10. Run the `$ORACLE_HOME/rdbms/admin/utlrvp.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 on setting the degree explicitly.

If for any reason the script is abnormally terminated, rerun the `utlrvp.sql` script to recompile any remaining invalid PL/SQL modules.

11. After the compilation completes successfully, verify that there are no new 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. Re-execute the query in step 6. If recompiling with `dbmsupgmv.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 the following statement:

```
ALTER SYSTEM DISABLE RESTRICTED SESSION;
```

Performing Multiple Transformations with Pipelined Table Functions

This section describes how to chain together special kinds of functions known as pipelined table functions. These functions are used in situations such as data warehousing to apply multiple transformations to data.

Topics:

- [Overview of Pipelined Table Functions](#)
- [Writing a Pipelined Table Function](#)
- [Using Pipelined Table Functions for Transformations](#)
- [Returning Results from Pipelined Table Functions](#)
- [Pipelining Data Between PL/SQL Table Functions](#)
- [Optimizing Multiple Calls to Pipelined Table Functions](#)
- [Fetching from Results of Pipelined Table Functions](#)
- [Passing Data with Cursor Variables](#)
- [Performing DML Operations Inside Pipelined Table Functions](#)
- [Performing DML Operations on Pipelined Table Functions](#)
- [Handling Exceptions in Pipelined Table Functions](#)

Overview of Pipelined Table Functions

Pipelined table functions are functions that produce a collection of rows (either a nested table or a varray) that can be queried like a physical database table or assigned to a PL/SQL collection variable. You can use a table function in place of the name of a database table in the `FROM` clause of a query or in place of a column name in the `SELECT` list of a query.

A table function can take a collection of rows as input. An input collection parameter can be either a collection type (such as a `VARRAY` or a PL/SQL table) or a `REF CURSOR`.

Execution of a table function can be parallelized, and returned rows can be streamed directly to the next process without intermediate staging. Rows from a collection returned by a table function can also be pipelined, that is, iteratively returned as they are produced instead of in a batch after all processing of the table function's input is completed.

Streaming, pipelining, and parallel execution of table functions can improve performance:

- By enabling multithreaded, concurrent execution of table functions
- By eliminating intermediate staging between processes
- By improving query response time: With non-pipelined table functions, the entire collection returned by a table function must be constructed and returned to the server before the query can return a single result row. Pipelining enables rows to be returned iteratively, as they are produced. This also reduces the memory that a table function requires, as the object cache does not need to materialize the entire collection.
- By iteratively providing result rows from the collection returned by a table function as the rows are produced instead of waiting until the entire collection is staged in tables or memory and then returning the entire collection.

Writing a Pipelined Table Function

You declare a pipelined table function by specifying the `PIPELINED` keyword. Pipelined functions can be defined at the schema level with `CREATE FUNCTION` or in a package. The `PIPELINED` keyword indicates that the function returns rows iteratively. The return type of the pipelined table function must be a supported collection type, such as a nested table or a varray. This collection type can be declared at the schema level or inside a package. Inside the function, you return individual elements of the collection type. The elements of the collection type must be supported SQL datatypes, such as `NUMBER` and `VARCHAR2`. PL/SQL datatypes, such as `PLS_INTEGER` and `BOOLEAN`, are not supported as collection elements in a pipelined function.

[Example 12–18](#) shows how to assign the result of a pipelined table function to a PL/SQL collection variable and use the function in a `SELECT` statement.

Example 12–18 Assigning the Result of a Table Function

```
CREATE 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;
END pkg1;
/

-- pipelined function is used in FROM clause of SELECT statement
SELECT * FROM TABLE(pkg1.f1(5));
```

Using Pipelined Table Functions for Transformations

A pipelined table function can accept any argument that regular functions accept. A table function that accepts a `REF CURSOR` as an argument can serve as a transformation function. That is, it can use the `REF CURSOR` to fetch the input rows, perform some transformation on them, and then pipeline the results out.

In [Example 12–19](#), the `f_trans` function converts a row of the `employees` table into two rows.

Example 12–19 Using a Pipelined Table Function For a Transformation

```
-- Define the ref cursor types and function
CREATE OR REPLACE PACKAGE refcur_pkg IS
  TYPE refcur_t IS REF CURSOR RETURN employees%ROWTYPE;
  TYPE outrec_typ IS RECORD (
    var_num    NUMBER(6),
    var_char1  VARCHAR2(30),
    var_char2  VARCHAR2(30));
  TYPE outrecset IS TABLE OF outrec_typ;
  FUNCTION f_trans(p refcur_t)
    RETURN outrecset PIPELINED;
```

```

END refcur_pkg;
/

CREATE OR REPLACE PACKAGE BODY refcur_pkg IS
  FUNCTION f_trans(p refcur_t)
    RETURN outrecset PIPELINED IS
    out_rec outrec_typ;
    in_rec p%ROWTYPE;
  BEGIN
  LOOP
    FETCH p INTO in_rec;
    EXIT WHEN p%NOTFOUND;
    -- first row
    out_rec.var_num := in_rec.employee_id;
    out_rec.var_char1 := in_rec.first_name;
    out_rec.var_char2 := in_rec.last_name;
    PIPE ROW(out_rec);
    -- second row
    out_rec.var_char1 := in_rec.email;
    out_rec.var_char2 := in_rec.phone_number;
    PIPE ROW(out_rec);
  END LOOP;
  CLOSE p;
  RETURN;
  END;
END refcur_pkg;
/

-- SELECT query using the f_transc table function
SELECT * FROM TABLE(
  refcur_pkg.f_trans(CURSOR
    (SELECT * FROM employees WHERE department_id = 60));

```

In the preceding query, the pipelined table function `f_trans` fetches rows from the CURSOR subquery `SELECT * FROM employees ...`, performs the transformation, and pipelines the results back to the user as a table. The function produces two output rows (collection elements) for each input row.

Note that when a CURSOR subquery is passed from SQL to a REF CURSOR function argument as in [Example 12-19](#), the referenced cursor is already open when the function begins executing.

Returning Results from Pipelined Table Functions

In PL/SQL, the `PIPE ROW` statement causes a pipelined table function to pipe a row and continue processing. The statement enables a PL/SQL table function to return rows as soon as they are produced. For performance, the PL/SQL run-time system provides the rows to the consumer in batches.

In [Example 12-19](#), the `PIPE ROW(out_rec)` statement pipelines data out of the PL/SQL table function. `out_rec` is a record, and its type matches the type of an element of the output collection.

The `PIPE ROW` statement may be used only in the body of pipelined table functions; an error is raised if it is used anywhere else. The `PIPE ROW` statement can be omitted for a pipelined table function that returns no rows.

A pipelined table function may have a `RETURN` statement that does not return a value. The `RETURN` statement transfers the control back to the consumer and ensures that the next fetch gets a `NO_DATA_FOUND` exception.

Because table functions pass control back and forth to a calling routine as rows are produced, there is a restriction on combining table functions and `PRAGMA AUTONOMOUS_TRANSACTION`. If a table function is part of an autonomous transaction, it must `COMMIT` or `ROLLBACK` before each `PIPE ROW` statement, to avoid an error in the calling subprogram.

Oracle has three special SQL datatypes that enable you to dynamically encapsulate and access type descriptions, data instances, and sets of data instances of any other SQL type, including object and collection types. You can also use these three special types to create anonymous (that is, unnamed) types, including anonymous collection types. The types are `SYS.ANYTYPE`, `SYS.ANYDATA`, and `SYS.ANYDATASET`. The `SYS.ANYDATA` type can be useful in some situations as a return value from table functions.

See Also: *Oracle Database PL/SQL Packages and Types Reference* for information about the interfaces to the `ANYTYPE`, `ANYDATA`, and `ANYDATASET` types and about the `DBMS_TYPES` package for use with these types.

Pipelining Data Between PL/SQL Table Functions

With serial execution, results are pipelined from one PL/SQL table function to another using an approach similar to co-routine execution. For example, the following statement pipelines results from function `g` to function `f`:

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM TABLE(g()))));
```

Parallel execution works similarly except that each function executes in a different process (or set of processes).

Optimizing Multiple Calls to Pipelined Table Functions

Multiple calls to a pipelined table function, either within the same query or in separate queries result in multiple executions of the underlying implementation. By default, there is no buffering or reuse of rows. For example:

```
SELECT * FROM TABLE(f(...)) t1, TABLE(f(...)) t2
  WHERE t1.id = t2.id;
SELECT * FROM TABLE(f());
SELECT * FROM TABLE(f());
```

If the function always produces the same result value for each combination of values passed in, you can declare the function `DETERMINISTIC`, and Oracle automatically buffers rows for it. If the function is not really deterministic, results are unpredictable.

Fetching from Results of Pipelined Table Functions

PL/SQL cursors and ref cursors can be defined for queries over table functions. For example:

```
OPEN c FOR SELECT * FROM TABLE(f(...));
```

Cursors over table functions have the same fetch semantics as ordinary cursors. `REF CURSOR` assignments based on table functions do not have any special semantics.

However, the SQL optimizer will not optimize across PL/SQL statements. For example:

```
DECLARE
  r SYS_REFCURSOR;
```

```

BEGIN
  OPEN r FOR SELECT *
    FROM TABLE(f(CURSOR(SELECT * FROM tab)));
  SELECT * BULK COLLECT INTO rec_tab FROM TABLE(g(r));
END;
/

```

does not execute as well as:

```

SELECT * FROM TABLE(g(CURSOR(SELECT * FROM
  TABLE(f(CURSOR(SELECT * FROM tab))))));

```

This is so even ignoring the overhead associated with executing two SQL statements and assuming that the results can be pipelined between the two statements.

Passing Data with Cursor Variables

You can pass a set of rows to a PL/SQL function in a REF CURSOR parameter. For example, this function is declared to accept an argument of the predefined weakly typed REF CURSOR type SYS_REFCURSOR:

```

FUNCTION f(p1 IN SYS_REFCURSOR) RETURN ... ;

```

Results of a subquery can be passed to a function directly:

```

SELECT * FROM TABLE(f(CURSOR(SELECT empid FROM tab)));

```

In the preceding example, the CURSOR keyword causes the results of a subquery to be passed as a REF CURSOR parameter.

A predefined weak REF CURSOR type SYS_REFCURSOR is also supported. With SYS_REFCURSOR, you do not need to first create a REF CURSOR type in a package before you can use it.

To use a strong REF CURSOR type, you still must create a PL/SQL package and declare a strong REF CURSOR type in it. Also, if you are using a strong REF CURSOR type as an argument to a table function, then the actual type of the REF CURSOR argument must match the column type, or an error is generated. Weak REF CURSOR arguments to table functions can only be partitioned using the PARTITION BY ANY clause. You cannot use range or hash partitioning for weak REF CURSOR arguments.

PL/SQL functions can accept multiple REF CURSOR input variables as shown in [Example 12-20](#).

For more information about cursor variables, see "[Declaring REF CURSOR Types and Cursor Variables](#)" on page 6-23.

Example 12-20 Using Multiple REF CURSOR Input Variables

```

-- Define the ref cursor types
CREATE 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;
END refcur_pkg;
/

-- SELECT query using the g_trans table function
SELECT * FROM TABLE(refcur_pkg.g_trans(
    CURSOR(SELECT * FROM employees WHERE department_id = 60),
    CURSOR(SELECT * FROM departments WHERE department_id = 60)));
    
```

You can pass table function return values to other table functions by creating a REF CURSOR that iterates over the returned data:

```
SELECT * FROM TABLE(f(CURSOR(SELECT * FROM TABLE(g(...))));
```

You can explicitly open a REF CURSOR for a query and pass it as a parameter to a table function:

```

DECLARE
    r SYS_REFCURSOR;
    rec ...;
BEGIN
    OPEN r FOR SELECT * FROM TABLE(f(...));
    -- Must return a single row result set.
    SELECT * INTO rec FROM TABLE(g(r));
END;
/
    
```

In this case, the table function closes the cursor when it completes, so your program must not explicitly try to close the cursor.

A table function can compute aggregate results using the input ref cursor.

[Example 12–21](#) computes a weighted average by iterating over a set of input rows.

Example 12–21 Using a Pipelined Table Function as an Aggregate Function

```
CREATE TABLE gradereport (student VARCHAR2(30),
                          subject VARCHAR2(30),
                          weight NUMBER, grade NUMBER);
INSERT INTO gradereport VALUES('Mark', 'Physics', 4, 4);
INSERT INTO gradereport VALUES('Mark', 'Chemistry', 4, 3);
INSERT INTO gradereport VALUES('Mark', 'Maths', 3, 3);
INSERT INTO gradereport VALUES('Mark', 'Economics', 3, 4);

CREATE PACKAGE pkg_gpa IS
  TYPE gpa IS TABLE OF NUMBER;
  FUNCTION weighted_average(input_values SYS_REFCURSOR)
    RETURN gpa PIPELINED;
END pkg_gpa;
/

CREATE PACKAGE BODY pkg_gpa IS
FUNCTION weighted_average(input_values SYS_REFCURSOR)
  RETURN gpa PIPELINED IS
  grade NUMBER;
  total NUMBER := 0;
  total_weight NUMBER := 0;
  weight NUMBER := 0;
BEGIN
-- Function accepts ref cursor and loops through all input rows
  LOOP
    FETCH input_values INTO weight, grade;
    EXIT WHEN input_values%NOTFOUND;
-- Accumulate the weighted average
    total_weight := total_weight + weight;
    total := total + grade*weight;
  END LOOP;
  PIPE ROW (total / total_weight);
  RETURN; -- the function returns a single result
END;
END pkg_gpa;
/

-- Query result is a nested table with single row
-- COLUMN_VALUE is keyword that returns contents of nested table
SELECT w.column_value "weighted result" FROM TABLE(
  pkg_gpa.weighted_average(CURSOR(SELECT weight,
    grade FROM gradereport))) w;
```

Performing DML Operations Inside Pipelined Table Functions

To execute DML statements, declare a pipelined table function with the `AUTONOMOUS_TRANSACTION` pragma, which causes the function to execute in a new transaction not shared by other processes:

```
CREATE FUNCTION f(p SYS_REFCURSOR)
  RETURN CollType PIPELINED IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  NULL;
END;
/
```

During parallel execution, each instance of the table function creates an independent transaction.

Performing DML Operations on Pipelined Table Functions

Pipelined table functions cannot be the target table in `UPDATE`, `INSERT`, or `DELETE` statements. For example, the following statements will raise an error:

```
UPDATE F(CURSOR(SELECT * FROM tab)) SET col = value;
INSERT INTO f(...) VALUES ('any', 'thing');
```

However, you can create a view over a table function and use `INSTEAD OF` triggers to update it. For example:

```
CREATE VIEW BookTable AS SELECT x.Name, x.Author
FROM TABLE(GetBooks('data.txt')) x;
```

The following `INSTEAD OF` trigger fires when the user inserts a row into the `BookTable` view:

```
CREATE TRIGGER BookTable_insert
INSTEAD OF INSERT ON BookTable
REFERENCING NEW AS n
FOR EACH ROW
BEGIN
    ...
END
/
INSERT INTO BookTable VALUES (...);
```

`INSTEAD OF` triggers can be defined for all DML operations on a view built on a table function.

Handling Exceptions in Pipelined Table Functions

Exception handling in pipelined table functions works just as it does with regular functions.

Some languages, such as C and Java, provide a mechanism for user-supplied exception handling. If an exception raised within a table function is handled, the table function executes the exception handler and continues processing. Exiting the exception handler takes control to the enclosing scope. If the exception is cleared, execution proceeds normally.

An unhandled exception in a table function causes the parent transaction to roll back.

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 on how to read syntax diagrams, see *Oracle Database SQL Language Reference*.

Topics:

- [Assignment Statement](#)
- [AUTONOMOUS_TRANSACTION Pragma](#)
- [Block Declaration](#)
- [CASE Statement](#)
- [CLOSE Statement](#)
- [Collection Definition](#)
- [Collection Methods](#)
- [Comments](#)
- [COMMIT Statement](#)
- [Constant and Variable Declaration](#)
- [CONTINUE Statement](#)
- [Cursor Attributes](#)
- [Cursor Variables](#)
- [Cursor Declaration](#)
- [DELETE Statement](#)
- [EXCEPTION_INIT Pragma](#)
- [Exception Definition](#)
- [EXECUTE IMMEDIATE Statement](#)
- [EXIT Statement](#)
- [Expression Definition](#)
- [FETCH Statement](#)
- [FORALL Statement](#)
- [Function Declaration and Definition](#)
- [GOTO Statement](#)

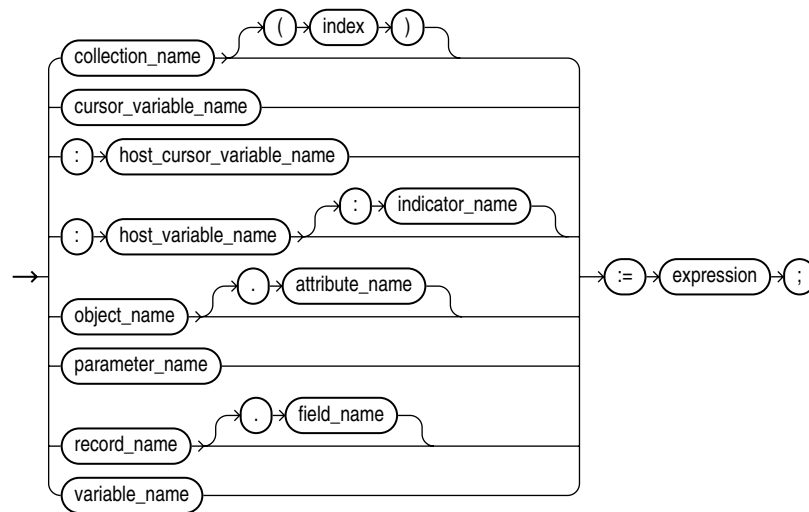
-
- IF Statement
 - INLINE Pragma
 - INSERT Statement
 - Literal Declaration
 - LOCK TABLE Statement
 - LOOP Statements
 - MERGE Statement
 - NULL Statement
 - Object Type Declaration
 - OPEN Statement
 - OPEN-FOR Statement
 - Package Declaration
 - Procedure Declaration and Definition
 - RAISE Statement
 - Record Definition
 - RESTRICT_REFERENCES Pragma (deprecated)
 - RETURN Statement
 - RETURNING INTO Clause
 - ROLLBACK Statement
 - %ROWTYPE Attribute
 - SAVEPOINT Statement
 - SELECT INTO Statement
 - SERIALLY_REUSABLE Pragma
 - SET TRANSACTION Statement
 - SQL Cursor
 - SQLCODE Function
 - SQLERRM Function
 - %TYPE Attribute
 - UPDATE Statement

Assignment Statement

An assignment statement sets the current value of a variable, field, parameter, or element. The statement consists of an assignment target followed by the assignment operator and an expression. When the statement is executed, the expression is evaluated and the resulting value is stored in the target.

Syntax

assignment_statement ::=



Keyword and Parameter Description

attribute_name

An attribute of an object type. The name must be unique within the object type (but can be reused in other object types). You cannot initialize an attribute in its declaration using the assignment operator or `DEFAULT` clause. Also, you cannot impose the `NOT NULL` constraint on an attribute.

collection_name

A nested table, index-by table, or varray previously declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable previously declared within the current scope. Only the value of another cursor variable can be assigned to a cursor variable.

expression

A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. For the syntax of `expression`, see ["Expression Definition"](#) on page 13-57. When the assignment statement is executed, the expression is evaluated and the resulting value is stored in the assignment target. The value and target must have compatible datatypes.

field_name

A field in a user-defined or %ROWTYPE record.

host_cursor_variable_name

A cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

host_variable_name

A variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. Host variables must be prefixed with a colon.

index

A numeric expression that must return a value of type PLS_INTEGER or a value implicitly convertible to that datatype.

indicator_name

An indicator variable declared in a PL/SQL host environment and passed to PL/SQL. Indicator variables must be prefixed with a colon. An indicator variable indicates the value or condition of its associated host variable. For example, in the Oracle Precompiler environment, indicator variables let you detect nulls or truncated values in output host variables.

object_name

An instance of an object type previously declared within the current scope.

parameter_name

A formal OUT or IN OUT parameter of the subprogram in which the assignment statement appears.

record_name

A user-defined or %ROWTYPE record previously declared within the current scope.

variable_name

A PL/SQL variable previously declared within the current scope.

Usage Notes

By default, unless a variable is initialized in its declaration, it is initialized to NULL every time a block or subprogram is entered. Always assign a value to a variable before using that variable in an expression.

You cannot assign nulls to a variable defined as NOT NULL. If you try, PL/SQL raises the predefined exception VALUE_ERROR. Only the values TRUE, FALSE, and NULL can be assigned to a Boolean variable. You can assign the result of a comparison or other test to a Boolean variable.

You can assign the value of an expression to a specific field in a record. You can assign values to all fields in a record at once. PL/SQL allows aggregate assignment between entire records if their declarations refer to the same cursor or table. [Example 1-3, "Assigning Values to Variables with the Assignment Operator"](#) on page 1-6 shows how to copy values from all the fields of one record to another:

You can assign the value of an expression to a specific element in a collection, by subscripting the collection name.

Examples

[Example 1–3, "Assigning Values to Variables with the Assignment Operator"](#) on page 1-6

[Example 1–4, "Using SELECT INTO to Assign Values to Variables"](#) on page 1-6

[Example 1–5, "Assigning Values to Variables as Parameters of a Subprogram"](#) on page 1-7

[Example 2–10, "Assigning Values to a Record with a %ROWTYPE Declaration"](#) on page 2-13

Related Topics

["Assigning Values to Variables"](#) on page 2-19

["Constant and Variable Declaration"](#) on page 13-29

["Expression Definition"](#) on page 13-57

["SELECT INTO Statement"](#) on page 13-128

AUTONOMOUS_TRANSACTION Pragma

The `AUTONOMOUS_TRANSACTION` pragma marks a routine as autonomous. When an autonomous routine is invoked, the main transaction is suspended. Without affecting the main transaction, the autonomous routine can commit or roll back its own operations.

In this context, a routine is either a top-level (not nested) anonymous PL/SQL block or a PL/SQL subprogram. For more information, see ["Doing Independent Units of Work with Autonomous Transactions"](#) on page 6-40.

Syntax

autonomous_transaction_pragma ::=

→ `PRAGMA` → `AUTONOMOUS_TRANSACTION` → `(;)`

Keyword and Parameter Description

PRAGMA

Signifies that the statement is a pragma (compiler directive). Pragmas are processed at compile time, not at run time. They pass information to the compiler.

Usage Notes

You can apply this pragma to:

- Top-level (not nested) anonymous PL/SQL blocks
- Local, standalone, and packaged functions and procedures
- Methods of a SQL object type
- Database triggers

You cannot apply this pragma to an entire package or an entire an object type. Instead, you can apply the pragma to each packaged subprogram or object method.

You can code the pragma anywhere in the declarative section. For readability, code the pragma at the top of the section.

Once started, an autonomous transaction is fully independent. It shares no locks, resources, or commit-dependencies with the main transaction. You can log events, increment retry counters, and so on, even if the main transaction rolls back.

Unlike regular triggers, autonomous triggers can contain transaction control statements such as `COMMIT` and `ROLLBACK`, and can issue DDL statements (such as `CREATE` and `DROP`) through the `EXECUTE IMMEDIATE` statement.

Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits. The changes also become visible to the main transaction when it resumes, but only 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.

In the main transaction, rolling back to a savepoint located before the call to the autonomous subprogram does *not* roll back the autonomous transaction. Remember, autonomous transactions are fully independent of the main transaction.

If an autonomous transaction attempts to access a resource held by the main transaction (which cannot resume until the autonomous routine exits), a deadlock can occur. Oracle raises an exception in the autonomous transaction, which is rolled back if the exception goes unhandled.

If you try to exit an active autonomous transaction without committing or rolling back, Oracle raises an exception. If the exception goes unhandled, or if the transaction ends because of some other unhandled exception, the transaction is rolled back.

Examples

[Example 6-43, "Declaring an Autonomous Function in a Package"](#) on page 6-42

[Example 6-44, "Declaring an Autonomous Standalone Procedure"](#) on page 6-42

[Example 6-45, "Declaring an Autonomous PL/SQL Block"](#) on page 6-42

[Example 6-46, "Declaring an Autonomous Trigger"](#) on page 6-43

Related Topics

["Doing Independent Units of Work with Autonomous Transactions"](#) on page 6-40

["EXCEPTION_INIT Pragma"](#) on page 13-47

["INLINE Pragma"](#) on page 13-82

["RESTRICT_REFERENCES Pragma"](#) on page 13-118

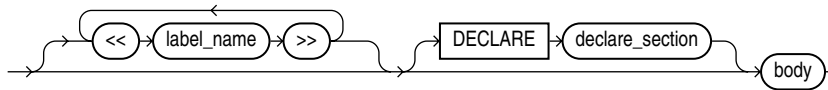
["SERIALLY_REUSABLE Pragma"](#) on page 13-132

Block Declaration

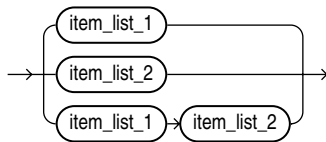
The basic program unit in PL/SQL is the block. A PL/SQL block is defined by the keywords `DECLARE`, `BEGIN`, `EXCEPTION`, and `END`. These keywords partition the block into a declarative part, an executable part, and an exception-handling part. Only the executable part is required. You can nest a block within another block wherever you can place an executable statement. For more information, see ["Understanding PL/SQL Block Structure"](#) on page 1-5 and ["Scope and Visibility of PL/SQL Identifiers"](#) on page 2-16.

Syntax

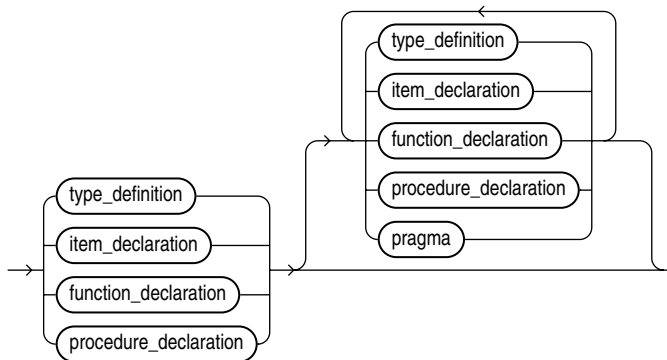
plsql_block ::=



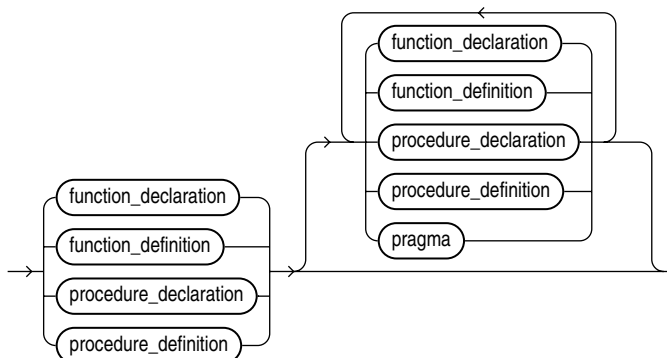
declare_section ::=

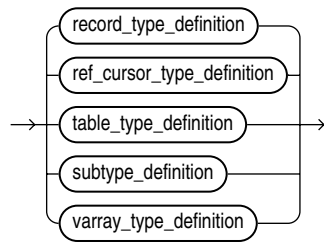
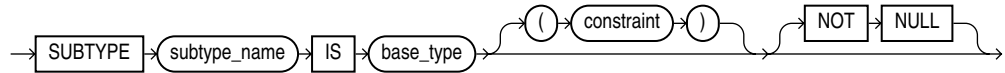
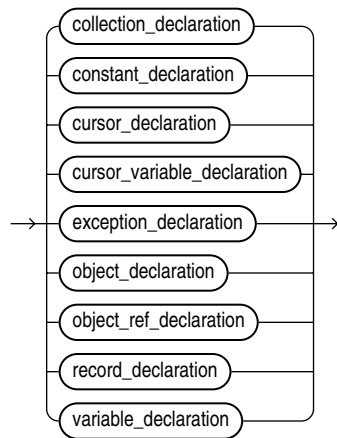
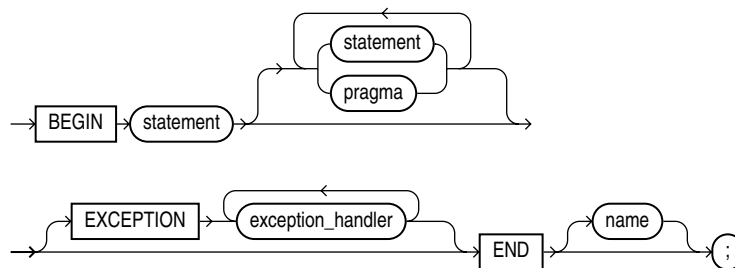


item_list_1 ::=

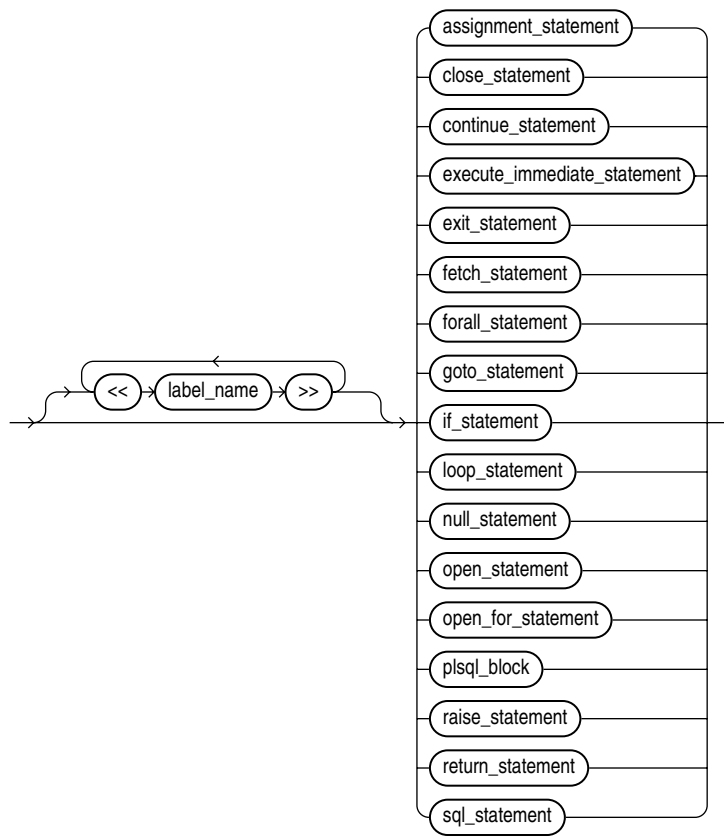


item_list_2 ::=

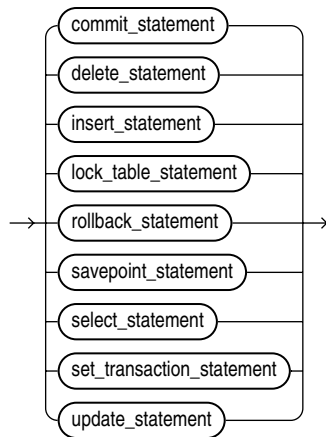


type_definition ::=**subtype_definition ::=****item_declaration ::=****body ::=**

statement ::=



sql_statement ::=



Keyword and Parameter Description

base_type

Any scalar or user-defined PL/SQL datatype specifier such as CHAR, DATE, or RECORD.

BEGIN

Signals the start of the executable part of a PL/SQL block, which contains executable statements. A PL/SQL block must contain at least one executable statement (even just the `NULL` statement). See ["Understanding PL/SQL Block Structure"](#) on page 1-5.

collection_declaration

Declares a collection (index-by table, nested table, or varray). For the syntax of `collection_declaration`, see ["Collection Definition"](#) on page 13-19.

constant_declaration

Declares a constant. For the syntax of `constant_declaration`, see ["Constant and Variable Declaration"](#) on page 13-29.

constraint

Applies only to datatypes that can be constrained such as `CHAR` and `NUMBER`. For character datatypes, this specifies a maximum size in bytes. For numeric datatypes, this specifies a maximum precision and scale.

cursor_declaration

Declares an explicit cursor. For the syntax of `cursor_declaration`, see ["Cursor Declaration"](#) on page 13-41.

cursor_variable_declaration

Declares a cursor variable. For the syntax of `cursor_variable_declaration`, see ["Cursor Variables"](#) on page 13-37.

DECLARE

Signals the start of the declarative part of a PL/SQL block, which contains local declarations. Items declared locally exist only within the current block and all its sub-blocks and are not visible to enclosing blocks. The declarative part of a PL/SQL block is optional. It is terminated implicitly by the keyword `BEGIN`, which introduces the executable part of the block. For more information, see ["Declarations"](#) on page 2-9.

PL/SQL does not allow forward references. You must declare an item before referencing it in any other statements. Also, you must declare subprograms at the end of a declarative section after all other program items.

END

Signals the end of a PL/SQL block. It must be the last keyword in a block. Remember, `END` does not signal the end of a transaction. Just as a block can span multiple transactions, a transaction can span multiple blocks. See ["Understanding PL/SQL Block Structure"](#) on page 1-5.

EXCEPTION

Signals the start of the exception-handling part of a PL/SQL block. When an exception is raised, normal execution of the block stops and control transfers to the appropriate exception handler. After the exception handler completes, execution proceeds with the statement following the block. See ["Understanding PL/SQL Block Structure"](#) on page 1-5.

If there is no exception handler for the raised exception in the current block, control passes to the enclosing block. This process repeats until an exception handler is found or there are no more enclosing blocks. If PL/SQL can find no exception handler for the

exception, execution stops and an unhandled exception error is returned to the host environment. For more information on exceptions, see [Chapter 11, "Handling PL/SQL Errors"](#).

exception_declaration

Declares an exception. For the syntax of `exception_declaration`, see ["Exception Definition"](#) on page 13-48.

exception_handler

Associates an exception with a sequence of statements, which is executed when that exception is raised. For the syntax of `exception_handler`, see ["Exception Definition"](#) on page 13-48.

function_declaration

Declares a function. See ["Function Declaration and Definition"](#) on page 13-74.

label_name

An undeclared identifier that optionally labels a PL/SQL block or statement. If used, `label_name` must be enclosed by double angle brackets and must appear at the beginning of the block or statement which it labels. Optionally, when used to label a block, the `label_name` can also appear at the end of the block without the angle brackets. Multiple labels are allowed for a block or statement, but they must be unique for each block or statement.

A global identifier declared in an enclosing block can be redeclared in a sub-block, in which case the local declaration prevails and the sub-block cannot reference the global identifier unless you use a block label to qualify the reference. See [Example 2-19, "PL/SQL Block Using Multiple and Duplicate Labels"](#) on page 2-19.

name

Is the label name (without the delimiters << and >>).

object_declaration

Declares an instance of an object type. For the syntax of `object_declaration`, see ["Object Type Declaration"](#) on page 13-100.

pragma

One of the following:

- ["AUTONOMOUS_TRANSACTION Pragma"](#) on page 13-6
- ["EXCEPTION_INIT Pragma"](#) on page 13-47
- ["INLINE Pragma"](#) on page 13-82
- ["RESTRICT_REFERENCES Pragma"](#) on page 13-118
- ["SERIALLY_REUSABLE Pragma"](#) on page 13-132

procedure_declaration

Declare a procedure. See ["Procedure Declaration and Definition"](#) on page 13-110.

record_declaration

Declares a user-defined record. For the syntax of `record_declaration`, see ["Record Definition"](#) on page 13-115.

statement

An executable (not declarative) statement that. A sequence of statements can include procedural statements such as `RAISE`, SQL statements such as `UPDATE`, and PL/SQL blocks. PL/SQL statements are free format. That is, they can continue from line to line if you do not split keywords, delimiters, or literals across lines. A semicolon (;) serves as the statement terminator.

subtype_name

A user-defined subtype that was defined using any scalar or user-defined PL/SQL datatype specifier such as `CHAR`, `DATE`, or `RECORD`.

variable_declaration

Declares a variable. For the syntax of `variable_declaration`, see ["Constant and Variable Declaration"](#) on page 13-29.

PL/SQL supports a subset of SQL statements that includes data manipulation, cursor control, and transaction control statements but excludes data definition and data control statements such as `ALTER`, `CREATE`, `GRANT`, and `REVOKE`.

Examples

[Example 1–5, "Assigning Values to Variables as Parameters of a Subprogram"](#) on page 1-7

[Example 2–19, "PL/SQL Block Using Multiple and Duplicate Labels"](#) on page 2-19

Related Topics

["Constant and Variable Declaration"](#) on page 13-29

["Exception Definition"](#) on page 13-48

["Function Declaration and Definition"](#) on page 13-74

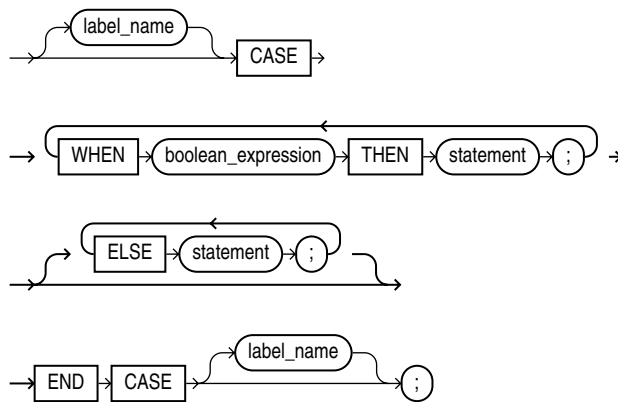
["Procedure Declaration and Definition"](#) on page 13-110

CASE Statement

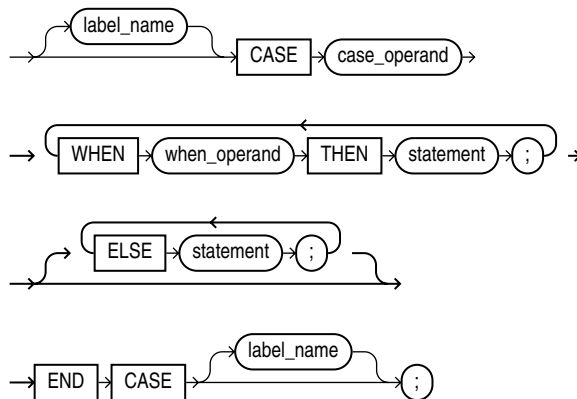
The CASE statement chooses from a sequence of conditions, and executes a corresponding statement. The CASE statement evaluates a single expression and compares it against several potential values, or evaluates multiple Boolean expressions and chooses the first one that is TRUE.

Syntax

searched_case_statement ::=



simple_case_statement ::=



Keyword and Parameter Description

The value of the CASE operand and WHEN operands in a simple CASE statement can be any PL/SQL type other than BLOB, BFILE, an object type, a PL/SQL record, an index-by table, a varray, or a nested table.

If the ELSE clause is omitted, the system substitutes a default action. For a CASE statement, the default when none of the conditions matches is to raise a CASE_NOT_FOUND exception. For a CASE expression, the default is to return NULL.

Usage Notes

The WHEN clauses are executed in order. Each WHEN clause is executed only once. After a matching WHEN clause is found, subsequent WHEN clauses are not executed. You can

use multiple statements after a `WHEN` clause, and that the expression in the `WHEN` clause can be a literal, variable, function call, or any other kind of expression. The `WHEN` clauses can use different conditions rather than all testing the same variable or using the same operator.

The statements in a `WHEN` clause can modify the database and invoke nondeterministic functions. There is no fall-through mechanism as in the `C switch` statement. Once a `WHEN` clause is matched and its statements are executed, the `CASE` statement ends.

The `CASE` statement is appropriate when there is some different action to be taken for each alternative. If you just need to choose among several values to assign to a variable, you can code an assignment statement using a `CASE` expression instead.

You can include `CASE` expressions inside SQL queries, for example instead of a call to the `DECODE` function or some other function that translates from one value to another.

Examples

[Example 1–8, "Using the IF-THEN_ELSE and CASE Statement for Conditional Control"](#) on page 1-10

[Example 4–6, "Using the CASE-WHEN Statement"](#) on page 4-5

[Example 4–7, "Using the Searched CASE Statement"](#) on page 4-6

Related Topics

["Testing Conditions \(IF and CASE Statements\)"](#) on page 4-2

["CASE Expressions"](#) on page 2-28

["Using the Simple CASE Statement"](#) on page 4-4

See Also:

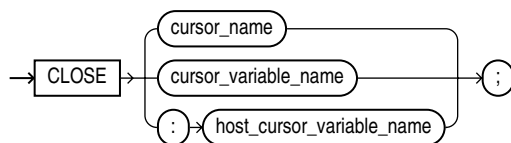
- [Oracle Database SQL Language Reference](#) for information about the `NULLIF` function
- [Oracle Database SQL Language Reference](#) for information about the `COALESCE` function

CLOSE Statement

The `CLOSE` statement indicates that you are finished fetching from a cursor or cursor variable, and that the resources held by the cursor can be reused.

Syntax

close_statement ::=



Keyword and Parameter Description

cursor_name, cursor_variable_name, host_cursor_variable_name

When you close the cursor, you can specify an explicit cursor or a PL/SQL cursor variable, previously declared within the current scope and currently open.

You can also specify a cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

Usage Notes

Once a cursor or cursor variable is closed, you can reopen it using the `OPEN` or `OPEN-FOR` statement, respectively. You must close a cursor before opening it again, otherwise PL/SQL raises the predefined exception `CURSOR_ALREADY_OPEN`. You do not need to close a cursor variable before opening it again.

If you try to close an already-closed or never-opened cursor or cursor variable, PL/SQL raises the predefined exception `INVALID_CURSOR`.

Examples

[Example 4-20, "Using EXIT in a LOOP"](#) on page 4-15

[Example 6-10, "Fetching with a Cursor"](#) on page 6-11

[Example 6-13, "Fetching Bulk Data with a Cursor"](#) on page 6-12

Related Topics

["Closing a Cursor"](#) on page 6-13

["FETCH Statement"](#) on page 13-67

["OPEN Statement"](#) on page 13-102

["OPEN-FOR Statement"](#) on page 13-104

["Querying Data with PL/SQL"](#) on page 6-16

Collection Definition

A collection is an ordered group of elements, all of the same type. For example, the grades for a class of students. Each element has a unique subscript that determines its position in the collection. PL/SQL offers three kinds of collections: associative arrays, nested tables, and varrays (short for variable-size arrays). Nested tables extend the functionality of associative arrays (formerly called PL/SQL tables or index-by tables).

Collections work like the arrays found in most third-generation programming languages. Collections can have only one dimension. Most collections are indexed by integers, although associative arrays can also be indexed by strings. To model multi-dimensional arrays, you can declare collections whose items are other collections.

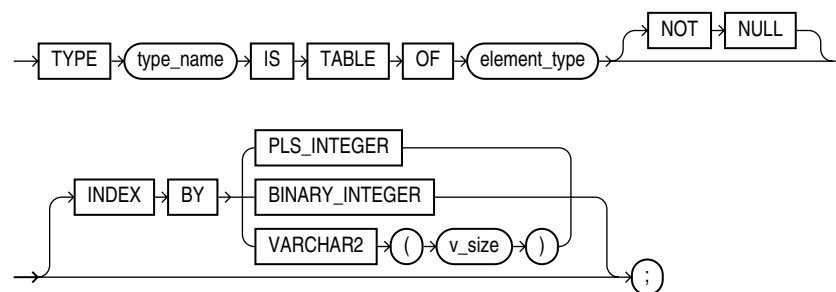
Nested tables and varrays can store instances of an object type and, conversely, can be attributes of an object type. Collections can also be passed as parameters. You can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

For more information, see "Defining Collection Types" on page 5-7.

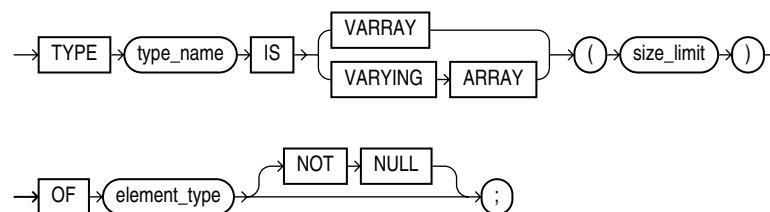
Note: Schema level collection types created with the `CREATE TYPE` statement have a different syntax than PL/SQL collection types. For information on the `CREATE TYPE` SQL statement, see *Oracle Database SQL Language Reference*. For information on the `CREATE TYPE BODY` SQL statement, see *Oracle Database SQL Language Reference*.

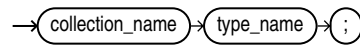
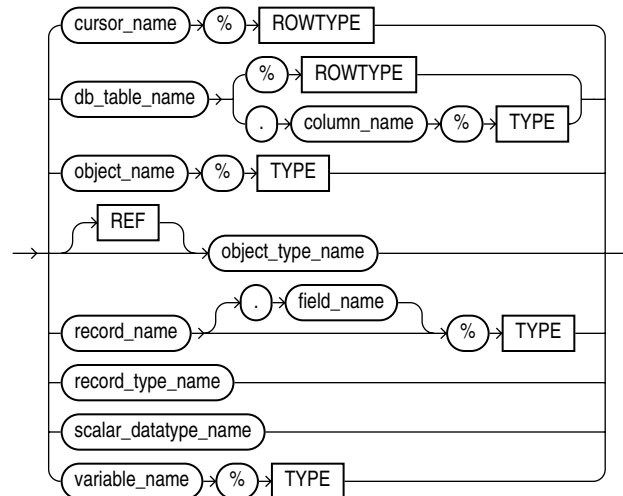
Syntax

table_type_definition ::=



varray_type_definition ::=



collection_type_definition ::=**element_type_definition ::=****Keyword and Parameter Description****element_type**

The type of PL/SQL collection element. The type can be any PL/SQL datatype except REF CURSOR.

INDEX BY type_name

Optional. Defines an associative array, where you specify the subscript values to use rather than the system defining them in sequence.

`type_name` can be `BINARY_INTEGER`, `PLS_INTEGER`, or `VARCHAR2`, or one of `VARCHAR2` subtypes `VARCHAR`, `STRING`, or `LONG`. `v_size` specifies the length of the `VARCHAR2` key.

size_limit

A positive integer literal that specifies the maximum size of a varray, which is the maximum number of elements the varray can contain. Note that a maximum limit is imposed. See ["Referencing Collection Elements"](#) on page 5-12.

type_name

A user-defined collection type that was defined using the datatype specifier `TABLE` or `VARRAY`.

Usage Notes

Nested tables extend the functionality of associative arrays (formerly known as index-by tables), so they differ in several ways. See ["Choosing Between Nested Tables and Associative Arrays"](#) on page 5-6.

Every element reference includes the collection name and one or more subscripts enclosed in parentheses; the subscripts determine which element is processed. Except for associative arrays, which can have negative subscripts, collection subscripts have a

fixed lower bound of 1. Subscripts for multilevel collections are evaluated in any order; if a subscript includes an expression that modifies the value of a different subscript, the result is undefined. See ["Referencing Collection Elements"](#) on page 5-12.

You can define all three collection types in the declarative part of any PL/SQL block, subprogram, or package. But, only nested table and varray types can be created and stored in an Oracle database.

Associative arrays and nested tables can be sparse (have nonconsecutive subscripts), but varrays are always dense (have consecutive subscripts). Unlike nested tables, varrays retain their ordering and subscripts when stored in the database. Initially, associative arrays are sparse. That enables you, for example, to store reference data in a temporary variable using a primary key (account numbers or employee numbers for example) as the index.

Collections follow the usual scoping and instantiation rules. In a package, collections are instantiated when you first reference the package and cease to exist when you end the database session. In a block or subprogram, local collections are instantiated when you enter the block or subprogram and cease to exist when you exit.

Until you initialize it, a nested table or varray is atomically null (that is, the collection itself is null, not its elements). To initialize a nested table or varray, you use a constructor, which is a system-defined function with the same name as the collection type. This function constructs (creates) a collection from the elements passed to it.

For information on collection comparisons that are allowed, see ["Comparing Collections"](#) on page 5-17.

Collections can store instances of an object type and, conversely, can be attributes of an object type. Collections can also be passed as parameters. You can use them to move columns of data into and out of database tables or between client-side applications and stored subprograms.

When invoking a function that returns a collection, you use the following syntax to reference elements in the collection:

```
function_name(parameter_list)(subscript)
```

See [Example 5-16, "Referencing an Element of an Associative Array"](#) on page 5-13 and [Example B-2, "Using the Dot Notation to Qualify Names"](#) on page B-2.

With the Oracle Call Interface (OCI) or the Oracle Precompilers, you can bind host arrays to associative arrays (index-by tables) declared as the formal parameters of a subprogram. That lets you pass host arrays to stored functions and procedures.

Examples

[Example 5-1, "Declaring Collection Types"](#) on page 5-4

[Example 5-3, "Declaring Nested Tables, Varrays, and Associative Arrays"](#) on page 5-8

[Example 5-4, "Declaring Collections with %TYPE"](#) on page 5-8

[Example 5-5, "Declaring a Procedure Parameter as a Nested Table"](#) on page 5-9

[Example 5-42, "Declaring and Initializing Record Types"](#) on page 5-31

Related Topics

["Collection Methods"](#) on page 13-23

["Object Type Declaration"](#) on page 13-100

["Record Definition"](#) on page 13-115

Collection Methods

A collection method is a built-in subprogram that operates on collections and is called using dot notation. You can use the methods `EXISTS`, `COUNT`, `LIMIT`, `FIRST`, `LAST`, `PRIOR`, `NEXT`, `EXTEND`, `TRIM`, and `DELETE` to manage collections whose size is unknown or varies.

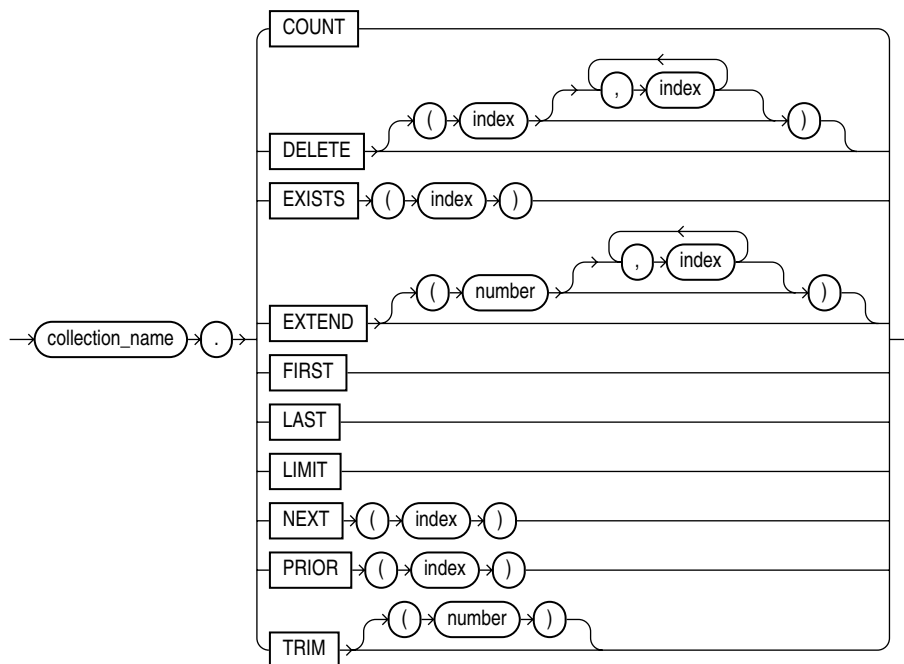
`EXISTS`, `COUNT`, `LIMIT`, `FIRST`, `LAST`, `PRIOR`, and `NEXT` are functions that check the properties of a collection or individual collection elements. `EXTEND`, `TRIM`, and `DELETE` are procedures that modify a collection.

`EXISTS`, `PRIOR`, `NEXT`, `TRIM`, `EXTEND`, and `DELETE` take integer parameters. `EXISTS`, `PRIOR`, `NEXT`, and `DELETE` can also take `VARCHAR2` parameters for associative arrays with string keys. `EXTEND` and `TRIM` cannot be used with index-by tables.

For more information, see ["Using Collection Methods"](#) on page 5-20.

Syntax

collection_call_method ::=



Keyword and Parameter Description

collection_name

An associative array, nested table, or varray previously declared within the current scope.

COUNT

Returns the number of elements that a collection currently contains, which is useful because the current size of a collection is not always known. You can use `COUNT` wherever an integer expression is allowed. For varrays, `COUNT` always equals `LAST`.

For nested tables, normally, `COUNT` equals `LAST`. But, if you delete elements from the middle of a nested table, `COUNT` is smaller than `LAST`.

DELETE

This procedure has three forms. `DELETE` removes all elements from a collection. `DELETE (n)` removes the n th element from an associative array or nested table. If n is null, `DELETE (n)` does nothing. `DELETE (m, n)` removes all elements in the range $m..n$ from an associative array or nested table. If m is larger than n or if m or n is null, `DELETE (m, n)` does nothing.

EXISTS

`EXISTS (n)` returns `TRUE` if the n th element in a collection exists. Otherwise, `EXISTS (n)` returns `FALSE`. Mainly, you use `EXISTS` with `DELETE` to maintain sparse nested tables. You can also use `EXISTS` to avoid raising an exception when you reference a nonexistent element. When passed an out-of-range subscript, `EXISTS` returns `FALSE` instead of raising `SUBSCRIPT_OUTSIDE_LIMIT`.

EXTEND

This procedure has three forms. `EXTEND` appends one null element to a collection. `EXTEND (n)` appends n null elements to a collection. `EXTEND (n, i)` appends n copies of the i th element to a collection. `EXTEND` operates on the internal size of a collection. If `EXTEND` encounters deleted elements, it includes them in its tally. You cannot use `EXTEND` with associative arrays.

FIRST, LAST

`FIRST` and `LAST` return the first and last (smallest and largest) subscript values in a collection. The subscript values are usually integers, but can also be strings for associative arrays. If the collection is empty, `FIRST` and `LAST` return `NULL`. If the collection contains only one element, `FIRST` and `LAST` return the same subscript value. For varrays, `FIRST` always returns 1 and `LAST` always equals `COUNT`. For nested tables, normally, `LAST` equals `COUNT`. But, if you delete elements from the middle of a nested table, `LAST` is larger than `COUNT`.

index

An expression that must return (or convert implicitly to) an integer in most cases, or a string for an associative array declared with string keys.

LIMIT

For nested tables, which have no maximum size, `LIMIT` returns `NULL`. For varrays, `LIMIT` returns the maximum number of elements that a varray can contain (which you must specify in its type definition).

NEXT

`NEXT (n)` returns the subscript that succeeds index n . If n has no successor, `NEXT (n)` returns `NULL`.

PRIOR

`PRIOR (n)` returns the subscript that precedes index n in a collection. If n has no predecessor, `PRIOR (n)` returns `NULL`.

TRIM

This procedure has two forms. TRIM removes one element from the end of a collection. TRIM(*n*) removes *n* elements from the end of a collection. If *n* is greater than COUNT, TRIM(*n*) raises SUBSCRIPT_BEYOND_COUNT. You cannot use TRIM with index-by tables. TRIM operates on the internal size of a collection. If TRIM encounters deleted elements, it includes them in its tally.

Usage Notes

You cannot use collection methods in a SQL statement. If you try, you get a compilation error.

Only EXISTS can be applied to atomically null collections. If you apply another method to such collections, PL/SQL raises COLLECTION_IS_NULL.

If the collection elements have sequential subscripts, you can use collection.FIRST .. collection.LAST in a FOR loop to iterate through all the elements. You can use PRIOR or NEXT to traverse collections indexed by any series of subscripts. For example, you can use PRIOR or NEXT to traverse a nested table from which some elements were deleted, or an associative array where the subscripts are string values.

EXTEND operates on the internal size of a collection, which includes deleted elements. You cannot use EXTEND to initialize an atomically null collection. Also, if you impose the NOT NULL constraint on a TABLE or VARRAY type, you cannot apply the first two forms of EXTEND to collections of that type.

If an element to be deleted does not exist, DELETE simply skips it; no exception is raised. Varrays are dense, so you cannot delete their individual elements. Because PL/SQL keeps placeholders for deleted elements, you can replace a deleted element by assigning it a new value. However, PL/SQL does not keep placeholders for trimmed elements.

The amount of memory allocated to a nested table can increase or decrease dynamically. As you delete elements, memory is freed page by page. If you delete the entire table, all the memory is freed.

In general, do not depend on the interaction between TRIM and DELETE. It is better to treat nested tables like fixed-size arrays and use only DELETE, or to treat them like stacks and use only TRIM and EXTEND.

Within a subprogram, a collection parameter assumes the properties of the argument bound to it. You can apply methods FIRST, LAST, COUNT, and so on to such parameters. For varray parameters, the value of LIMIT is always derived from the parameter type definition, regardless of the parameter mode.

Examples

- [Example 5-28, "Checking Whether a Collection Element EXISTS" on page 5-21](#)
- [Example 5-29, "Counting Collection Elements with COUNT" on page 5-21](#)
- [Example 5-30, "Checking the Maximum Size of a Collection with LIMIT" on page 5-22](#)
- [Example 5-31, "Using FIRST and LAST with a Collection" on page 5-23](#)
- [Example 5-32, "Using PRIOR and NEXT to Access Collection Elements" on page 5-24](#)
- [Example 5-34, "Using EXTEND to Increase the Size of a Collection" on page 5-25](#)
- [Example 5-35, "Using TRIM to Decrease the Size of a Collection" on page 5-26](#)
- [Example 5-37, "Using the DELETE Method on a Collection" on page 5-27](#)

Related Topics

["Collection Definition" on page 13-19](#)

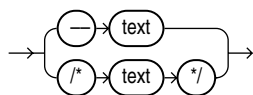
Comments

Comments let you include arbitrary text within your code to explain what the code does. You can also disable obsolete or unfinished pieces of code by turning them into comments.

PL/SQL supports two comment styles: single-line and multi-line. A double hyphen (--) anywhere on a line (except within a character literal) turns the rest of the line into a comment. Multi-line comments begin with a slash-asterisk (/*) and end with an asterisk-slash (* /). For more information, see ["Comments"](#) on page 2-8.

Syntax

comment ::=



Usage Notes

While testing or debugging a program, you might want to disable lines of code. Single-line comments can appear within a statement at the end of a line. You can include single-line comments inside multi-line comments, but you cannot nest multi-line comments.

You cannot use single-line comments in a PL/SQL block that will be processed dynamically by an Oracle Precompiler program. End-of-line characters are ignored, making the single-line comments extend to the end of the block. Instead, use multi-line comments. You can use multi-line comment delimiters to comment-out whole sections of code.

Examples

[Example 2-4, "Using Single-Line Comments"](#) on page 2-8

[Example 2-5, "Using Multi-Line Comments"](#) on page 2-8

COMMIT Statement

The `COMMIT` statement makes permanent any changes made to the database during the current transaction. A commit also makes the changes visible to other users. For more information on PL/SQL transaction processing, see "[Overview of Transaction Processing in PL/SQL](#)" on page 6-32.

The SQL `COMMIT` statement can be embedded as static SQL in PL/SQL. For syntax details on the SQL `COMMIT` statement, see the *Oracle Database SQL Language Reference*.

Usage Notes

The `COMMIT` statement releases all row and table locks, and erases any savepoints you marked since the last commit or rollback. Until your changes are committed:

- You can see the changes when you query the tables you modified, but other users cannot see the changes.
- If you change your mind or need to correct a mistake, you can use the `ROLLBACK` statement to roll back (undo) the changes.

If you commit while a `FOR UPDATE` cursor is open, a subsequent fetch on that cursor raises an exception. The cursor remains open, so you must still close it. For more information, see "[Using FOR UPDATE](#)" on page 6-38.

Examples

[Example 6-1, "Data Manipulation with PL/SQL"](#) on page 6-1

[Example 6-3, "Substituting PL/SQL Variables"](#) on page 6-2

[Example 6-36, "Using COMMIT with the WRITE Clause"](#) on page 6-33

[Example 6-40, "Using SET TRANSACTION to Begin a Read-only Transaction"](#) on page 6-37

[Example 6-43, "Declaring an Autonomous Function in a Package"](#) on page 6-42

Related Topics

["ROLLBACK Statement"](#) on page 13-124

["SAVEPOINT Statement"](#) on page 13-127

["Transaction Control Language \(TCL\) Statements"](#) on page 6-3

["Fetching Across Commits"](#) on page 6-39

See Also:

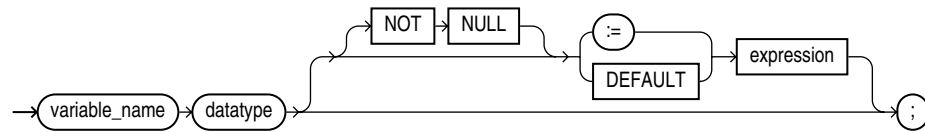
- *Oracle Database Advanced Application Developer's Guide* for information about committing transactions
- *Oracle Database Reference* for information about the `COMMIT_WRITE` initialization parameter

Constant and Variable Declaration

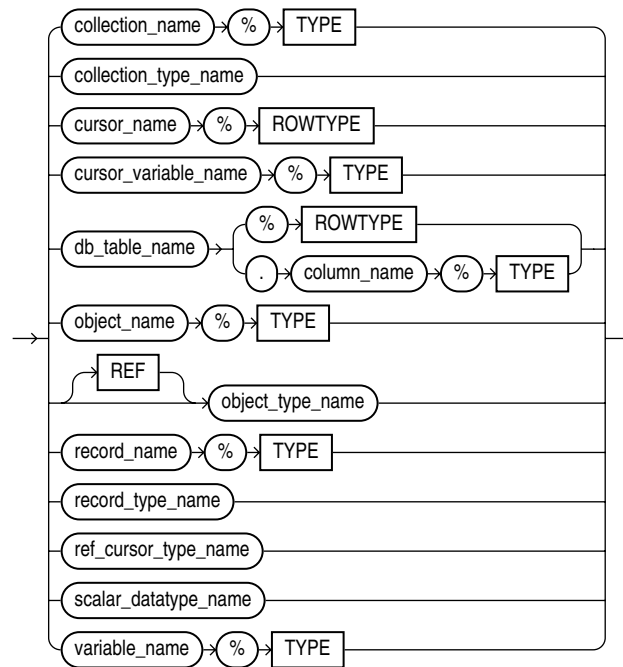
You can declare constants and variables in the declarative part of any PL/SQL block, subprogram, or package. Declarations allocate storage for a value, specify its datatype, and specify a name that you can reference. Declarations can also assign an initial value and impose the NOT NULL constraint. For more information, see "Declarations" on page 2-9.

Syntax

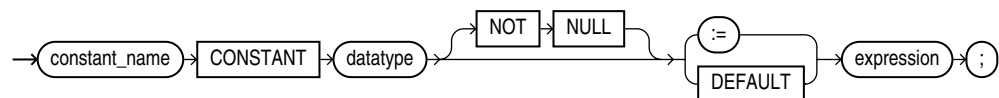
variable_declaration ::=



datatype ::=



constant_declaration ::=



Keyword and Parameter Description

collection_name

A collection (associative array, nested table, or varray) previously declared within the current scope.

collection_type_name

A user-defined collection type defined using the datatype specifier `TABLE` or `VARRAY`.

CONSTANT

Denotes the declaration of a constant. You must initialize a constant in its declaration. Once initialized, the value of a constant cannot be changed.

constant_name

A program constant. For naming conventions, see ["Identifiers"](#) on page 2-3.

cursor_name

An explicit cursor previously declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable previously declared within the current scope.

db_table_name

A database table or view that must be accessible when the declaration is elaborated.

db_table_name.column_name

A database table and column that must be accessible when the declaration is elaborated.

expression

A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the constant or variable. The value and the constant or variable must have compatible datatypes.

NOT NULL

A constraint that prevents the program from assigning a null value to a variable or constant. Assigning a null to a variable defined as `NOT NULL` raises the predefined exception `VALUE_ERROR`. The constraint `NOT NULL` must be followed by an initialization clause.

object_name

An instance of an object type previously declared within the current scope.

record_name

A user-defined or `%ROWTYPE` record previously declared within the current scope.

record_name.field_name

A field in a user-defined or `%ROWTYPE` record previously declared within the current scope.

record_type_name

A user-defined record type that is defined using the datatype specifier `RECORD`.

ref_cursor_type_name

A user-defined cursor variable type, defined using the datatype specifier `REF CURSOR`.

%ROWTYPE

Represents a record that can hold a row from a database table or a cursor. Fields in the record have the same names and datatypes as columns in the row.

scalar_datatype_name

A predefined scalar datatype such as `BOOLEAN`, `NUMBER`, or `VARCHAR2`. Includes any qualifiers for size, precision, and character or byte semantics.

%TYPE

Represents the datatype of a previously declared collection, cursor variable, field, object, record, database column, or variable.

variable_name

A program variable.

Usage Notes

Constants and variables are initialized every time a block or subprogram is entered. By default, variables are initialized to `NULL`. Whether public or private, constants and variables declared in a package spec are initialized only once for each session.

An initialization clause is required when declaring `NOT NULL` variables and when declaring constants. If you use `%ROWTYPE` to declare a variable, initialization is not allowed.

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

Examples

[Example 1–2, "Declaring Variables in PL/SQL"](#) on page 1-5

[Example 1–3, "Assigning Values to Variables with the Assignment Operator"](#) on page 1-6

[Example 1–4, "Using SELECT INTO to Assign Values to Variables"](#) on page 1-6

[Example 2–9, "Using the %ROWTYPE Attribute"](#) on page 2-12

Related Topics

["Declaring Variables"](#) on page 1-5

["Declarations"](#) on page 2-9

["Predefined PL/SQL Scalar Datatypes and Subtypes"](#) on page 3-1

["Assignment Statement"](#) on page 13-3

["Expression Definition"](#) on page 13-57

["%ROWTYPE Attribute"](#) on page 13-125

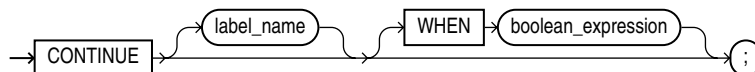
["%TYPE Attribute"](#) on page 13-142

CONTINUE Statement

The `CONTINUE` statement exits the current iteration of a loop and transfers control to the next iteration. The `CONTINUE` statement has two forms: the unconditional `CONTINUE` and the conditional `CONTINUE WHEN`. With either form, you can name the loop to be exited. For more information, see "[Controlling Loop Iterations \(LOOP, EXIT, and CONTINUE Statements\)](#)" on page 4-7.

Syntax

continue_statement ::=



Keyword and Parameter Description

boolean_expression

An expression that returns the Boolean value `TRUE`, `FALSE`, or `NULL`. It is evaluated with each iteration of the loop. If the expression returns `TRUE`, the current iteration of the loop (or the iteration of the loop identified by `label_name`) is exited immediately. For the syntax of `boolean_expression`, see "[Expression Definition](#)" on page 13-57.

CONTINUE

An unconditional `CONTINUE` statement (that is, one without a `WHEN` clause) exits the current iteration of the loop immediately. Execution resumes with the next iteration of the loop.

label_name

Identifies the loop exit from either the current loop, or any enclosing labeled loop.

Usage Notes

The `CONTINUE` statement can be used only inside a loop; you cannot exit from a block directly.

If you use a `CONTINUE` statement to exit 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 is closed automatically (in this context, `CONTINUE` works like `GOTO`). The cursor is also closed automatically if an exception is raised inside the loop.

Examples

[Example](#) , "[Using the CONTINUE Statement](#)" on page 4-9

[Example](#) , "[Using the CONTINUE-WHEN Statement](#)" on page 4-10

Related Topics

["Expression Definition"](#) on page 13-57

["LOOP Statements"](#) on page 13-93

["EXIT Statement"](#) on page 13-55

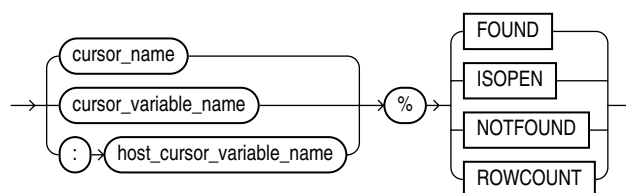
Cursor Attributes

Every explicit cursor and cursor variable has four attributes: `%FOUND`, `%ISOPEN`, `%NOTFOUND`, and `%ROWCOUNT`. When appended to the cursor or cursor variable, these attributes return useful information about the execution of a data manipulation statement. For more information, see ["Using Cursor Expressions"](#) on page 6-31.

The implicit cursor `SQL` has additional attributes, `%BULK_ROWCOUNT` and `%BULK_EXCEPTIONS`. For more information, see ["SQL Cursor"](#) on page 13-136.

Syntax

cursor_attribute ::=



Keyword and Parameter Description

cursor_name

An explicit cursor previously declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable (or parameter) previously declared within the current scope.

%FOUND Attribute

A cursor attribute that can be appended to the name of a cursor or cursor variable. Before the first fetch from an open cursor, `cursor_name%FOUND` returns `NULL`. Afterward, it returns `TRUE` if the last fetch returned a row, or `FALSE` if the last fetch failed to return a row.

host_cursor_variable_name

A cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

%ISOPEN Attribute

A cursor attribute that can be appended to the name of a cursor or cursor variable. If a cursor is open, `cursor_name%ISOPEN` returns `TRUE`; otherwise, it returns `FALSE`.

%NOTFOUND Attribute

A cursor attribute that can be appended to the name of a cursor or cursor variable. Before the first fetch from an open cursor, `cursor_name%NOTFOUND` returns `NULL`. Thereafter, it returns `FALSE` if the last fetch returned a row, or `TRUE` if the last fetch failed to return a row.

%ROWCOUNT Attribute

A cursor attribute that can be appended to the name of a cursor or cursor variable. When a cursor is opened, %ROWCOUNT is zeroed. Before the first fetch, `cursor_name%ROWCOUNT` returns 0. Thereafter, it returns the number of rows fetched so far. The number is incremented if the latest fetch returned a row.

Usage Notes

The cursor attributes apply to every cursor or cursor variable. For example, you can open multiple cursors, then use %FOUND or %NOTFOUND to tell which cursors have rows left to fetch. Likewise, you can use %ROWCOUNT to tell how many rows were fetched so far.

If a cursor or cursor variable is not open, referencing it with %FOUND, %NOTFOUND, or %ROWCOUNT raises the predefined exception `INVALID_CURSOR`.

When a cursor or cursor variable is opened, the rows that satisfy the associated query are identified and form the result set. Rows are fetched from the result set one at a time.

If a `SELECT INTO` statement returns more than one row, PL/SQL raises the predefined exception `TOO_MANY_ROWS` and sets %ROWCOUNT to 1, not the actual number of rows that satisfy the query.

Before the first fetch, %NOTFOUND evaluates to NULL. If `FETCH` never executes successfully, the `EXIT WHEN` condition is never TRUE and the loop is never exited. To be safe, use the following `EXIT` statement instead:

```
EXIT WHEN c1%NOTFOUND OR c1%NOTFOUND IS NULL;
```

You can use the cursor attributes in procedural statements, but not in SQL statements.

Examples

[Example 6-7, "Using SQL%FOUND" on page 6-8](#)

[Example 6-8, "Using SQL%ROWCOUNT" on page 6-8](#)

[Example 6-10, "Fetching with a Cursor" on page 6-11](#)

[Example 6-15, "Using %ISOPEN" on page 6-14](#)

Related Topics

["Cursor Declaration" on page 13-41](#)

["Cursor Variables" on page 13-37](#)

["Managing Cursors in PL/SQL" on page 6-7](#)

Cursor Variables

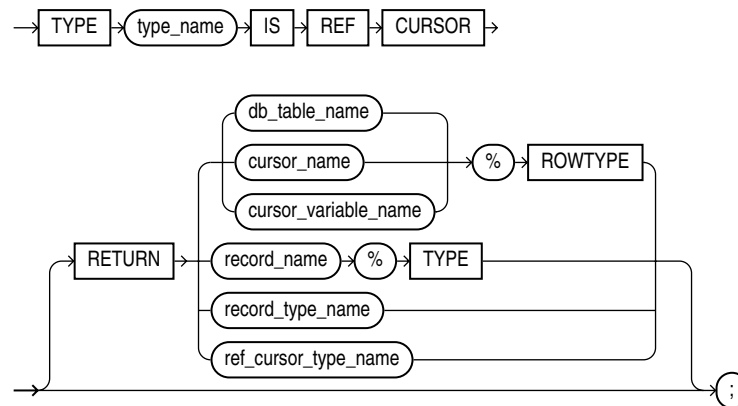
To execute a multiple-row query, Oracle opens an unnamed work area that stores processing information. You can access this area through an explicit cursor, which names the work area, or through a cursor variable, which points to the work area. To create cursor variables, you define a REF CURSOR type, then declare cursor variables of that type.

Cursor variables are like C or Pascal pointers, which hold the address of some item instead of the item itself. Declaring a cursor variable creates a pointer, not an item.

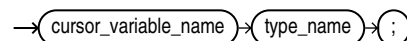
For more information, see ["Using Cursor Variables \(REF CURSORS\)"](#) on page 6-22.

Syntax

ref_cursor_type_definition ::=



ref_cursor_variable_declaration ::=



Keyword and Parameter Description

cursor_name

An explicit cursor previously declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable previously declared within the current scope.

db_table_name

A database table or view, which must be accessible when the declaration is elaborated.

record_name

A user-defined record previously declared within the current scope.

record_type_name

A user-defined record type that was defined using the datatype specifier RECORD.

REF CURSOR

Cursor variables all have the datatype `REF CURSOR`.

RETURN

Specifies the datatype of a cursor variable return value. You can use the `%ROWTYPE` attribute in the `RETURN` clause to provide a record type that represents a row in a database table, or a row from a cursor or strongly typed cursor variable. You can use the `%TYPE` attribute to provide the datatype of a previously declared record.

%ROWTYPE

A record type that represents a row in a database table or a row fetched from a cursor or strongly typed cursor variable. Fields in the record and corresponding columns in the row have the same names and datatypes.

%TYPE

Provides the datatype of a previously declared user-defined record.

type_name

A user-defined cursor variable type that was defined as a `REF CURSOR`.

Usage Notes

Cursor variables are available to every PL/SQL client. For example, you can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program, then pass it as a bind argument to PL/SQL. Application development tools that have a PL/SQL engine can use cursor variables entirely on the client side.

You can pass cursor variables back and forth between an application and the database server through remote procedure invokes using a database link. If you have a PL/SQL engine on the client side, you can use the cursor variable in either location. For example, you can declare a cursor variable on the client side, open and fetch from it on the server side, then continue to fetch from it back on the client side.

You use cursor variables to pass query result sets between PL/SQL stored subprograms and client programs. Neither PL/SQL nor any client program owns a result set; they share a pointer to the work area where the result set is stored. For example, an OCI program, Oracle Forms application, and the database can all refer to the same work area.

`REF CURSOR` types can be *strong* or *weak*. A strong `REF CURSOR` type definition specifies a return type, but a weak definition does not. Strong `REF CURSOR` types are less error-prone because PL/SQL lets you associate a strongly typed cursor variable only with type-compatible queries. Weak `REF CURSOR` types are more flexible because you can associate a weakly typed cursor variable with any query.

Once you define a `REF CURSOR` type, you can declare cursor variables of that type. You can use `%TYPE` to provide the datatype of a record variable. Also, in the `RETURN` clause of a `REF CURSOR` type definition, you can use `%ROWTYPE` to specify a record type that represents a row returned by a strongly (not weakly) typed cursor variable.

Currently, cursor variables are subject to several restrictions. See "[Restrictions on Cursor Variables](#)" on page 6-30.

You use three statements to control a cursor variable: `OPEN-FOR`, `FETCH`, and `CLOSE`. First, you `OPEN` a cursor variable `FOR` a multiple-row query. Then, you `FETCH` rows from the result set. When all the rows are processed, you `CLOSE` the cursor variable.

Other `OPEN-FOR` statements can open the same cursor variable for different queries. You need not close a cursor variable before reopening it. When you reopen a cursor variable for a different query, the previous query is lost.

PL/SQL makes sure the return type of the cursor variable is compatible with the `INTO` clause of the `FETCH` statement. For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-compatible field or variable in the `INTO` clause. Also, the number of fields or variables must equal the number of column values. Otherwise, you get an error.

If both cursor variables involved in an assignment are strongly typed, they must have the same datatype. However, if one or both cursor variables are weakly typed, they need not have the same datatype.

When declaring a cursor variable as the formal parameter of a subprogram that fetches from or closes the cursor variable, you must specify the `IN` or `IN OUT` mode. If the subprogram opens the cursor variable, you must specify the `IN OUT` mode.

Be careful when passing cursor variables as parameters. At run time, PL/SQL raises `ROWTYPE_MISMATCH` if the return types of the actual and formal parameters are incompatible.

You can apply the cursor attributes `%FOUND`, `%NOTFOUND`, `%ISOPEN`, and `%ROWCOUNT` to a cursor variable.

If you try to fetch from, close, or apply cursor attributes to a cursor variable that does not point to a query work area, PL/SQL raises the predefined exception `INVALID_CURSOR`. You can make a cursor variable (or parameter) point to a query work area in two ways:

- `OPEN` the cursor variable `FOR` the query.
- Assign to the cursor variable the value of an already `OPENED` host cursor variable or PL/SQL cursor variable.

A query work area remains accessible as long as any cursor variable points to it. Therefore, you can pass the value of a cursor variable freely from one scope to another. For example, if you pass a host cursor variable to a PL/SQL block embedded in a Pro*C program, the work area to which the cursor variable points remains accessible after the block completes.

Examples

[Example 6-9, "Declaring a Cursor"](#) on page 6-10

[Example 6-10, "Fetching with a Cursor"](#) on page 6-11

[Example 6-13, "Fetching Bulk Data with a Cursor"](#) on page 6-12

[Example 6-27, "Passing a REF CURSOR as a Parameter"](#) on page 6-24

[Example 6-29, "Stored Procedure to Open a Ref Cursor"](#) on page 6-26

[Example 6-30, "Stored Procedure to Open Ref Cursors with Different Queries"](#) on page 6-26

[Example 6-31, "Cursor Variable with Different Return Types"](#) on page 6-27

Related Topics

["CLOSE Statement"](#) on page 13-18

["Cursor Attributes"](#) on page 13-35

["Cursor Declaration"](#) on page 13-41

["FETCH Statement"](#) on page 13-67

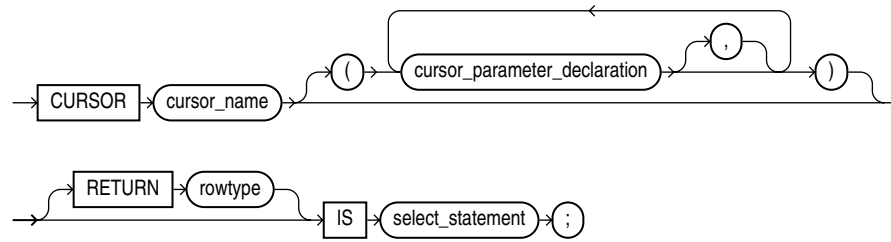
["OPEN-FOR Statement"](#) on page 13-104

Cursor Declaration

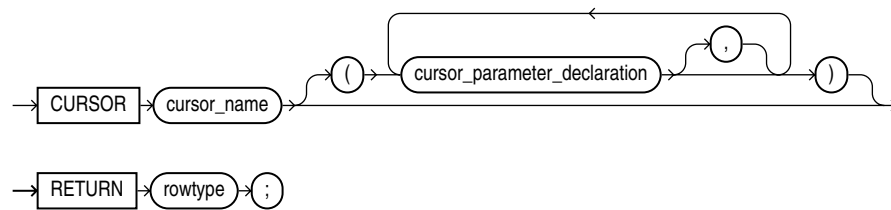
To execute a multiple-row query, Oracle opens an unnamed work area that stores processing information. A cursor lets you name the work area, access the information, and process the rows individually. For more information, see ["Querying Data with PL/SQL"](#) on page 6-16.

Syntax

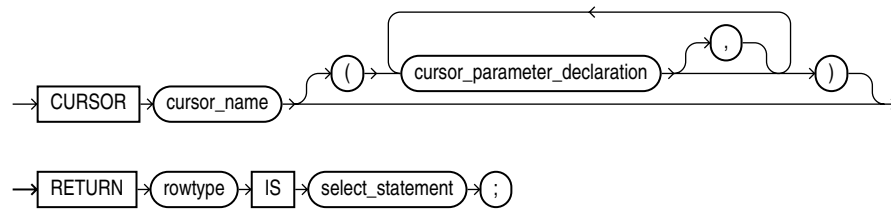
cursor_declaration ::=



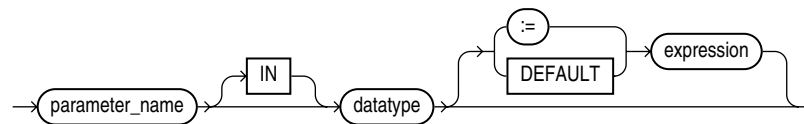
cursor_spec ::=



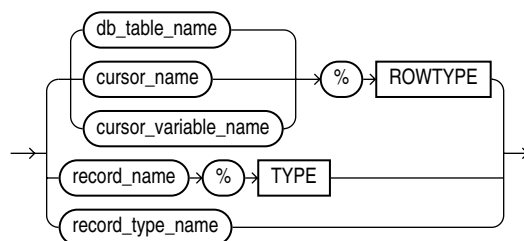
cursor_body ::=



cursor_parameter_declaration ::=



rowtype ::=



Keyword and Parameter Description

cursor_name

An explicit cursor previously declared within the current scope.

datatype

A type specifier. For the syntax of `datatype`, see ["Constant and Variable Declaration"](#) on page 13-29.

db_table_name

A database table or view that must be accessible when the declaration is elaborated.

expression

A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the parameter. The value and the parameter must have compatible datatypes.

Note: If you supply an actual parameter for `parameter_name` when you open the cursor, then `expression` is not evaluated.

parameter_name

A variable declared as the formal parameter of a cursor. A cursor parameter can appear in a query wherever a constant can appear. The formal parameters of a cursor must be `IN` parameters. The query can also reference other PL/SQL variables within its scope.

record_name

A user-defined record previously declared within the current scope.

record_type_name

A user-defined record type that was defined using the datatype specifier `RECORD`.

RETURN

Specifies the datatype of a cursor return value. You can use the `%ROWTYPE` attribute in the `RETURN` clause to provide a record type that represents a row in a database table or a row returned by a previously declared cursor. Also, you can use the `%TYPE` attribute to provide the datatype of a previously declared record.

A cursor body must have a `SELECT` statement and the same `RETURN` clause as its corresponding cursor spec. Also, the number, order, and datatypes of select items in the `SELECT` clause must match the `RETURN` clause.

%ROWTYPE

A record type that represents a row in a database table or a row fetched from a previously declared cursor or cursor variable. Fields in the record and corresponding columns in the row have the same names and datatypes.

select_statement

A query that returns a result set of rows. Its syntax is like that of `select_into_statement` without the `INTO` clause. See ["SELECT INTO Statement"](#) on page 13-128. If the cursor declaration declares parameters, each parameter must be used in the query.

%TYPE

Provides the datatype of a previously declared user-defined record.

Usage Notes

You must declare a cursor before referencing it in an `OPEN`, `FETCH`, or `CLOSE` statement. You must declare a variable before referencing it in a cursor declaration. The word `SQL` is reserved by PL/SQL as the default name for implicit cursors, and cannot be used in a cursor declaration.

You cannot assign values to a cursor name or use it in an expression. However, cursors and variables follow the same scoping rules. For more information, see ["Scope and Visibility of PL/SQL Identifiers"](#) on page 2-16.

You retrieve data from a cursor by opening it, then fetching from it. Because the `FETCH` statement specifies the target variables, using an `INTO` clause in the `SELECT` statement of a `cursor_declaration` is redundant and invalid.

The scope of cursor parameters is local to the cursor, meaning that they can be referenced only within the query used in the cursor declaration. The values of cursor parameters are used by the associated query when the cursor is opened. The query can also reference other PL/SQL variables within its scope.

The datatype of a cursor parameter must be specified without constraints, that is, without precision and scale for numbers, and without length for strings.

Examples

[Example 6–9, "Declaring a Cursor"](#) on page 6-10

[Example 6–10, "Fetching with a Cursor"](#) on page 6-11

[Example 6–13, "Fetching Bulk Data with a Cursor"](#) on page 6-12

[Example 6–27, "Passing a REF CURSOR as a Parameter"](#) on page 6-24

[Example 6–29, "Stored Procedure to Open a Ref Cursor"](#) on page 6-26

[Example 6–30, "Stored Procedure to Open Ref Cursors with Different Queries"](#) on page 6-26

Related Topics

["CLOSE Statement"](#) on page 13-18

["FETCH Statement"](#) on page 13-67

["OPEN Statement"](#) on page 13-102

["SELECT INTO Statement"](#) on page 13-128

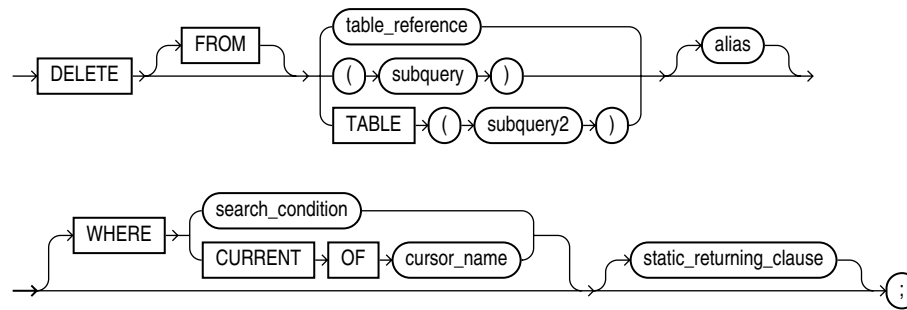
["Declaring a Cursor"](#) on page 6-10

DELETE Statement

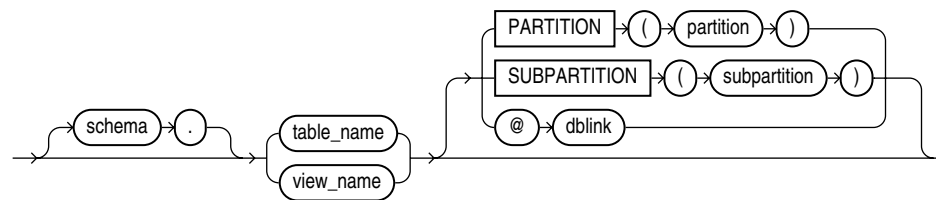
The `DELETE` statement removes rows of data from a specified table or view. For a full description of the `DELETE` statement, see *Oracle Database SQL Language Reference*.

Syntax

delete_statement ::=



table_reference ::=



Keyword and Parameter Description

alias

Another (usually short) name for the referenced table or view. Typically referred to later in the `WHERE` clause.

static_returning_clause

Returns the column values of the deleted rows, in either individual variables or collections (eliminating the need to `SELECT` the rows first). For details, see "[RETURNING INTO Clause](#)" on page 13-121.

subquery

A `SELECT` statement that provides a set of rows for processing. Its syntax is like the `select_into_statement` without the `INTO` clause. See "[SELECT INTO Statement](#)" on page 13-128.

table_reference

A table or view, which must be accessible when you execute the `DELETE` statement, and for which you must have `DELETE` privileges.

TABLE (subquery2)

The operand of `TABLE` is a `SELECT` statement that returns a single column value, which must be a nested table. Operator `TABLE` informs Oracle that the value is a collection, not a scalar value.

WHERE CURRENT OF cursor_name

Refers to the latest row processed by the `FETCH` statement associated with the cursor identified by `cursor_name`. The cursor must be `FOR UPDATE` and must be open and positioned on a row. If the cursor is not open, the `CURRENT OF` clause causes an error.

If the cursor is open, but no rows were fetched or the last fetch returned no rows, PL/SQL raises the predefined exception `NO_DATA_FOUND`.

WHERE search_condition

Conditionally chooses rows to be deleted from the referenced table or view. Only rows that meet the search condition are deleted. If you omit the `WHERE` clause, all rows in the table or view are deleted.

Usage Notes

You can use the `DELETE WHERE CURRENT OF` statement after a fetch from an open cursor (this includes implicit fetches executed in a cursor `FOR` loop), provided the associated query is `FOR UPDATE`. This statement deletes the current row; that is, the one just fetched.

The implicit cursor `SQL` and the cursor attributes `%NOTFOUND`, `%FOUND`, and `%ROWCOUNT` let you access useful information about the execution of a `DELETE` statement.

Examples

[Example 6-1, "Data Manipulation with PL/SQL"](#) on page 6-1

[Example 6-5, "Using CURRVAL and NEXTVAL"](#) on page 6-4

[Example 6-7, "Using SQL%FOUND"](#) on page 6-8

[Example 6-8, "Using SQL%ROWCOUNT"](#) on page 6-8

[Example 12-2, "Issuing DELETE Statements in a Loop"](#) on page 12-12

[Example 12-16, "Using FORALL with BULK COLLECT"](#) on page 12-23

Related Topics

["FETCH Statement"](#) on page 13-67

["INSERT Statement"](#) on page 13-85

["SELECT INTO Statement"](#) on page 13-128

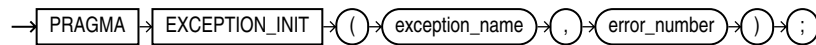
["UPDATE Statement"](#) on page 13-144

EXCEPTION_INIT Pragma

The `EXCEPTION_INIT` pragma associates a user-defined exception name with an Oracle error number. You can intercept any ORA- error and write a specific handler for it instead of using the `OTHERS` handler. For more information, see ["Associating a PL/SQL Exception with a Number \(Pragma EXCEPTION_INIT\)"](#) on page 11-7.

Syntax

exception_init_pragma ::=



Keyword and Parameter Description

error_number

Any valid Oracle error number. These are the same error numbers (always negative) returned by the function `SQLCODE`.

exception_name

A user-defined exception declared within the current scope.

PRAGMA

Signifies that the statement is a pragma (compiler directive). Pragmas are processed at compile time, not at run time. They pass information to the compiler.

Usage Notes

You can use `EXCEPTION_INIT` in the declarative part of any PL/SQL block, subprogram, or package. The pragma must appear in the same declarative part as its associated exception, somewhere after the exception declaration.

Be sure to assign only one exception name to an error number.

Examples

[Example 11-4, "Using PRAGMA EXCEPTION_INIT"](#) on page 11-7

[Example 12-9, "Bulk Operation that Continues Despite Exceptions"](#) on page 12-18

Related Topics

["AUTONOMOUS_TRANSACTION Pragma"](#) on page 13-6

["Exception Definition"](#) on page 13-48

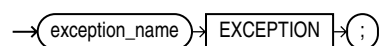
["SQLCODE Function"](#) on page 13-139

Exception Definition

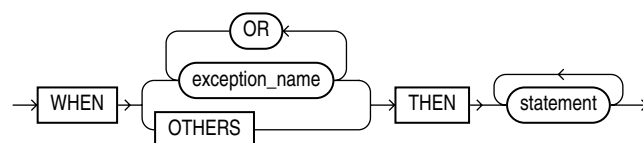
An exception is a run-time error or warning condition, which can be predefined or user-defined. Predefined exceptions are raised implicitly (automatically) by the run-time system. User-defined exceptions must be raised explicitly by `RAISE` statements. To handle raised exceptions, you write separate routines called exception handlers. For more information, see [Chapter 11, "Handling PL/SQL Errors"](#).

Syntax

exception_definition ::=



exception_handler ::=



Keyword and Parameter Description

exception_name

A predefined exception such as `ZERO_DIVIDE`, or a user-defined exception previously declared within the current scope.

OTHERS

Stands for all the exceptions not explicitly named in the exception-handling part of the block. The use of `OTHERS` is optional and is allowed only as the last exception handler. You cannot include `OTHERS` in a list of exceptions following the keyword `WHEN`.

statement

An executable statement. For the syntax of `statement`, see ["Block Declaration"](#) on page 13-8.

WHEN

Introduces an exception handler. You can have multiple exceptions execute the same sequence of statements by following the keyword `WHEN` with a list of the exceptions, separating them by the keyword `OR`. If any exception in the list is raised, the associated statements are executed.

Usage Notes

An exception declaration can appear only in the declarative part of a block, subprogram, or package. The scope rules for exceptions and variables are the same. But, unlike variables, exceptions cannot be passed as parameters to subprograms.

Some exceptions are predefined by PL/SQL. For a list of these exceptions, see ["Summary of Predefined PL/SQL Exceptions"](#) on page 11-4. PL/SQL declares

predefined exceptions globally in package `STANDARD`, so you need not declare them yourself.

Redeclaring predefined exceptions is error prone because your local declaration overrides the global declaration. In such cases, you must use dot notation to specify the predefined exception, as follows:

```
EXCEPTION
    WHEN invalid_number OR STANDARD.INVALID_NUMBER THEN ...
```

The exception-handling part of a PL/SQL block is optional. Exception handlers must come at the end of the block. They are introduced by the keyword `EXCEPTION`. The exception-handling part of the block is terminated by the same keyword `END` that terminates the entire block. An exception handler can reference only those variables that the current block can reference.

Raise an exception only when an error occurs that makes it undesirable or impossible to continue processing. If there is no exception handler in the current block for a raised exception, the exception propagates according to the following rules:

- If there is an enclosing block for the current block, the exception is passed on to that block. The enclosing block then becomes the current block. If a handler for the raised exception is not found, the process repeats.
- If there is no enclosing block for the current block, an *unhandled exception* error is passed back to the host environment.

Only one exception at a time can be active in the exception-handling part of a block. Therefore, if an exception is raised inside a handler, the block that encloses the current block is the first block searched to find a handler for the newly raised exception. From there on, the exception propagates normally.

Example

[Example 1–10, "Using WHILE-LOOP for Control" on page 1-11](#)
[Example 1–13, "Creating a Stored Subprogram" on page 1-14](#)
[Example 2–19, "PL/SQL Block Using Multiple and Duplicate Labels" on page 2-19](#)
[Example 5–35, "Using TRIM to Decrease the Size of a Collection" on page 5-26](#)
[Example 5–38, "Collection Exceptions" on page 5-28](#)
[Example 6–37, "Using ROLLBACK" on page 6-34](#)
[Example 7–12, "Dynamic SQL" on page 7-14](#)
[Example 8–1, "Simple PL/SQL Procedure" on page 8-3](#)
[Example 10–3, "Creating the emp_admin Package" on page 10-6](#)
[Example 11–1, "Run-Time Error Handling" on page 11-2](#)
[Example 11–3, "Scope of PL/SQL Exceptions" on page 11-6](#)
[Example 11–9, "Reraising a PL/SQL Exception" on page 11-12](#)
[Example 12–6, "Using Rollbacks with FORALL" on page 12-15](#)
[Example 12–9, "Bulk Operation that Continues Despite Exceptions" on page 12-18](#)

Related Topics

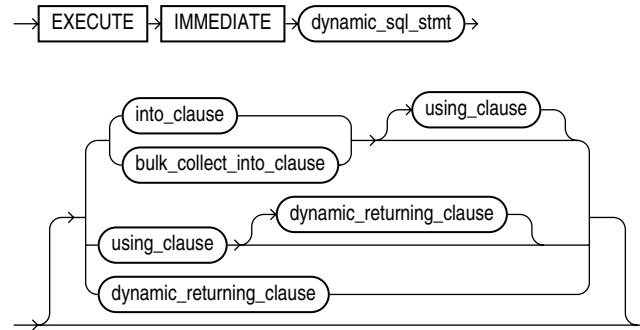
["Block Declaration" on page 13-8](#)
["EXCEPTION_INIT Pragma" on page 13-47](#)
["RAISE Statement" on page 13-114](#)

EXECUTE IMMEDIATE Statement

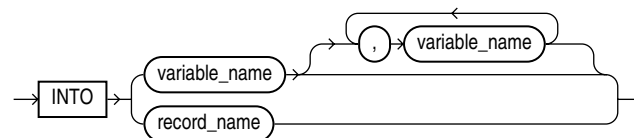
The `EXECUTE IMMEDIATE` statement builds and executes a dynamic SQL statement in a single operation. For more information about this statement, see ["Using the EXECUTE IMMEDIATE Statement"](#) on page 7-2.

Syntax

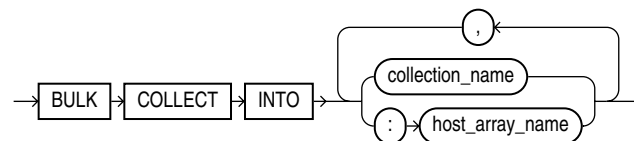
execute_immediate_statement ::=



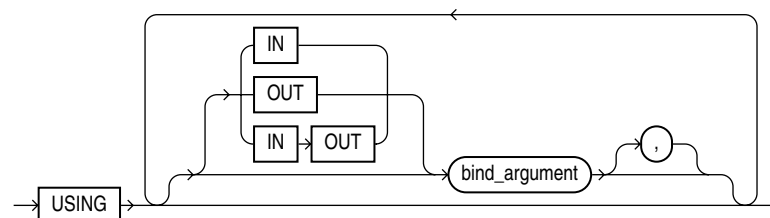
into_clause ::=



bulk_collect_into_clause ::=



using_clause ::=



Keyword and Parameter Description

bind_argument

Either an expression whose value is passed to the dynamic SQL statement (an **in bind**), or a variable in which a value returned by the dynamic SQL statement is stored (an **out bind**).

BULK COLLECT INTO

Used if and only if `dynamic_sql_stmt` can return multiple rows, this clause specifies one or more collections in which to store the returned rows. This clause must have a corresponding, type-compatible `collection_item` or `:host_array_name` for each `select_item` in `dynamic_sql_stmt`.

collection_name

The name of a declared collection, in which to store rows returned by `dynamic_sql_stmt`.

dynamic_returning_clause

Used if and only if `dynamic_sql_stmt` has a `RETURNING INTO` clause, this clause returns the column values of the rows affected by `dynamic_sql_stmt`, in either individual variables or records (eliminating the need to select the rows first). This clause can include `OUT` bind arguments. For details, see ["RETURNING INTO Clause"](#) on page 13-121.

dynamic_sql_stmt

A string literal, string variable, or string expression that represents any SQL statement. It must be of type `CHAR`, `VARCHAR2`, or `CLOB`.

host_array_name

An array into which returned rows are stored. The array must be declared in a PL/SQL host environment and passed to PL/SQL as a bind argument (hence the colon (:)) prefix).

IN, OUT, IN OUT

Parameter modes of bind arguments. An `IN` bind argument passes its value to the dynamic SQL statement. An `OUT` bind argument stores a value that the dynamic SQL statement returns. An `IN OUT` bind argument passes its initial value to the dynamic SQL statement and stores a value that the dynamic SQL statement returns. The default parameter mode for `bind_argument` is `IN`.

INTO

Used if and only if `dynamic_sql_stmt` is a `SELECT` statement that can return at most one row, this clause specifies the variables or record into which the column values of the returned row are stored. For each `select_item` in `dynamic_sql_stmt`, this clause must have either a corresponding, type-compatible `define_variable` or a type-compatible record.

record_name

A user-defined or `%ROWTYPE` record into which a returned row is stored.

USING

Used only if `dynamic_sql_stmt` includes placeholders, this clause specifies a list of bind arguments.

variable_name

The name of a define variable in which to store a column value of the row returned by `dynamic_sql_stmt`.

Usage Notes

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

At run time, bind arguments or define variables replace corresponding placeholders in the dynamic SQL statement. Every placeholder must be associated with a bind argument in the USING clause or RETURNING INTO clause (or both) or with a define variable in the INTO clause.

The value of a bind argument cannot be a Boolean literal (TRUE, FALSE, or NULL). To pass the value NULL to the dynamic SQL statement, use an uninitialized variable where you want to use NULL, as in ["Uninitialized Variable for NULL in USING Clause"](#) on page 7-4.

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

Note: When using dynamic SQL, be aware of SQL injection, a security risk. For more information on SQL injection, see ["Avoiding SQL Injection in PL/SQL"](#) on page 7-9.

Examples

[Example 7-1, "Invoking a Subprogram from a Dynamic PL/SQL Block"](#)
[Example 7-1, "Invoking a Subprogram from a Dynamic PL/SQL Block"](#)
[Example 7-2, "Unsupported Datatype in Native Dynamic SQL"](#)
[Example 7-3, "Uninitialized Variable for NULL in USING Clause"](#)
[Example 7-5, "Repeated Placeholder Names in Dynamic PL/SQL Block"](#)

Related Topics

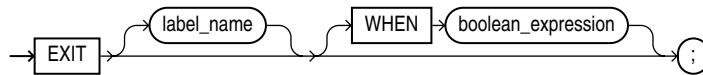
["OPEN-FOR Statement"](#) on page 13-104
["RETURNING INTO Clause"](#) on page 13-121

EXIT Statement

The `EXIT` statement exits a loop and transfers control to the end of the loop. The `EXIT` statement has two forms: the unconditional `EXIT` and the conditional `EXIT WHEN`. With either form, you can name the loop to be exited. For more information, see ["Controlling Loop Iterations \(LOOP, EXIT, and CONTINUE Statements\)"](#) on page 4-7.

Syntax

exit_statement ::=



Keyword and Parameter Description

boolean_expression

An expression that returns the Boolean value `TRUE`, `FALSE`, or `NULL`. It is evaluated with each iteration of the loop. If the expression returns `TRUE`, the current loop (or the loop labeled by `label_name`) is exited immediately. For the syntax of `boolean_expression`, see ["Expression Definition"](#) on page 13-57.

EXIT

An unconditional `EXIT` statement (that is, one without a `WHEN` clause) exits the current loop immediately. Execution resumes with the statement following the loop.

label_name

Identifies the loop exit from: either the current loop, or any enclosing labeled loop.

Usage Notes

The `EXIT` statement can be used only inside a loop; you cannot exit from a block directly. PL/SQL lets you code an infinite loop. For example, the following loop will never terminate normally so you must use an `EXIT` statement to exit the loop.

```
WHILE TRUE LOOP ... END LOOP;
```

If you use an `EXIT` statement to exit a cursor `FOR` loop prematurely, the cursor is closed automatically. The cursor is also closed automatically if an exception is raised inside the loop.

Examples

[Example 4-8, "Using an EXIT Statement"](#) on page 4-8

[Example 4-20, "Using EXIT in a LOOP"](#) on page 4-15

[Example 4-21, "Using EXIT with a Label in a LOOP"](#) on page 4-15

Related Topics

["Expression Definition"](#) on page 13-57

["LOOP Statements"](#) on page 13-93

["CONTINUE Statement"](#) on page 13-33

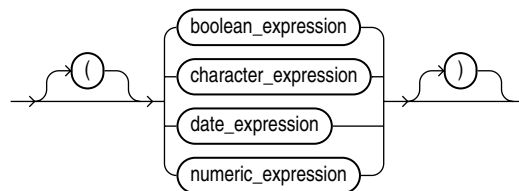
Expression Definition

An expression is an arbitrarily complex combination of operands (variables, constants, literals, operators, function calls, and placeholders) and operators. The simplest expression is a single variable.

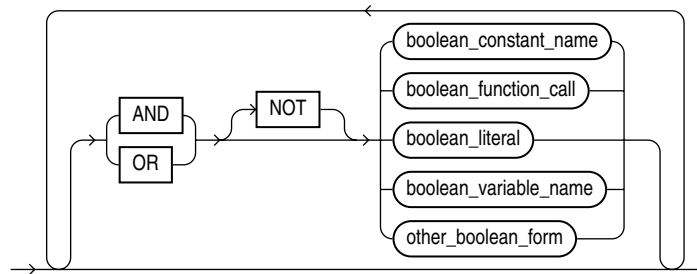
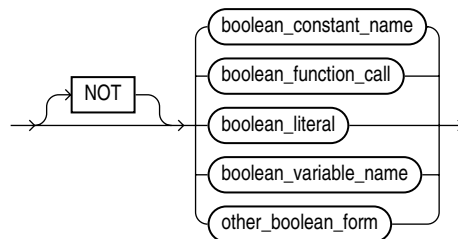
The PL/SQL compiler determines the datatype of an expression from the types of the operands and operators that comprise the expression. Every time the expression is evaluated, a single value of that type results.

Syntax

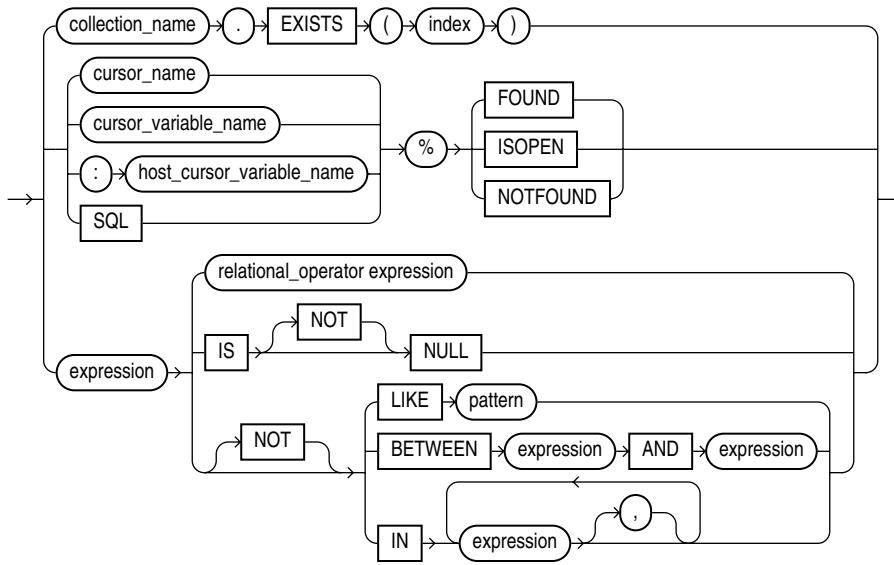
expression ::=



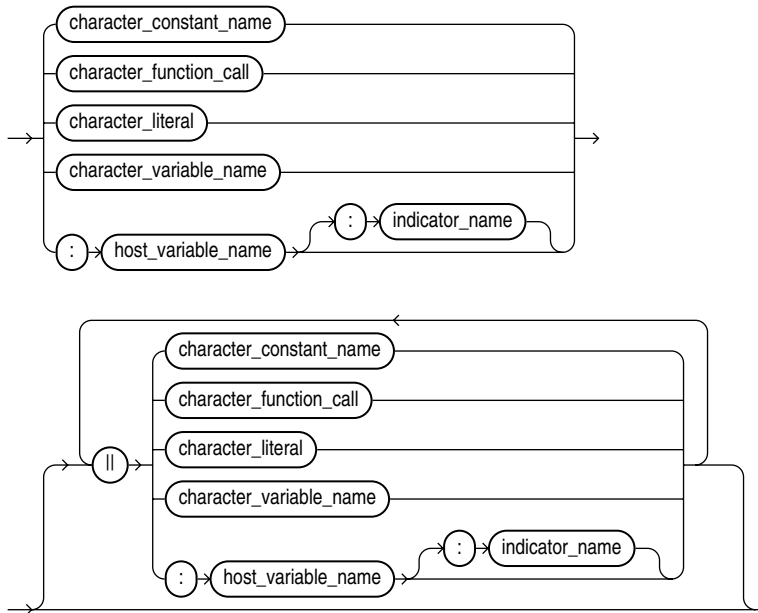
boolean_expression ::=



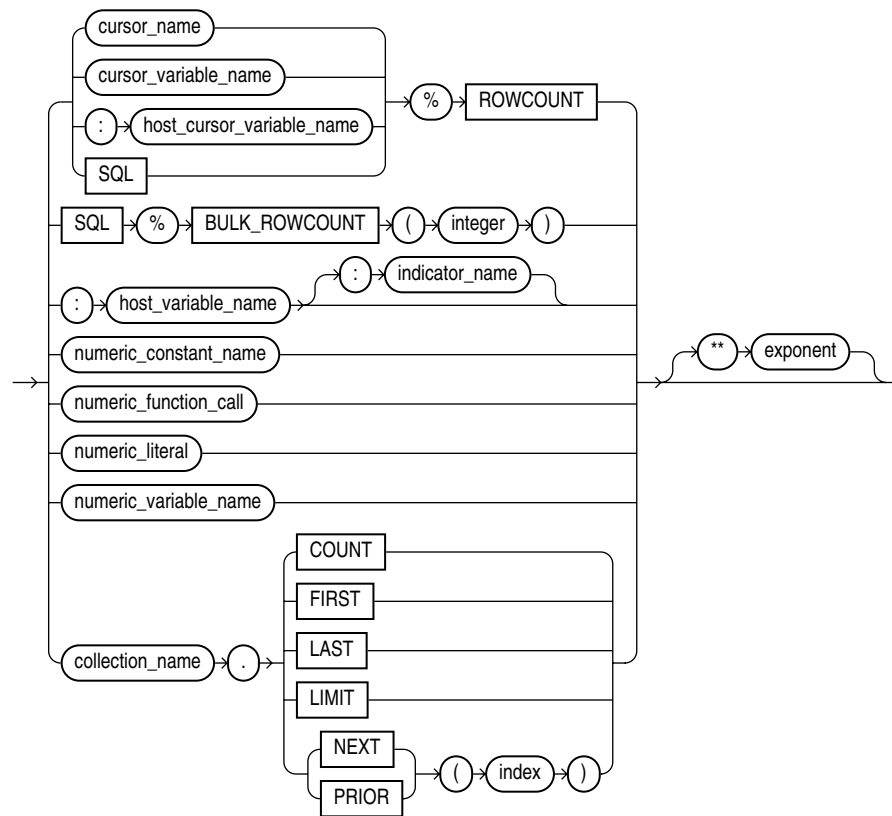
other_boolean_form ::=



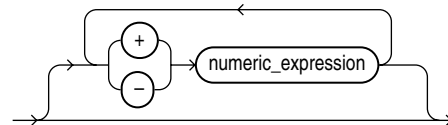
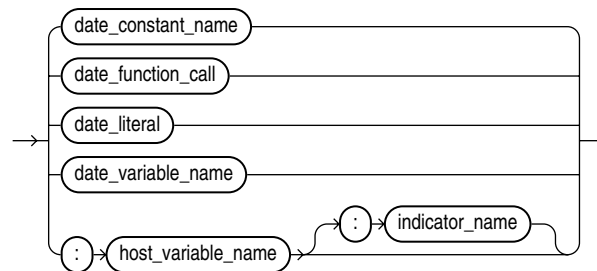
character_expression ::=



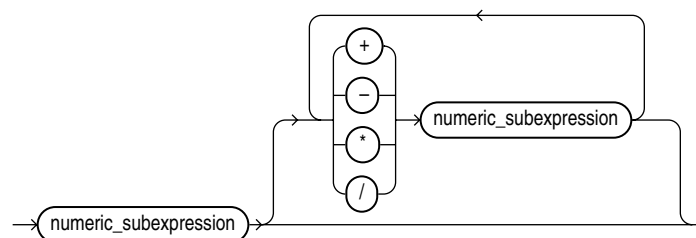
numeric_subexpression ::=



date_expression ::=



numeric_expression ::=



Keyword and Parameter Description

BETWEEN

This comparison operator tests whether a value lies in a specified range. It means: greater than or equal to low value and less than or equal to high value.

boolean_constant_name

A constant of type `BOOLEAN`, which must be initialized to the value `TRUE`, `FALSE`, or `NULL`. Arithmetic operations on Boolean constants are not allowed.

boolean_expression

An expression that returns the Boolean value `TRUE`, `FALSE`, or `NULL`.

boolean_function_call

Any function call that returns a Boolean value.

boolean_literal

The predefined values `TRUE`, `FALSE`, or `NULL` (which stands for a missing, unknown, or inapplicable value). You cannot insert the value `TRUE` or `FALSE` into a database column.

boolean_variable_name

A variable of type `BOOLEAN`. Only the values `TRUE`, `FALSE`, and `NULL` can be assigned to a `BOOLEAN` variable. You cannot select or fetch column values into a `BOOLEAN` variable. Also, arithmetic operations on `BOOLEAN` variables are not allowed.

%BULK_ROWCOUNT

Designed for use with the `FORALL` statement, this is a composite attribute of the implicit cursor `SQL`. For more information, see "[SQL Cursor](#)" on page 13-136.

character_constant_name

A previously declared constant that stores a character value. It must be initialized to a character value or a value implicitly convertible to a character value.

character_expression

An expression that returns a character or character string.

character_function_call

A function call that returns a character value or a value implicitly convertible to a character value.

character_literal

A literal that represents a character value or a value implicitly convertible to a character value.

character_variable_name

A previously declared variable that stores a character value.

collection_name

A collection (nested table, index-by table, or varray) previously declared within the current scope.

cursor_name

An explicit cursor previously declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable previously declared within the current scope.

date_constant_name

A previously declared constant that stores a date value. It must be initialized to a date value or a value implicitly convertible to a date value.

date_expression

An expression that returns a date/time value.

date_function_call

A function call that returns a date value or a value implicitly convertible to a date value.

date_literal

A literal representing a date value or a value implicitly convertible to a date value.

date_variable_name

A previously declared variable that stores a date value.

EXISTS, COUNT, FIRST, LAST, LIMIT, NEXT, PRIOR

Collection methods. When appended to the name of a collection, these methods return useful information. For example, `EXISTS (n)` returns `TRUE` if the *n*th element of a collection exists. Otherwise, `EXISTS (n)` returns `FALSE`. For more information, see "[Collection Methods](#)" on page 13-23.

exponent

An expression that must return a numeric value.

%FOUND, %ISOPEN, %NOTFOUND, %ROWCOUNT

Cursor attributes. When appended to the name of a cursor or cursor variable, these attributes return useful information about the execution of a multiple-row query. You can also append them to the implicit cursor `SQL`.

host_cursor_variable_name

A cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. Host cursor variables must be prefixed with a colon.

host_variable_name

A variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. The datatype of the host variable must be implicitly convertible to the appropriate PL/SQL datatype. Also, host variables must be prefixed with a colon.

IN

Comparison operator that tests set membership. It means: equal to any member of. The set can contain nulls, but they are ignored. Also, expressions of the form

value NOT IN set

return FALSE if the set contains a null.

index

A numeric expression that must return a value of type `BINARY_INTEGER`, `PLS_INTEGER`, or a value implicitly convertible to that datatype.

indicator_name

An indicator variable declared in a PL/SQL host environment and passed to PL/SQL. Indicator variables must be prefixed with a colon. An indicator variable indicates the value or condition of its associated host variable. For example, in the Oracle Precompiler environment, indicator variables can detect nulls or truncated values in output host variables.

IS NULL

Comparison operator that returns the Boolean value TRUE if its operand is null, or FALSE if its operand is not null.

LIKE

Comparison operator that compares a character value to a pattern. Case is significant. LIKE returns the Boolean value TRUE if the character patterns match, or FALSE if they do not match.

NOT, AND, OR

Logical operators, which follow the tri-state logic of [Table 2-3](#) on page 2-22. AND returns the value TRUE only if both its operands are true. OR returns the value TRUE if either of its operands is true. NOT returns the opposite value (logical negation) of its operand. For more information, see "[Logical Operators](#)" on page 2-22.

NULL

Keyword that represents a null. It stands for a missing, unknown, or inapplicable value. When NULL is used in a numeric or date expression, the result is a null.

numeric_constant_name

A previously declared constant that stores a numeric value. It must be initialized to a numeric value or a value implicitly convertible to a numeric value.

numeric_expression

An expression that returns an integer or real value.

numeric_function_call

A function call that returns a numeric value or a value implicitly convertible to a numeric value.

numeric_literal

A literal that represents a number or a value implicitly convertible to a number.

numeric_variable_name

A previously declared variable that stores a numeric value.

pattern

A character string compared by the `LIKE` operator to a specified string value. It can include two special-purpose characters called wildcards. An underscore (`_`) matches exactly one character; a percent sign (`%`) matches zero or more characters. The pattern can be followed by `ESCAPE 'character_literal'`, which turns off wildcard expansion wherever the escape character appears in the string followed by a percent sign or underscore.

relational_operator

Operator that compares expressions. For the meaning of each operator, see "[Comparison Operators](#)" on page 2-24.

SQL

A cursor opened implicitly by Oracle to process a SQL data manipulation statement. The implicit cursor `SQL` always refers to the most recently executed SQL statement.

+, -, /, *, **

Symbols for the addition, subtraction, division, multiplication, and exponentiation operators.

||

The concatenation operator. As the following example shows, the result of concatenating *string1* with *string2* is a character string that contains *string1* followed by *string2*:

```
'Good' || ' morning!' = 'Good morning!'
```

The next example shows that nulls have no effect on the result of a concatenation:

```
'suit' || NULL || 'case' = 'suitcase'
```

A null string (`' '`), which is zero characters in length, is treated like a null.

Usage Notes

In a Boolean expression, you can only compare values that have compatible datatypes. For more information, see "[PL/SQL Datatype Conversion](#)" on page 3-26.

In conditional control statements, if a Boolean expression returns `TRUE`, its associated sequence of statements is executed. But, if the expression returns `FALSE` or `NULL`, its associated sequence of statements is *not* executed.

The relational operators can be applied to operands of type `BOOLEAN`. By definition, `TRUE` is greater than `FALSE`. Comparisons involving nulls always return a null. The value of a Boolean expression can be assigned only to Boolean variables, not to host variables or database columns. Also, datatype conversion to or from type `BOOLEAN` is not supported.

You can use the addition and subtraction operators to increment or decrement a date value, as the following examples show:

```
hire_date := '10-MAY-95';
hire_date := hire_date + 1; -- makes hire_date '11-MAY-95'
hire_date := hire_date - 5; -- makes hire_date '06-MAY-95'
```

When PL/SQL evaluates a boolean expression, NOT has the highest precedence, AND has the next-highest precedence, and OR has the lowest precedence. However, you can use parentheses to override the default operator precedence.

Within an expression, operations occur in the following order (first to last):

1. Parentheses
2. Exponents
3. Unary operators
4. Multiplication and division
5. Addition, subtraction, and concatenation

PL/SQL evaluates operators of equal precedence in no particular order. When parentheses enclose an expression that is part of a larger expression, PL/SQL evaluates the parenthesized expression first, then uses the result in the larger expression. When parenthesized expressions are nested, PL/SQL evaluates the innermost expression first and the outermost expression last.

Examples

```
(a + b) > c           -- Boolean expression
NOT finished         -- Boolean expression
TO_CHAR(acct_no)     -- character expression
'Fat ' || 'cats'     -- character expression
'15-NOV-05'          -- date expression
MONTHS_BETWEEN(d1, d2) -- date expression
pi * r**2            -- numeric expression
emp_cv%ROWCOUNT    -- numeric expression
```

Related Topics

[Example 1–3, "Assigning Values to Variables with the Assignment Operator"](#) on page 1-6

["PL/SQL Expressions and Comparisons"](#) on page 2-21

["Assignment Statement"](#) on page 13-3

["Constant and Variable Declaration"](#) on page 13-29

["EXIT Statement"](#) on page 13-55 ["IF Statement"](#) on page 13-80

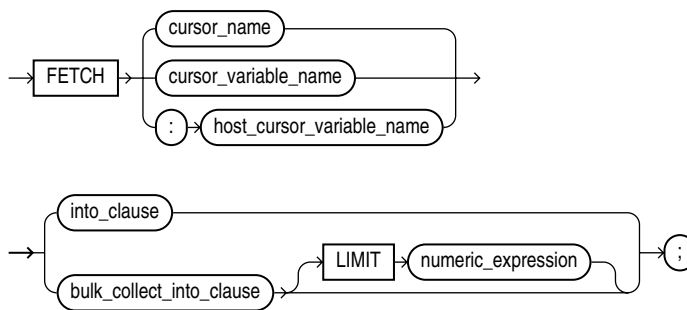
["LOOP Statements"](#) on page 13-93

FETCH Statement

The `FETCH` statement retrieves rows of data from the result set of a multiple-row query. You can fetch rows one at a time, several at a time, or all at once. The data is stored in variables or fields that correspond to the columns selected by the query. For more information, see ["Querying Data with PL/SQL"](#) on page 6-16.

Syntax

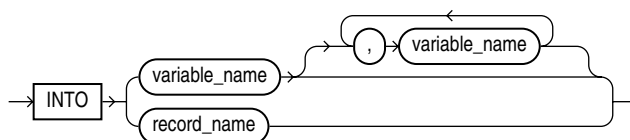
fetch_statement ::=



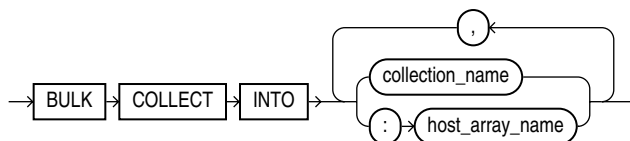
into_clause

bulk_collect_into_clause

into_clause ::=



bulk_collect_into_clause ::=



Keyword and Parameter Description

BULK COLLECT INTO

Instructs the SQL engine to bulk-bind output collections before returning them to the PL/SQL engine. The SQL engine bulk-binds all collections referenced in the `INTO` list.

collection_name

The name of a declared collection into which column values are bulk fetched. For each query `select_item`, there must be a corresponding, type-compatible collection in the list.

cursor_name

An explicit cursor declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable (or parameter) declared within the current scope.

host_array_name

An array (declared in a PL/SQL host environment and passed to PL/SQL as a bind argument) into which column values are bulk fetched. For each query `select_item`, there must be a corresponding, type-compatible array in the list. Host arrays must be prefixed with a colon.

host_cursor_variable_name

A cursor variable declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. The datatype of the host cursor variable is compatible with the return type of any PL/SQL cursor variable. Host variables must be prefixed with a colon.

LIMIT

This optional clause, allowed only in bulk (not scalar) `FETCH` statements, lets you bulk fetch several rows at a time, rather than the entire result set.

record_name

A user-defined or `%ROWTYPE` record into which rows of values are fetched. For each column value returned by the query associated with the cursor or cursor variable, there must be a corresponding, type-compatible field in the record.

variable_name

A variable into which a column value is fetched. For each column value returned by the query associated with the cursor or cursor variable, there must be a corresponding, type-compatible variable in the list.

Usage Notes

You must use either a cursor `FOR` loop or the `FETCH` statement to process a multiple-row query.

Any variables in the `WHERE` clause of the query are evaluated only when the cursor or cursor variable is opened. To change the result set or the values of variables in the query, you must reopen the cursor or cursor variable with the variables set to their new values.

To reopen a cursor, you must close it first. However, you need not close a cursor variable before reopening it.

You can use different `INTO` lists on separate fetches with the same cursor or cursor variable. Each fetch retrieves another row and assigns values to the target variables.

If you `FETCH` past the last row in the result set, the values of the target fields or variables are indeterminate and the `%NOTFOUND` attribute returns `TRUE`.

PL/SQL makes sure the return type of a cursor variable is compatible with the `INTO` clause of the `FETCH` statement. For each column value returned by the query associated with the cursor variable, there must be a corresponding, type-compatible field or variable in the `INTO` clause. Also, the number of fields or variables must equal the number of column values.

When you declare a cursor variable as the formal parameter of a subprogram that fetches from the cursor variable, you must specify the `IN` or `IN OUT` mode. However, if the subprogram also opens the cursor variable, you must specify the `IN OUT` mode.

Because a sequence of `FETCH` statements always runs out of data to retrieve, no exception is raised when a `FETCH` returns no data. To detect this condition, you must use the cursor attribute `%FOUND` or `%NOTFOUND`.

PL/SQL raises the predefined exception `INVALID_CURSOR` if you try to fetch from a closed or never-opened cursor or cursor variable.

Restrictions on `BULK COLLECT INTO`

The following restrictions apply to the `BULK COLLECT INTO` clause:

- You cannot bulk collect into an associative array that has a string type for the key.
- You can use the `BULK COLLECT INTO` clause only in server-side programs (not in client-side programs). Otherwise, you get the following error:

```
this feature is not supported in client-side programs
```

- All target variables listed in a `BULK COLLECT INTO` clause must be collections.
- Composite targets (such as objects) cannot be used in the `RETURNING INTO` clause. Otherwise, you get the following error:

```
error unsupported feature with RETURNING clause
```
- When implicit datatype conversions are needed, multiple composite targets cannot be used in the `BULK COLLECT INTO` clause.
- When an implicit datatype conversion is needed, a collection of a composite target (such as a collection of objects) cannot be used in the `BULK COLLECT INTO` clause.

Examples

[Example 6–10, "Fetching with a Cursor" on page 6-11](#)

[Example 6–13, "Fetching Bulk Data with a Cursor" on page 6-12](#)

[Example 6–23, "Passing Parameters to Explicit Cursors" on page 6-21](#)

[Example 6–27, "Passing a REF CURSOR as a Parameter" on page 6-24](#)

[Example 6–32, "Fetching from a Cursor Variable into a Record" on page 6-28](#)

[Example 6–33, "Fetching from a Cursor Variable into Collections" on page 6-28](#)

[Example 6–35, "Using a Cursor Expression" on page 6-32](#)

[Example 6–41, "Using CURRENT OF to Update the Latest Row Fetched from a Cursor" on page 6-38](#)

Related Topics

["CLOSE Statement" on page 13-18,](#)

["Cursor Declaration" on page 13-41](#)

["Cursor Variables" on page 13-37](#)

["LOOP Statements" on page 13-93](#)

["OPEN Statement" on page 13-102](#)

["OPEN-FOR Statement" on page 13-104](#)

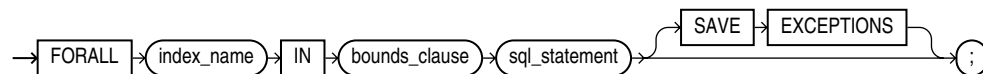
["RETURNING INTO Clause" on page 13-121](#)

FORALL Statement

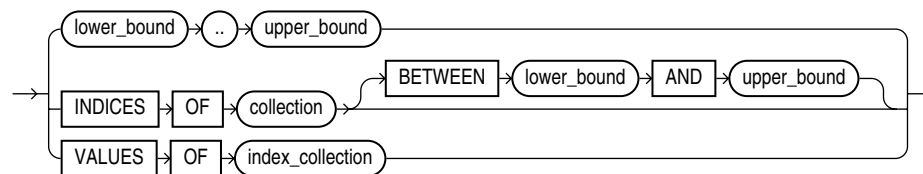
The `FORALL` statement issues a series of static or dynamic DML statements, usually much faster than an equivalent `FOR` loop. It requires some setup code, because each iteration of the loop must use values from one or more collections in its `VALUES` or `WHERE` clauses. For more information, see ["Reducing Loop Overhead for DML Statements and Queries with Bulk SQL"](#) on page 12-11.

Syntax

forall_statement ::=



bounds_clause ::=



Keyword and Parameter Description

INDICES OF collection_name

A clause specifying that the values of the index variable correspond to the subscripts of the elements of the specified collection. With this clause, you can use `FORALL` with nested tables where some elements were deleted, or with associative arrays that have numeric subscripts.

BETWEEN lower_bound AND upper_bound

Limits the range of subscripts in the `INDICES OF` clause. If a subscript in the range does not exist in the collection, that subscript is skipped.

VALUES OF index_collection_name

A clause specifying that the subscripts for the `FORALL` index variable are taken from the values of the elements in another collection, specified by `index_collection_name`. This other collection acts as a set of pointers; `FORALL` can iterate through subscripts in arbitrary order, even using the same subscript more than once, depending on what elements you include in `index_collection_name`.

The index collection must be a nested table, or an associative array indexed by `PLS_INTEGER` or `BINARY_INTEGER`, whose elements are also `PLS_INTEGER` or `BINARY_INTEGER`. If the index collection is empty, an exception is raised and the `FORALL` statement is not executed.

index_name

An undeclared identifier that can be referenced only within the `FORALL` statement and only as a collection subscript.

The implicit declaration of `index_name` overrides any other declaration outside the loop. You cannot refer to another variable with the same name inside the statement. Inside a `FORALL` statement, `index_name` cannot appear in expressions and cannot be assigned a value.

lower_bound .. upper_bound

Numeric expressions that specify a valid range of consecutive index numbers. PL/SQL rounds them to the nearest integer, if necessary. The SQL engine executes the SQL statement once for each index number in the range. The expressions are evaluated once, when the `FORALL` statement is entered.

SAVE EXCEPTIONS

Optional keywords that cause the `FORALL` loop to continue even if some DML operations fail. Instead of raising an exception immediately, the program raises a single exception after the `FORALL` statement finishes. The details of the errors are available after the loop in `SQL%BULK_EXCEPTIONS`. The program can report or clean up all the errors after the `FORALL` loop, rather than handling each exception as it happens. See ["Handling FORALL Exceptions \(%BULK_EXCEPTIONS Attribute\)"](#) on page 12-17.

sql_statement

A static, such as `UPDATE` or `DELETE`, or dynamic (`EXECUTE IMMEDIATE`) DML statement that references collection elements in the `VALUES` or `WHERE` clauses.

Usage Notes

Although the SQL statement can reference more than one collection, the performance benefits apply only to subscripted collections.

If *sql-statement* is an `INSERT` statement that fires a compound trigger, each iteration of the `FORALL` loop fires the compound trigger anew, destroying its performance benefit. For information about compound triggers, see ["Compound Triggers"](#) on page 9-12.

If a `FORALL` statement fails, database changes are rolled back to an implicit savepoint marked before each execution of the SQL statement. Changes made during previous iterations of the `FORALL` loop are not rolled back.

Restrictions

The following restrictions apply to the `FORALL` statement:

- You cannot loop through the elements of an associative array that has a string type for the key.
- Within a `FORALL` loop, you cannot refer to the same collection in both the `SET` clause and the `WHERE` clause of an `UPDATE` statement. You might need to make a second copy of the collection and refer to the new name in the `WHERE` clause.
- You can use the `FORALL` statement only in server-side programs, not in client-side programs.
- The `INSERT`, `UPDATE`, or `DELETE` statement must reference at least one collection. For example, a `FORALL` statement that inserts a set of constant values in a loop raises an exception.
- When you specify an explicit range, all collection elements in that range must exist. If an element is missing or was deleted, you get an error.

- When you use the `INDICES OF` or `VALUES OF` clauses, all the collections referenced in the DML statement must have subscripts matching the values of the index variable. Make sure that any `DELETE`, `EXTEND`, and so on operations are applied to all the collections so that they have the same set of subscripts. If any of the collections is missing a referenced element, you get an error. If you use the `SAVE EXCEPTIONS` clause, this error is treated like any other error and does not stop the `FORALL` statement.
- You cannot refer to individual record fields within DML statements invoked by a `FORALL` statement. Instead, you can specify the entire record with the `SET ROW` clause in an `UPDATE` statement, or the `VALUES` clause in an `INSERT` statement.
- Collection subscripts must be just the index variable rather than an expression, such as `i` rather than `i+1`.
- The cursor attribute `%BULK_ROWCOUNT` cannot be assigned to other collections, or be passed as a parameter to subprograms.
- If the `FORALL` uses a dynamic SQL statement, then values (binds for the dynamic SQL statement) in the `USING` clause must be simple references to the collection, not expressions. For example, `collection_name(i)` is valid, but `UPPER(collection_name(i))` is not valid.

Examples

[Example 12-2, "Issuing DELETE Statements in a Loop"](#) on page 12-12

[Example 12-3, "Issuing INSERT Statements in a Loop"](#) on page 12-12

[Example 12-4, "Using FORALL with Part of a Collection"](#) on page 12-13

[Example 12-5, "Using FORALL with Nonconsecutive Index Values"](#) on page 12-13

[Example 12-9, "Bulk Operation that Continues Despite Exceptions"](#) on page 12-18

[Example 12-16, "Using FORALL with BULK COLLECT"](#) on page 12-23

Related Topics

["Retrieving Query Results into Collections \(BULK COLLECT Clause\)"](#) on page 12-19

Function Declaration and Definition

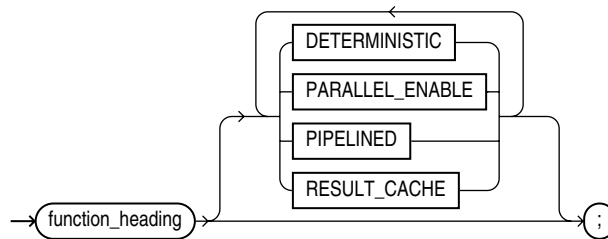
A function is a subprogram that returns a single value. A PL/SQL block or parent subprogram must declare and define a function before invoking it. The declaration always includes the specification ("spec"). The declaration can also include the definition. If the declaration does not include the definition, the definition must appear later in the same block or subprogram as the declaration.

For more information about functions, see ["What Are PL/SQL Subprograms?"](#) on page 8-1.

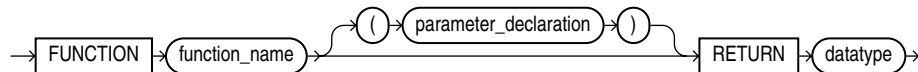
Note: Declaring and defining a function in a PL/SQL block or package is different from creating a function with the SQL statement `CREATE FUNCTION`. For information about `CREATE FUNCTION`, see *Oracle Database SQL Language Reference*.

Syntax

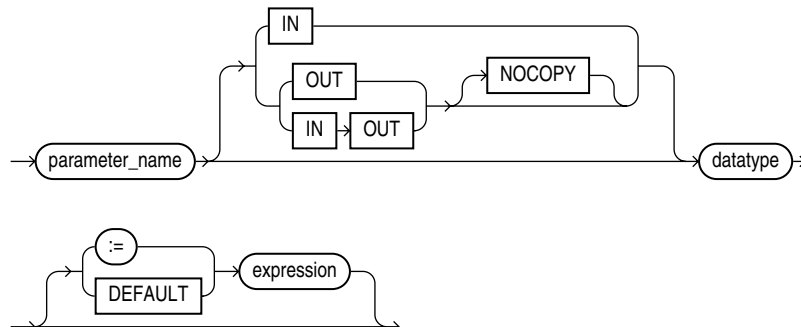
function_declaration ::=

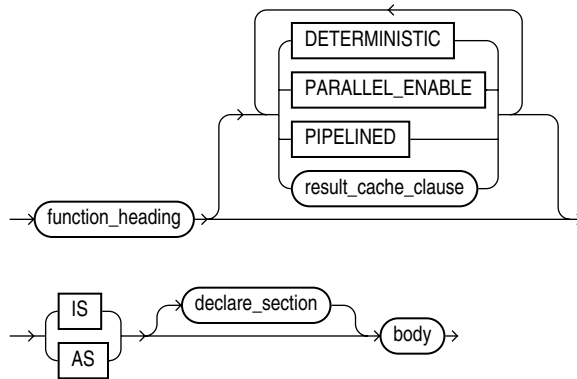
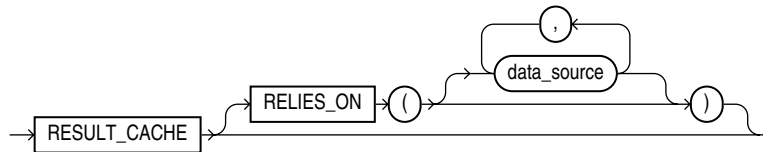


function_heading ::=



parameter_declaration ::=



function_definition ::=**result_cache_clause ::=****Keyword and Parameter Description****body**

For syntax, see ["Block Declaration"](#) on page 13-8.

datatype

A type specifier. For syntax, see ["Constant and Variable Declaration"](#) on page 13-29.

You cannot constrain (with `NOT NULL` for example) the datatype of a function parameter or a function return value.

data_source

The name of either a database table or a database view.

declare_section

Declares function elements. For syntax, see ["Block Declaration"](#) on page 13-8.

DETERMINISTIC

Specify `DETERMINISTIC` to indicate that the function returns the same result value whenever it is invoked with the same values for its parameters. This helps the optimizer avoid redundant function calls: If a stored function was invoked previously with the same arguments, the optimizer can elect to use the previous result.

Do not specify `DETERMINISTIC` for a function whose result depends on the state of session variables or schema objects, because results might vary across calls. Instead, consider making the function result-cached (see ["Making Result-Cached Functions Handle Session-Specific Settings"](#) on page 8-35 and ["Making Result-Cached Functions Handle Session-Specific Application Contexts"](#) on page 8-36).

Only `DETERMINISTIC` functions can be invoked from a function-based index or a materialized view that has query-rewrite enabled. For more information and possible

limitations of the DETERMINISTIC option, see the CREATE FUNCTION statement in the *Oracle Database SQL Language Reference*.

See Also: CREATE INDEX statement in *Oracle Database SQL Language Reference*

expression

An arbitrarily complex combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the parameter. The value and the parameter must have compatible datatypes. For syntax, see "[Expression Definition](#)" on page 13-57.

Note: If a function call includes an actual parameter for `parameter_name`, then `expression` is not evaluated for that function call (see [Example 8-8](#)).

function_name

The name you choose for the function.

IN, OUT, IN OUT

Parameter modes that define the action of formal parameters. An IN parameter passes values to the subprogram being invoked. An OUT parameter returns values to the invoker of the subprogram. An IN OUT parameter passes initial values to the subprogram being invoked, and returns updated values to the invoker.

NOCOPY

A compiler hint (not directive) that allows the PL/SQL compiler to pass OUT and IN OUT parameters by reference instead of by value (the default). The function can run faster, because it does not have to make temporary copies of these parameters, but the results can be different if the function ends with an unhandled exception. For more information, see "[Using Default Values for Subprogram Parameters](#)" on page 8-10.

PARALLEL_ENABLE

Declares that a stored function can be used safely in the slave sessions of parallel DML evaluations. The state of a main (logon) session is never shared with slave sessions. Each slave session has its own state, which is initialized when the session begins. The function result must not depend on the state of session (`static`) variables. Otherwise, results might vary across sessions. For information on the PARALLEL_ENABLE option, see the CREATE FUNCTION statement in the *Oracle Database SQL Language Reference*.

parameter_name

A formal parameter, a variable declared in a function spec and referenced in the function body.

PIPELINED

PIPELINED specifies to return the results of a table function iteratively. A table function returns a collection type (a nested table or varray) with elements that are SQL datatypes. You can query table functions using the TABLE keyword before the function name in the FROM clause of a SQL query. For more information, see "[Performing Multiple Transformations with Pipelined Table Functions](#)" on page 12-31.

procedure_declaration

Declares, and might also define, a procedure. If the declaration does not define the procedure, the definition must appear later in the same block or subprogram as the declaration. See ["Procedure Declaration and Definition"](#) on page 13-110.

RELIES_ON

Specifies the data sources on which the results of a function depend. For more information, see ["Using the Cross-Session PL/SQL Function Result Cache"](#) on page 8-29.

RESULT_CACHE

Causes the results of the function to be cached. For more information, see ["Using the Cross-Session PL/SQL Function Result Cache"](#) on page 8-29.

RETURN

Introduces the RETURN clause, which specifies the datatype of the return value.

statement

A statement. For syntax, see ["Block Declaration"](#) on page 13-8.

type_definition

Specifies a user-defined datatype. For syntax, see ["Block Declaration"](#) on page 13-8.

view_name

The name of a database view.

:= | DEFAULT

Initializes IN parameters to default values.

Usage Notes

A function is invoked as part of an expression. For example:

```
promotable := sal_ok(new_sal, new_title) AND (rating > 3);
```

To be callable from SQL statements, a stored function must obey certain rules that control side effects. See ["Controlling Side Effects of PL/SQL Subprograms"](#) on page 8-26.

In a function, at least one execution path must lead to a RETURN statement. Otherwise, you get the following run-time error:

```
function returned without value
```

The RETURN statement must contain an expression, which is evaluated when the RETURN statement is executed. The resulting value is assigned to the function identifier, which acts like a variable.

You can write the function spec and body as a unit. Or, you can separate the function spec from its body. That way, you can hide implementation details by placing the function in a package. You can define functions in a package body without declaring their specs in the package spec. However, such functions can be invoked only from inside the package.

Inside a function, an `IN` parameter acts like a constant; you cannot assign it a value. An `OUT` parameter acts like a local variable; you can change its value and reference the value in any way. An `IN OUT` parameter acts like an initialized variable; you can assign it a value, which can be assigned to another variable. For information about the parameter modes, see [Table 8-3](#) on page 8-10.

Avoid using the `OUT` and `IN OUT` modes with functions. The purpose of a function is to take zero or more parameters and return a single value. Functions must be free from side effects, which change the values of variables not local to the subprogram.

Examples

[Example 1-14, "Creating a Package and Package Body"](#) on page 1-15

[Example 2-15, "Using a Subprogram Name for Name Resolution"](#) on page 2-16

[Example 2-27, "Using a Search Condition with a CASE Statement"](#) on page 2-29

[Example 5-44, "Returning a Record from a Function"](#) on page 5-33

[Example 6-43, "Declaring an Autonomous Function in a Package"](#) on page 6-42

[Example 6-48, "Invoking an Autonomous Function"](#) on page 6-46

[Example 10-3, "Creating the emp_admin Package"](#) on page 10-6

Related Topics

["Using the Cross-Session PL/SQL Function Result Cache"](#) on page 8-29

["Package Declaration"](#) on page 13-107

["Procedure Declaration and Definition"](#) on page 13-110

GOTO Statement

The `GOTO` statement branches unconditionally to a statement label or block label. The label must be unique within its scope and must precede an executable statement or a PL/SQL block. The `GOTO` statement transfers control to the labelled statement or block. For more information, see ["Using the GOTO Statement"](#) on page 4-16.

Syntax

label_declaration ::=

→ (<< label_name >>) →

goto_statement ::=

→ GOTO label_name ;

Keyword and Parameter Description

label_name

A label that you assigned to an executable statement or a PL/SQL block. A `GOTO` statement transfers control to the statement or block following `<<label_name>>`.

Usage Notes

A `GOTO` label must precede an executable statement or a PL/SQL block. A `GOTO` statement cannot branch into an `IF` statement, `LOOP` statement, or sub-block. To branch to a place that does not have an executable statement, add the `NULL` statement.

From the current block, a `GOTO` statement can branch to another place in the block or into an enclosing block, but not into an exception handler. From an exception handler, a `GOTO` statement can branch into an enclosing block, but not into the current block.

If you use the `GOTO` statement to exit a cursor `FOR` loop prematurely, the cursor is closed automatically. The cursor is also closed automatically if an exception is raised inside the loop.

A given label can appear only once in a block. However, the label can appear in other blocks including enclosing blocks and sub-blocks. If a `GOTO` statement cannot find its target label in the current block, it branches to the first enclosing block in which the label appears.

Examples

[Example 4-22, "Using a Simple GOTO Statement"](#) on page 4-16

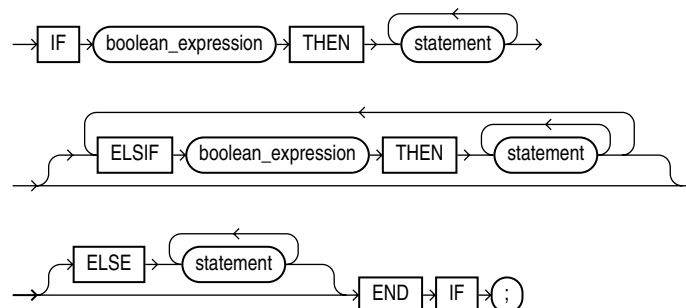
[Example 4-24, "Using a GOTO Statement to Branch an Enclosing Block"](#) on page 4-17

IF Statement

The **IF** statement executes or skips a sequence of statements, depending on the value of a Boolean expression. For more information, see "[Testing Conditions \(IF and CASE Statements\)](#)" on page 4-2.

Syntax

if_statement ::=



Keyword and Parameter Description

boolean_expression

An expression that returns the Boolean value **TRUE**, **FALSE**, or **NULL**. Examples are comparisons for equality, greater-than, or less-than. The sequence following the **THEN** keyword is executed only if the expression returns **TRUE**.

ELSE

If control reaches this keyword, the sequence of statements that follows it is executed. This occurs when none of the previous conditional tests returned **TRUE**.

ELSIF

Introduces a Boolean expression that is evaluated if none of the preceding conditions returned **TRUE**.

THEN

If the expression returns **TRUE**, the statements after the **THEN** keyword are executed.

Usage Notes

There are three forms of **IF** statements: **IF-THEN**, **IF-THEN-ELSE**, and **IF-THEN-ELSIF**. The simplest form of **IF** statement associates a Boolean expression with a sequence of statements enclosed by the keywords **THEN** and **END IF**. The sequence of statements is executed only if the expression returns **TRUE**. If the expression returns **FALSE** or **NULL**, the **IF** statement does nothing. In either case, control passes to the next statement.

The second form of **IF** statement adds the keyword **ELSE** followed by an alternative sequence of statements. The sequence of statements in the **ELSE** clause is executed only if the Boolean expression returns **FALSE** or **NULL**. Thus, the **ELSE** clause ensures that a sequence of statements is executed.

The third form of IF statement uses the keyword `ELSIF` to introduce additional Boolean expressions. If the first expression returns `FALSE` or `NULL`, the `ELSIF` clause evaluates another expression. An IF statement can have any number of `ELSIF` clauses; the final `ELSE` clause is optional. Boolean expressions are evaluated one by one from top to bottom. If any expression returns `TRUE`, its associated sequence of statements is executed and control passes to the next statement. If all expressions return `FALSE` or `NULL`, the sequence in the `ELSE` clause is executed.

An IF statement never executes more than one sequence of statements because processing is complete after any sequence of statements is executed. However, the `THEN` and `ELSE` clauses can include more IF statements. That is, IF statements can be nested.

Examples

[Example 1–8, "Using the IF-THEN_ELSE and CASE Statement for Conditional Control"](#) on page 1-10

[Example 4–1, "Using a Simple IF-THEN Statement"](#) on page 4-2

[Example 4–2, "Using a Simple IF-THEN-ELSE Statement"](#) on page 4-3

[Example 4–3, "Nested IF Statements"](#) on page 4-3

[Example 4–4, "Using the IF-THEN-ELSEIF Statement"](#) on page 4-4

Related Topics

["Testing Conditions \(IF and CASE Statements\)"](#) on page 4-2

["CASE Statement"](#) on page 13-16

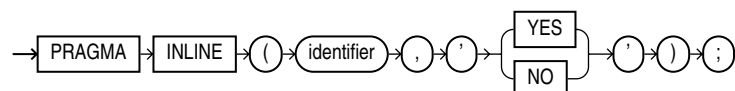
["Expression Definition"](#) on page 13-57

INLINE Pragma

The `INLINE` pragma specifies that a subprogram call is, or is not, to be inlined. Inlining replaces a subprogram call (to a subprogram in the same program unit) with a copy of the called subprogram. For more information, see ["How PL/SQL Optimizes Your Programs"](#) on page 12-3.

Syntax

inline_pragma ::=



Keyword and Parameter Descriptions

identifier

The name of a subprogram.

YES

If `PLSQL_OPTIMIZE_LEVEL=2`, `YES` specifies that the subprogram call is to be inlined.

If `PLSQL_OPTIMIZE_LEVEL=3`, `YES` specifies that the subprogram call has a high priority for inlining.

NO

Specifies that the subprogram call is not to be inlined.

Usage Notes

The `INLINE` pragma affects only the immediately following declaration or statement, and only some kinds of statements.

When the `INLINE` pragma immediately precedes one of the following statements, the pragma affects every call to the specified subprogram in that statement (see [Example 13-1](#)):

- Assignment
- Call
- Conditional
- CASE
- CONTINUE-WHEN
- EXECUTE IMMEDIATE
- EXIT-WHEN
- LOOP
- RETURN

The `INLINE` pragma does not affect statements that are not in the preceding list.

When the `INLINE` pragma immediately precedes a declaration, it affects the following:

- Every call to the specified subprogram in that declaration
- Every initialization value in that declaration except the default initialization values of records

If the name of the subprogram (*identifier*) is overloaded (that is, if it belongs to more than one subprogram), the `INLINE` pragma applies to every subprogram with that name (see [Example 13–2](#)). For information about overloaded subprogram names, see "[Overloading PL/SQL Subprogram Names](#)" on page 8-12.

The `PRAGMA INLINE (identifier, 'YES')` very strongly encourages the compiler to inline a particular call, but the compiler might not do so if other considerations or limits make the inlining undesirable. If you specify `PRAGMA INLINE (identifier, 'NO')`, the compiler does not inline calls to subprograms named *identifier* (see [Example 13–3](#)).

Multiple pragmas can affect the same declaration or statement. Each pragma applies its own effect to the statement. If `PRAGMA INLINE(identifier, 'YES')` and `PRAGMA INLINE (identifier, 'NO')` have the same *identifier*, 'NO' overrides 'YES' (see [Example 13–4](#)). One `PRAGMA INLINE (identifier, 'NO')` overrides any number of occurrences of `PRAGMA INLINE (identifier, 'YES')`, and the order of these pragmas is not important.

Examples

In [Example 13–1](#) and [Example 13–2](#), assume that `PLSQL_OPTIMIZE_LEVEL=2`.

In [Example 13–1](#), the `INLINE` pragma affects the procedure calls `p1(1)` and `p1(2)`, but not the procedure calls `p1(3)` and `p1(4)`.

Example 13–1 Specifying that a Subprogram Is To Be Inlined

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, 'YES');
x:= p1(1) + p1(2) + 17;    -- These 2 calls to p1 will be inlined
...
x:= p1(3) + p1(4) + 17;    -- These 2 calls to p1 will not be inlined
...
```

In [Example 13–2](#) the `INLINE` pragma affects both functions named `p2`.

Example 13–2 Specifying that an Overloaded Subprogram Is To Be Inlined

```
FUNCTION p2 (p boolean) return PLS_INTEGER IS ...
FUNCTION p2 (x PLS_INTEGER) return PLS_INTEGER IS ...
...
PRAGMA INLINE(p2, 'YES');
x := p2(true) + p2(3);
...
```

In [Example 13–3](#), assume that `PLSQL_OPTIMIZE_LEVEL=3`. The `INLINE` pragma affects the procedure calls `p1(1)` and `p1(2)`, but not the procedure calls `p1(3)` and `p1(4)`.

Example 13–3 Specifying that a Subprogram Is Not To Be Inlined

```
PROCEDURE p1 (x PLS_INTEGER) IS ...
...
```



```

PRAGMA INLINE (p1, 'NO');
x:= p1(1) + p1(2) + 17;    -- These 2 calls to p1 will not be inlined
...
x:= p1(3) + p1(4) + 17;    -- These 2 calls to p1 might be inlined
...

```

In [Example 13–4](#), the first `INLINE` pragma affects the two calls to procedure `p1`, but then the second `INLINE` pragma overrides it.

Example 13–4 Applying Two `INLINE` Pragmas to the Same Subprogram

```

PROCEDURE p1 (x PLS_INTEGER) IS ...
...
PRAGMA INLINE (p1, PRAGMA INLINE (p1, 'YES'));
PRAGMA INLINE (p1, PRAGMA INLINE (p1, 'NO'));
x:= p1(1) + p1(2) + 17;    -- These 2 calls to p1 will not be inlined
...

```

Related Topics

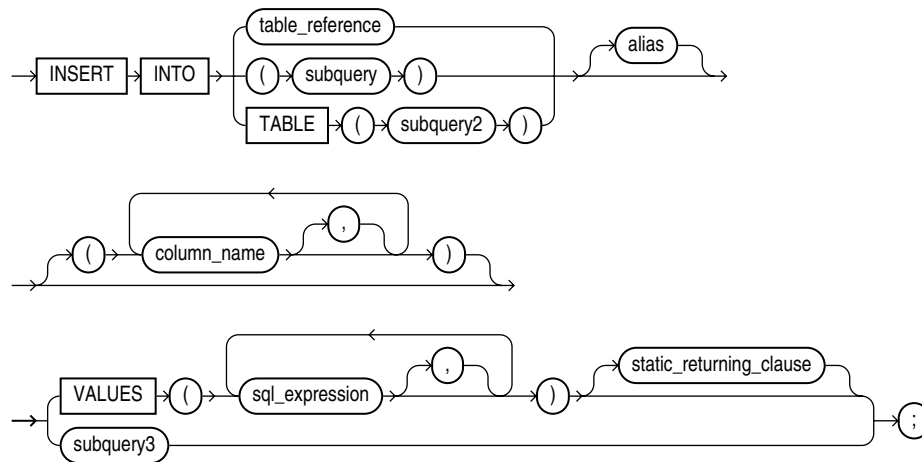
["How PL/SQL Optimizes Your Programs"](#) on page 12-3

INSERT Statement

The `INSERT` statement inserts one or more rows of data into a database table. For a full description of the `INSERT` statement, see *Oracle Database SQL Language Reference*.

Syntax

insert_statement ::=



Keyword and Parameter Description

alias

Another (usually short) name for the referenced table or view.

column_name [, column_name]...

A list of columns in a database table or view. The columns can be listed in any order, as long as the expressions in the `VALUES` clause are listed in the same order. Each column name can only be listed once. If the list does not include all the columns in a table, each missing columns is set to `NULL` or to a default value specified in the `CREATE TABLE` statement.

sql_expression

Any expression valid in SQL—for example, a literal, a PL/SQL variable, or a SQL query that returns a single value. For more information, see *Oracle Database SQL Language Reference*. PL/SQL also lets you use a record variable here.

static_returning_clause

Returns the column values of the inserted rows, in either individual variables or collections (eliminating the need to `SELECT` the rows first). For details, see "[RETURNING INTO Clause](#)" on page 13-121.

subquery

A `SELECT` statement that provides a set of rows for processing. Its syntax is like that of `select_into_statement` without the `INTO` clause. See "[SELECT INTO Statement](#)" on page 13-128.

subquery3

A `SELECT` statement that returns a set of rows. Each row returned by the select statement is inserted into the table. The subquery must return a value for every column in the column list, or for every column in the table if there is no column list.

table_reference

A table or view that must be accessible when you execute the `INSERT` statement, and for which you must have `INSERT` privileges. For the syntax of `table_reference`, see "[DELETE Statement](#)" on page 13-45.

TABLE (subquery2)

The operand of `TABLE` is a `SELECT` statement that returns a single column value representing a nested table. This operator specifies that the value is a collection, not a scalar value.

VALUES (...)

Assigns the values of expressions to corresponding columns in the column list. If there is no column list, the first value is inserted into the first column defined by the `CREATE TABLE` statement, the second value is inserted into the second column, and so on. There must be one value for each column in the column list. The datatypes of the values being inserted must be compatible with the datatypes of corresponding columns in the column list.

Usage Notes

Character and date literals in the `VALUES` list must be enclosed by single quotes (`'`). Numeric literals are not enclosed by quotes.

The implicit cursor `SQL` and the cursor attributes `%NOTFOUND`, `%FOUND`, `%ROWCOUNT`, and `%ISOPEN` let you access useful information about the execution of an `INSERT` statement.

Examples

[Example 6-1, "Data Manipulation with PL/SQL"](#) on page 6-1
[Example 6-5, "Using CURRVAL and NEXTVAL"](#) on page 6-4
[Example 6-7, "Using SQL%FOUND"](#) on page 6-8
[Example 6-37, "Using ROLLBACK"](#) on page 6-34
[Example 6-38, "Using SAVEPOINT with ROLLBACK"](#) on page 6-35
[Example 6-46, "Declaring an Autonomous Trigger"](#) on page 6-43
[Example 6-48, "Invoking an Autonomous Function"](#) on page 6-46
[Example 7-12, "Dynamic SQL"](#) on page 7-14
[Example 10-3, "Creating the emp_admin Package"](#) on page 10-6

Related Topics

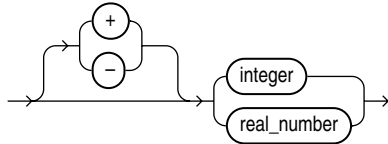
["DELETE Statement"](#) on page 13-45
["SELECT INTO Statement"](#) on page 13-128
["UPDATE Statement"](#) on page 13-144

Literal Declaration

A literal is an explicit numeric, character, string, or Boolean value not represented by an identifier. The numeric literal 135 and the string literal 'hello world' are examples. For more information, see ["Literals"](#) on page 2-5.

Syntax

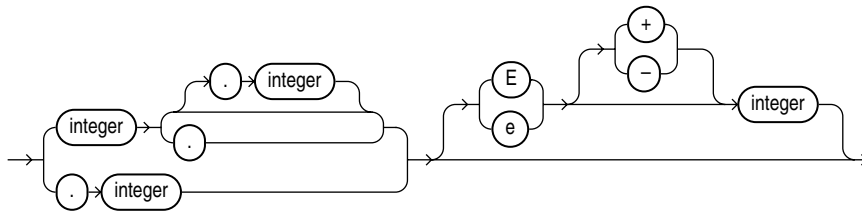
numeric_literal ::=



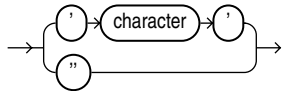
integer_literal ::=



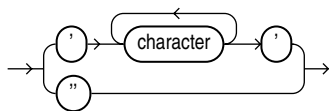
real_number_literal ::=



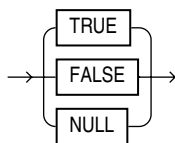
character_literal ::=



string_literal ::=



boolean_literal ::=



Keyword and Parameter Description

character

A member of the PL/SQL character set. For more information, see "[Character Sets and Lexical Units](#)" on page 2-1.

digit

One of the numerals 0 .. 9.

TRUE, FALSE, NULL

A predefined Boolean value.

Usage Notes

Integer and real numeric literals can be used in arithmetic expressions. Numeric literals must be separated by punctuation. Spaces can be used in addition to the punctuation. For additional information, see "[Numeric Literals](#)" on page 2-5.

A character literal is an individual character enclosed by single quotes (apostrophes). Character literals include all the printable characters in the PL/SQL character set: letters, numerals, spaces, and special symbols. PL/SQL is case sensitive within character literals. For example, PL/SQL considers the literals 'Q' and 'q' to be different. For additional information, see "[Character Literals](#)" on page 2-6.

A string literal is a sequence of zero or more characters enclosed by single quotes. The null string (' ') contains zero characters. A string literal can hold up to 32,767 characters. PL/SQL is case sensitive within string literals. For example, PL/SQL considers the literals 'white' and 'White' to be different.

To represent an apostrophe within a string, enter two single quotes instead of one. For literals where doubling the quotes is inconvenient or hard to read, you can designate an escape character using the notation `q' esc_char ... esc_char'`. This escape character must not occur anywhere else inside the string.

Trailing blanks are significant within string literals, so 'abc' and 'abc ' are different. Trailing blanks in a string literal are not trimmed during PL/SQL processing, although they are trimmed if you insert that value into a table column of type CHAR. For additional information, including NCHAR string literals, see "[String Literals](#)" on page 2-6.

The BOOLEAN values TRUE and FALSE cannot be inserted into a database column. For additional information, see "[BOOLEAN Literals](#)" on page 2-7.

Examples

Numeric literals:

```
25 6.34 7E2 25e-03 .1 1. +17 -4.4 -4.5D -4.6F
```

Character literals:

```
'H' '&' ' ' '9' ']' 'g'
```

String literals:

```
'$5,000'  
'02-AUG-87'  
'Don't leave until you're ready and I'm ready.'  
q'#Don't leave until you're ready and I'm ready.##'
```

More examples:

[Example 2-3, "Using DateTime Literals"](#) on page 2-7

[Example 2-34, "Using Conditional Compilation with Database Versions"](#) on page 2-40

Related Topics

["Constant and Variable Declaration"](#) on page 13-29

["Expression Definition"](#) on page 13-57

["Literals"](#) on page 2-5

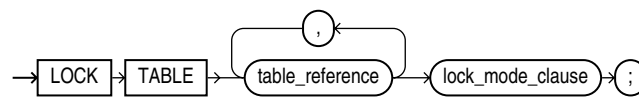
LOCK TABLE Statement

The `LOCK TABLE` statement locks entire database tables in a specified lock mode. That enables you to share or deny access to tables while maintaining their integrity. For more information, see ["Using LOCK TABLE"](#) on page 6-39.

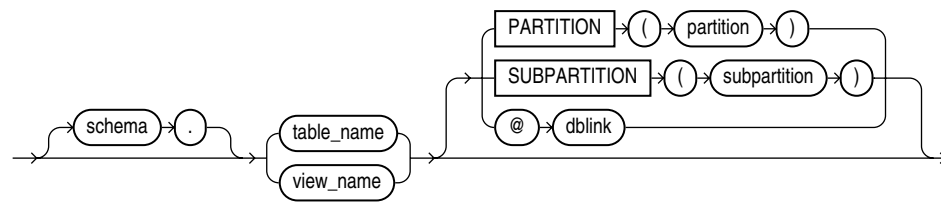
Oracle has extensive automatic features that allow multiple programs to read and write data simultaneously, while each program sees a consistent view of the data; you rarely, if ever, need to lock tables yourself. For additional information on the `LOCK TABLE` SQL statement, see *Oracle Database SQL Language Reference*.

Syntax

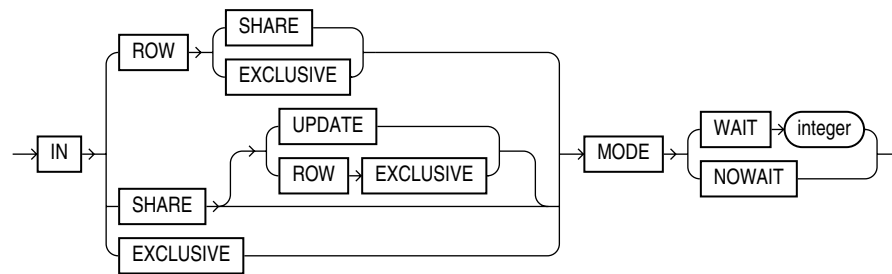
lock_table_statement ::=



table_reference ::=



lock_mode_clause ::=



Keyword and Parameter Description

See *Oracle Database SQL Language Reference*.

Usage Notes

If you omit the keyword `NOWAIT`, Oracle waits until the table is available; the wait has no set limit. Table locks are released when your transaction issues a commit or rollback. A table lock never keeps other users from querying a table, and a query never acquires a table lock. If your program includes SQL locking statements, make sure the Oracle users requesting locks have the privileges needed to obtain the locks. Your DBA can lock any table. Other users can lock tables they own or tables for which they have a privilege, such as `SELECT`, `INSERT`, `UPDATE`, or `DELETE`.

Examples

This statement locks the `employees` table in row shared mode with the `NOWAIT` option:

```
LOCK TABLE employees IN ROW SHARE MODE NOWAIT;
```

Related Topics

["COMMIT Statement" on page 13-28](#)

["ROLLBACK Statement" on page 13-124](#)

LOOP Statements

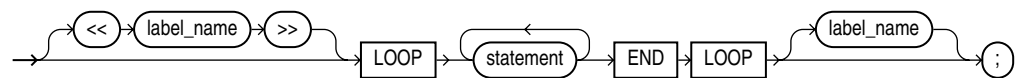
A LOOP statement executes a sequence of statements multiple times. The LOOP and END LOOP keywords enclose the sequence of statements. PL/SQL provides four kinds of loop statements:

- Basic loop
- WHILE loop
- FOR loop
- Cursor FOR loop

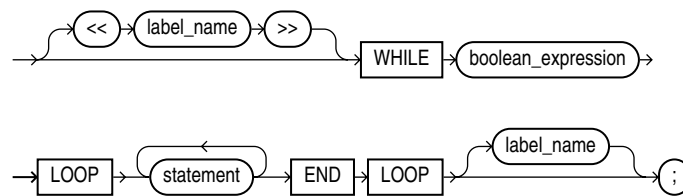
For usage information, see "[Controlling Loop Iterations \(LOOP, EXIT, and CONTINUE Statements\)](#)" on page 4-7.

Syntax

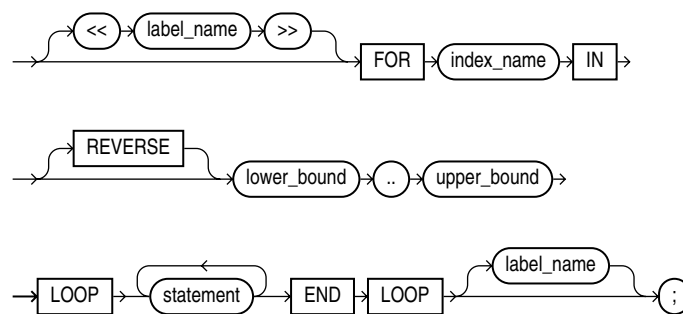
basic_loop_statement ::=



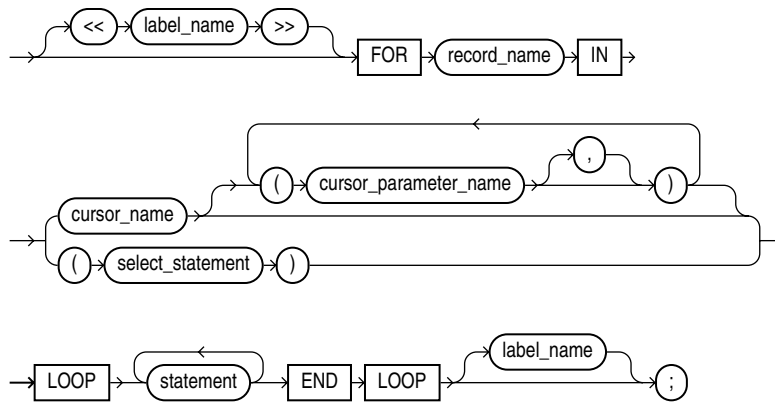
while_loop_statement ::=



for_loop_statement ::=



cursor_for_loop_statement ::=



Keyword and Parameter Description

basic_loop_statement

A loop that executes an unlimited number of times. It encloses a sequence of statements between the keywords LOOP and END LOOP. With each iteration, the sequence of statements is executed, then control resumes at the top of the loop. An EXIT, GOTO, or RAISE statement branches out of the loop. A raised exception also ends the loop.

boolean_expression

An expression that returns the Boolean value TRUE, FALSE, or NULL. It is associated with a sequence of statements, which is executed only if the expression returns TRUE. For the syntax of boolean_expression, see "[Expression Definition](#)" on page 13-57.

cursor_for_loop_statement

Issues a SQL query and loops through the rows in the result set. This is a convenient technique that makes processing a query as simple as reading lines of text in other programming languages.

A cursor FOR loop implicitly declares its loop index as a %ROWTYPE record, opens a cursor, repeatedly fetches rows of values from the result set into fields in the record, and closes the cursor when all rows were processed.

cursor_name

An explicit cursor previously declared within the current scope. When the cursor FOR loop is entered, cursor_name cannot refer to a cursor already opened by an OPEN statement or an enclosing cursor FOR loop.

cursor_parameter_name

A variable declared as the formal parameter of a cursor. For the syntax of cursor_parameter_declaration, see "[Cursor Declaration](#)" on page 13-41. A cursor parameter can appear in a query wherever a constant can appear. The formal parameters of a cursor must be IN parameters.

for_loop_statement

Numeric FOR_LOOP loops iterate over a specified range of integers. The range is part of an iteration scheme, which is enclosed by the keywords FOR and LOOP.

The range is evaluated when the FOR loop is first entered and is never re-evaluated. The loop body is executed once for each integer in the range defined by `lower_bound..upper_bound`. After each iteration, the loop index is incremented.

index_name

An undeclared identifier that names the loop index (sometimes called a loop counter). Its scope is the loop itself; you cannot reference the index outside the loop.

The implicit declaration of `index_name` overrides any other declaration outside the loop. To refer to another variable with the same name, use a label. See [Example 4-18, "Using a Label for Referencing Variables Outside a Loop"](#) on page 4-14.

Inside a loop, the index is treated like a constant: it can appear in expressions, but cannot be assigned a value.

label_name

An optional undeclared identifier that labels a loop. `label_name` must be enclosed by double angle brackets and must appear at the beginning of the loop. Optionally, `label_name` (not enclosed in angle brackets) can also appear at the end of the loop.

You can use `label_name` in an EXIT statement to exit the loop labelled by `label_name`. You can exit not only the current loop, but any enclosing loop.

You cannot reference the index of a FOR loop from a nested FOR loop if both indexes have the same name, unless the outer loop is labeled by `label_name` and you use dot notation. See [Example 4-19, "Using Labels on Loops for Referencing"](#) on page 4-15.

lower_bound .. upper_bound

Expressions that return numbers. Otherwise, PL/SQL raises the predefined exception `VALUE_ERROR`. The expressions are evaluated only when the loop is first entered. The lower bound need not be 1, it can be a negative integer as in the following example:

```
FOR i IN -5..10
```

The loop counter increment (or decrement) must be 1.

Internally, PL/SQL assigns the values of the bounds to temporary `PLS_INTEGER` variables, and, if necessary, rounds the values to the nearest integer. The magnitude range of a `PLS_INTEGER` is -2147483648 to 2147483647, represented in 32 bits. If a bound evaluates to a number outside that range, you get a *numeric overflow* error when PL/SQL attempts the assignment. See ["PLS_INTEGER and BINARY_INTEGER Datatypes"](#) on page 3-2.

By default, the loop index is assigned the value of `lower_bound`. If that value is not greater than the value of `upper_bound`, the sequence of statements in the loop is executed, then the index is incremented. If the value of the index is still not greater than the value of `upper_bound`, the sequence of statements is executed again. This process repeats until the value of the index is greater than the value of `upper_bound`. At that point, the loop completes.

record_name

An implicitly declared record. The record has the same structure as a row retrieved by `cursor_name` or `select_statement`.

The record is defined only inside the loop. You cannot refer to its fields outside the loop. The implicit declaration of `record_name` overrides any other declaration outside the loop. You cannot refer to another record with the same name inside the loop unless you qualify the reference using a block label.

Fields in the record store column values from the implicitly fetched row. The fields have the same names and datatypes as their corresponding columns. To access field values, you use dot notation, as follows:

```
record_name.field_name
```

Select-items fetched from the FOR loop cursor must have simple names or, if they are expressions, must have aliases. In the following example, `wages` is an alias for the select item `salary+NVL(commission_pct,0)*1000`:

```
CURSOR c1 IS SELECT employee_id,  
    salary + NVL(commission_pct,0) * 1000 wages FROM employees ...
```

REVERSE

By default, iteration proceeds upward from the lower bound to the upper bound. If you use the keyword `REVERSE`, iteration proceeds downward from the upper bound to the lower bound. An example follows:

```
BEGIN  
    FOR i IN REVERSE 1..10 LOOP -- i starts at 10, ends at 1  
        DBMS_OUTPUT.PUT_LINE(i); -- statements here execute 10 times  
    END LOOP;  
END;  
/
```

The loop index is assigned the value of `upper_bound`. If that value is not less than the value of `lower_bound`, the sequence of statements in the loop is executed, then the index is decremented. If the value of the index is still not less than the value of `lower_bound`, the sequence of statements is executed again. This process repeats until the value of the index is less than the value of `lower_bound`. At that point, the loop completes.

select_statement

A query associated with an internal cursor unavailable to you. Its syntax is like that of `select_into_statement` without the `INTO` clause. See "[SELECT INTO Statement](#)" on page 13-128. PL/SQL automatically declares, opens, fetches from, and closes the internal cursor. Because `select_statement` is not an independent statement, the implicit cursor `SQL` does not apply to it.

while_loop_statement

The `WHILE-LOOP` statement associates a Boolean expression with a sequence of statements enclosed by the keywords `LOOP` and `END LOOP`. Before each iteration of the loop, the expression is evaluated. If the expression returns `TRUE`, the sequence of statements is executed, then control resumes at the top of the loop. If the expression returns `FALSE` or `NULL`, the loop is bypassed and control passes to the next statement.

Usage Notes

You can use the `EXIT WHEN` statement to exit any loop prematurely. If the Boolean expression in the `WHEN` clause returns `TRUE`, the loop is exited immediately.

When you exit a cursor `FOR` loop, the cursor is closed automatically even if you use an `EXIT` or `GOTO` statement to exit the loop prematurely. The cursor is also closed automatically if an exception is raised inside the loop.

Examples

[Example 4-21, "Using EXIT with a Label in a LOOP"](#) on page 4-15

[Example 6–10, "Fetching with a Cursor"](#) on page 6-11

[Example 6–13, "Fetching Bulk Data with a Cursor"](#) on page 6-12

Related Topics

["Controlling Loop Iterations \(LOOP, EXIT, and CONTINUE Statements\)"](#) on page 4-7

["CONTINUE Statement"](#) on page 13-33

["Cursor Declaration"](#) on page 13-41

["EXIT Statement"](#) on page 13-55

["FETCH Statement"](#) on page 13-67

["OPEN Statement"](#) on page 13-102

MERGE Statement

The `MERGE` statement inserts some rows and updates others in a single operation. The decision about whether to update or insert into the target table is based upon a join condition: rows already in the target table that match the join condition are updated; otherwise, a row is inserted using values from a separate subquery.

For a full description and examples of the `MERGE` statement, see *Oracle Database SQL Language Reference*.

Usage Notes

This statement is primarily useful in data warehousing situations where large amounts of data are commonly inserted and updated. If you only need to insert or update a single row, it is more efficient to do that with the regular PL/SQL techniques: try to update the row, and do an insert instead if the update affects zero rows; or try to insert the row, and do an update instead if the insert raises an exception because the table already contains that primary key.

NULL Statement

The `NULL` statement is a no-op (no operation); it passes control to the next statement without doing anything. In the body of an `IF-THEN` clause, a loop, or a procedure, the `NULL` statement serves as a placeholder. For more information, see ["Using the NULL Statement"](#) on page 4-18.

Syntax

null_statement ::=

→ `NULL` → ;

Usage Notes

The `NULL` statement improves readability by making the meaning and action of conditional statements clear. It tells readers that the associated alternative was not overlooked, that you decided that no action is necessary.

Certain clauses in PL/SQL, such as in an `IF` statement or an exception handler, must contain at least one executable statement. You can use the `NULL` statement to make these constructs compile, while not taking any action.

You might not be able to branch to certain places with the `GOTO` statement because the next statement is `END`, `END IF`, and so on, which are not executable statements. In these cases, you can put a `NULL` statement where you want to branch.

The `NULL` statement and Boolean value `NULL` are not related.

Examples

[Example 1-13, "Creating a Stored Subprogram"](#) on page 1-14

[Example 1-17, "Declaring a Record Type"](#) on page 1-18

[Example 4-23, "Using a NULL Statement to Allow a GOTO to a Label"](#) on page 4-17

[Example 4-25, "Using the NULL Statement to Show No Action"](#) on page 4-18

[Example 4-26, "Using NULL as a Placeholder When Creating a Subprogram"](#) on page 4-18

Related Topics

["Sequential Control \(GOTO and NULL Statements\)"](#) on page 4-16

Object Type Declaration

An object type is a user-defined composite datatype that encapsulates a data structure along with the functions and procedures needed to manipulate the data. The variables that form the data structure are called attributes. The functions and procedures that characterize the action of the object type are called methods. A special kind of method called the constructor creates a new instance of the object type and fills in its attributes.

Object types must be created through SQL and stored in an Oracle database, where they can be shared by many programs. When you define an object type using the `CREATE TYPE` statement, you create an abstract template for some real-world object. The template specifies the attributes and actions the object needs in the application environment. For information on the `CREATE TYPE` SQL statement, see *Oracle Database SQL Language Reference*. For information on the `CREATE TYPE BODY` SQL statement, see *Oracle Database SQL Language Reference*.

The data structure formed by the set of attributes is public (visible to client programs). However, well designed programs do not manipulate it directly. Instead, they use the set of methods provided, so that the data is kept in a proper state.

For more information on object types, see *Oracle Database Object-Relational Developer's Guide*.

Usage Notes

Once an object type is created in your schema, you can use it to declare objects in any PL/SQL block, subprogram, or package. For example, you can use the object type to specify the datatype of an object attribute, table column, PL/SQL variable, bind argument, record field, collection element, formal procedure parameter, or function result.

Like a package, an object type has two parts: a specification and a body. The specification (spec for short) is the interface to your applications; it declares a data structure (set of attributes) along with the operations (methods) needed to manipulate the data. The body fully defines the methods, and so implements the spec.

All the information a client program needs to use the methods is in the spec. Think of the spec as an operational interface and of the body as a black box. You can debug, enhance, or replace the body without changing the spec.

An object type encapsulates data and operations. You can declare attributes and methods in an object type spec, but *not* constants, exceptions, cursors, or types. At least one attribute is required (the maximum is 1000); methods are optional.

In an object type spec, all attributes must be declared before any methods. Only subprograms have an underlying implementation. If an object type specification declares only attributes or call specifications, the object type body is unnecessary. You cannot declare attributes in the body. All declarations in the object type spec are public (visible outside the object type).

You can refer to an attribute only by name (not by its position in the object type). To access or change the value of an attribute, you use dot notation. Attribute names can be chained, which lets you access the attributes of a nested object type.

In an object type, methods can reference attributes and other methods without a qualifier. In SQL statements, calls to a parameterless method require an empty parameter list. In procedural statements, an empty parameter list is optional unless you chain calls, in which case it is required for all but the last call.

From a SQL statement, if you call a MEMBER method on a null instance (that is, SELF is null), the method is not called and a null is returned. From a procedural statement, if you call a MEMBER method on a null instance, PL/SQL raises the predefined exception SELF_IS_NULL before the method is called.

You can declare a map method or an order method but not both. If you declare either method, you can compare objects in SQL and procedural statements. However, if you declare neither method, you can compare objects only in SQL statements and only for equality or inequality. Two objects of the same type are equal *only if* the values of their corresponding attributes are equal.

Like packaged subprograms, methods of the same kind (functions or procedures) can be overloaded. That is, you can use the same name for different methods if their formal parameters differ in number, order, or datatype family.

Every object type has a default constructor method (constructor for short), which is a system-defined function with the same name as the object type. You use the constructor to initialize and return an instance of that object type. You can also define your own constructor methods that accept different sets of parameters. PL/SQL never calls a constructor implicitly, so you must call it explicitly. Constructor calls are allowed wherever function calls are allowed.

Related Topics

["Function Declaration and Definition"](#) on page 13-74

["Package Declaration"](#) on page 13-107

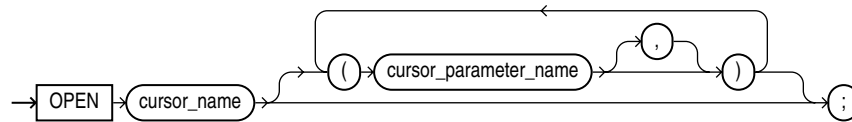
["Procedure Declaration and Definition"](#) on page 13-110

OPEN Statement

The `OPEN` statement executes the query associated with a cursor. It allocates database resources to process the query and identifies the result set—the rows that match the query conditions. The cursor is positioned before the first row in the result set. For more information, see ["Querying Data with PL/SQL"](#) on page 6-16.

Syntax

open_statement ::=



Keyword and Parameter Description

cursor_name

An explicit cursor previously declared within the current scope and not currently open.

cursor_parameter_name

A variable declared as the formal parameter of a cursor. (For the syntax of `cursor_parameter_declaration`, see ["Cursor Declaration"](#) on page 13-41.) A cursor parameter can appear in a query wherever a constant can appear.

Usage Notes

Generally, PL/SQL parses an explicit cursor only the first time it is opened and parses a SQL statement (creating an implicit cursor) only the first time the statement is executed. All the parsed SQL statements are cached. A SQL statement is reparsed only if it is aged out of the cache by a new SQL statement. Although you must close a cursor before you can reopen it, PL/SQL need not reparse the associated `SELECT` statement. If you close, then immediately reopen the cursor, a reparse is definitely not needed.

Rows in the result set are not retrieved when the `OPEN` statement is executed. The `FETCH` statement retrieves the rows. With a `FOR UPDATE` cursor, the rows are locked when the cursor is opened.

If formal parameters are declared, actual parameters must be passed to the cursor. The formal parameters of a cursor must be `IN` parameters; they cannot return values to actual parameters. The values of actual parameters are used when the cursor is opened. The datatypes of the formal and actual parameters must be compatible. The query can also reference PL/SQL variables declared within its scope.

Unless you want to accept default values, each formal parameter in the cursor declaration must have a corresponding actual parameter in the `OPEN` statement. Formal parameters declared with a default value do not need a corresponding actual parameter. They assume their default values when the `OPEN` statement is executed.

You can associate the actual parameters in an `OPEN` statement with the formal parameters in a cursor declaration using positional or named notation.

If a cursor is currently open, you cannot use its name in a cursor `FOR` loop.

Examples

[Example 6–10, "Fetching with a Cursor"](#) on page 6-11

[Example 6–13, "Fetching Bulk Data with a Cursor"](#) on page 6-12

Related Topics

["CLOSE Statement"](#) on page 13-18

["Cursor Declaration"](#) on page 13-41

["FETCH Statement"](#) on page 13-67

["LOOP Statements"](#) on page 13-93

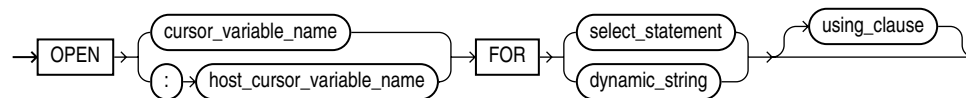
OPEN-FOR Statement

The OPEN-FOR statement executes the SELECT statement associated with a cursor variable. It allocates database resources to process the statement, identifies the result set (the rows that meet the conditions), and positions the cursor variable before the first row in the result set. For more information about cursor variables, see ["Using Cursor Variables \(REF CURSORS\)"](#) on page 6-22.

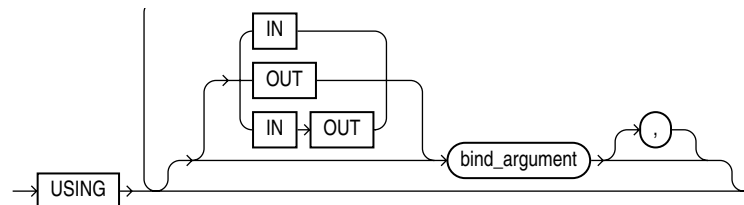
With the optional USING clause, the OPEN-FOR statement processes a dynamic SELECT statement that returns multiple rows: it associates a cursor variable with the SELECT statement, executes the statement, identifies the result set, positions the cursor before the first row in the result set, and zeroes the rows-processed count kept by %ROWCOUNT. For more information, see ["Using the OPEN-FOR, FETCH, and CLOSE Statements"](#) on page 7-4.

Syntax

open_for_statement ::=



using_clause ::=



Keyword and Parameter Description

bind_argument

Either an expression whose value is passed to the dynamic SQL statement (an **in bind**), or a variable in which a value returned by the dynamic SQL statement is stored (an **out bind**). The default parameter mode for `bind_argument` is IN.

cursor_variable_name

A cursor variable or parameter (without a return type), previously declared within the current scope.

host_cursor_variable_name

A cursor variable, which must be declared in a PL/SQL host environment and passed to PL/SQL as a bind argument (hence the colon (:) prefix). The datatype of the cursor variable is compatible with the return type of any PL/SQL cursor variable.

select_statement

A string literal, string variable, or string expression that represents a multiple-row `SELECT` statement (without the final semicolon) associated with `cursor_variable_name`. It must be of type `CHAR`, `VARCHAR2`, or `CLOB` (not `NCHAR` or `NVARCHAR2`).

USING

Used only if `select_statement` includes placeholders, this clause specifies a list of bind arguments.

Usage Notes

You can declare a cursor variable in a PL/SQL host environment such as an OCI or Pro*C program. To open the host cursor variable, you can pass it as a bind argument to an anonymous PL/SQL block. You can reduce network traffic by grouping `OPEN-FOR` statements. For example, the following PL/SQL block opens five cursor variables in a single round-trip:

```
/* anonymous PL/SQL block in host environment */
BEGIN
  OPEN :emp_cv FOR SELECT * FROM employees;
  OPEN :dept_cv FOR SELECT * FROM departments;
  OPEN :grade_cv FOR SELECT * FROM salgrade;
  OPEN :pay_cv FOR SELECT * FROM payroll;
  OPEN :ins_cv FOR SELECT * FROM insurance
END;
```

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

Unlike cursors, cursor variables do not take parameters. Instead, you can pass whole queries (not just parameters) to a cursor variable. Although a PL/SQL stored subprogram can open a cursor variable and pass it back to a calling subprogram, the calling and called subprograms must be in the same instance. You cannot pass or return cursor variables to procedures and functions called through database links. When you declare a cursor variable as the formal parameter of a subprogram that opens the cursor variable, you must specify the `IN OUT` mode. That way, the subprogram can pass an open cursor back to the caller.

Examples

[Example 6-27, "Passing a REF CURSOR as a Parameter"](#) on page 6-24
[Example 6-29, "Stored Procedure to Open a Ref Cursor"](#) on page 6-26
[Example 6-30, "Stored Procedure to Open Ref Cursors with Different Queries"](#) on page 6-26
[Example 6-31, "Cursor Variable with Different Return Types"](#) on page 6-27
[Example 6-32, "Fetching from a Cursor Variable into a Record"](#) on page 6-28
[Example 6-33, "Fetching from a Cursor Variable into Collections"](#) on page 6-28
[Example 7-4, "Native Dynamic SQL with OPEN-FOR, FETCH, and CLOSE Statements"](#)

Related Topics

["CLOSE Statement"](#) on page 13-18
["Cursor Variables"](#) on page 13-37
["EXECUTE IMMEDIATE Statement"](#) on page 13-51
["FETCH Statement"](#) on page 13-67
["LOOP Statements"](#) on page 13-93

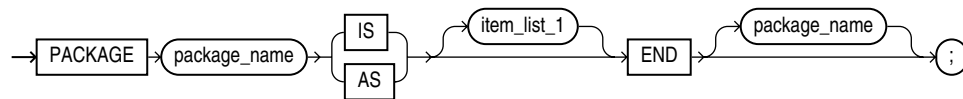
Package Declaration

A package is a schema object that groups logically related PL/SQL types, items, and subprograms. Use packages when writing a set of related subprograms that form an application programming interface (API) that you or others might reuse. Packages have two parts: a specification (spec for short) and a body. For more information, see [Chapter 10, "Using PL/SQL Packages"](#). For an example of a package declaration, see [Example 10-3](#) on page 10-6.

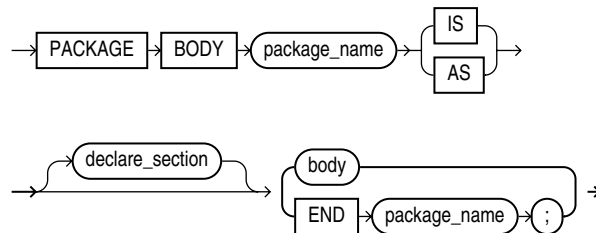
This section shows the package specification and body options for PL/SQL. For information on the `CREATE PACKAGE` SQL statement, see *Oracle Database SQL Language Reference*. For information on the `CREATE PACKAGE BODY` SQL statement, see *Oracle Database SQL Language Reference*.

Syntax

package_specification ::=



package_body ::=



Keyword and Parameter Description

body

For syntax, see ["Block Declaration"](#) on page 13-8.

declare_section

Declares package elements. For syntax, see ["Block Declaration"](#) on page 13-8.

item_list_1

Declares a list of items. For syntax, see ["Block Declaration"](#) on page 13-8.

If an item in `item_list_1` is a pragma, it must be one of the following:

- ["RESTRICT_REFERENCES Pragma"](#) on page 13-118
- ["SERIALLY_REUSABLE Pragma"](#) on page 13-132

package_name

A package stored in the database. For naming conventions, see ["Identifiers"](#) on page 2-3.

Usage Notes

You can use any Oracle tool that supports PL/SQL to create and store packages in an Oracle database. You can issue the `CREATE PACKAGE` and `CREATE PACKAGE BODY` statements interactively from SQL*Plus, or from an Oracle Precompiler or OCI host program. However, you cannot define packages in a PL/SQL block or subprogram.

Most packages have a specification and a body. The specification is the interface to your applications; it declares the types, variables, constants, exceptions, cursors, and subprograms available for use. The body fully defines cursors and subprograms, and so implements the spec.

Only subprograms and cursors have an underlying implementation. If a specification declares only types, constants, variables, exceptions, and call specifications, the package body is unnecessary. The body can still be used to initialize items declared in the specification:

```
CREATE OR REPLACE PACKAGE emp_actions AS
--  additional code here ...
    number_hired INTEGER;
END emp_actions;
/
CREATE OR REPLACE PACKAGE BODY emp_actions AS
BEGIN
    number_hired := 0;
END emp_actions;
/
```

You can code and compile a spec without its body. Once the spec is compiled, stored subprograms that reference the package can be compiled as well. You do not need to define the package bodies fully until you are ready to complete the application. You can debug, enhance, or replace a package body without changing the package spec, which saves you from recompiling subprograms that invoke the package.

Cursors and subprograms declared in a package spec must be defined in the package body. Other program items declared in the package spec cannot be redeclared in the package body.

To match subprogram specs and bodies, PL/SQL does a token-by-token comparison of their headers. Except for white space, the headers must match word for word. Otherwise, PL/SQL raises an exception.

Variables declared in a package keep their values throughout a session, so you can set the value of a package variable in one procedure, and retrieve the same value in a different procedure.

Examples

[Example 1–14, "Creating a Package and Package Body"](#) on page 1-15

[Example 6–43, "Declaring an Autonomous Function in a Package"](#) on page 6-42

[Example 10–3, "Creating the emp_admin Package"](#) on page 10-6

[Example 10–4, "Using PUT_LINE in the DBMS_OUTPUT Package"](#) on page 10-10

Related Topics

["Collection Definition"](#) on page 13-19

["Cursor Declaration"](#) on page 13-41

["Exception Definition"](#) on page 13-48

["Function Declaration and Definition"](#) on page 13-74

["Procedure Declaration and Definition"](#) on page 13-110

["Record Definition"](#) on page 13-115

Procedure Declaration and Definition

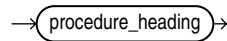
A procedure is a subprogram that performs a specific action. A PL/SQL block or parent subprogram must declare and define a procedure before invoking it. The declaration always includes the specification ("spec"). The declaration can also include the definition. If the declaration does not include the definition, the definition must appear later in the same block or subprogram as the declaration.

For more information about procedures, see ["What Are PL/SQL Subprograms?"](#) on page 8-1.

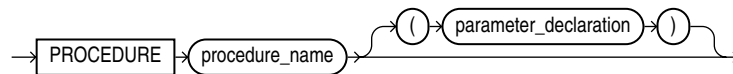
Note: Declaring and defining a procedure in a PL/SQL block or package is different from creating a function with the SQL statement `CREATE PROCEDURE`. For information about `CREATE PROCEDURE`, see *Oracle Database SQL Language Reference*.

Syntax

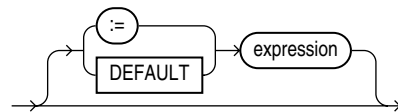
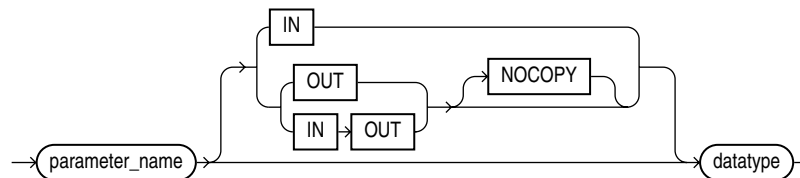
procedure_declaration ::=



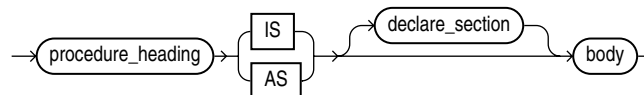
procedure_heading ::=



parameter_declaration ::=



procedure_definition ::=



Keyword and Parameter Description

body

For syntax, see ["Block Declaration"](#) on page 13-8.

datatype

A type specifier. For syntax, see ["Constant and Variable Declaration"](#) on page 13-29.

declare_section

Declares procedure elements. For syntax, see ["Block Declaration"](#) on page 13-8.

exception_handler

Associates an exception with a sequence of statements, which is executed when that exception is raised. For syntax, see ["Exception Definition"](#) on page 13-48.

expression

A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the declaration is elaborated, the value of `expression` is assigned to the parameter. The value and the parameter must have compatible datatypes.

Note: If a procedure call includes an actual parameter for `parameter_name`, then `expression` is not evaluated for that procedure call (see [Example 8-8](#)).

procedure_declaration

Declares, and might also define, a procedure. If the declaration does not define the procedure, the definition must appear later in the same block or subprogram as the declaration. See ["Function Declaration and Definition"](#) on page 13-74.

IN, OUT, IN OUT

Parameter modes that define the action of formal parameters. An `IN` parameter passes values to the subprogram being invoked. An `OUT` parameter returns values to the invoker of the subprogram. An `IN OUT` parameter passes initial values to the subprogram being invoked and returns updated values to the invoker.

item_declaration

Declares and defines a program object. For syntax, see ["Block Declaration"](#) on page 13-8.

name

Is the procedure name.

NOCOPY

A compiler hint (not directive) that allows the PL/SQL compiler to pass `OUT` and `IN OUT` parameters by reference instead of by value (the default). For more information, see ["Specifying Subprogram Parameter Modes"](#) on page 8-9.

parameter_name

A formal parameter, which is a variable declared in a procedure spec and referenced in the procedure body.

procedure_name

The name you choose for the procedure.

statement

A statement. For syntax, see ["Block Declaration"](#) on page 13-8.

type_definition

Specifies a user-defined datatype. For syntax, see ["Block Declaration"](#) on page 13-8.

:= | DEFAULT

Initializes IN parameters to default values, if they are not specified when the procedure is invoked.

Usage Notes

A procedure is invoked as a PL/SQL statement. For example, the procedure `raise_salary` might be invoked as follows:

```
raise_salary(emp_num, amount);
```

Inside a procedure, an IN parameter acts like a constant; you cannot assign it a value. An OUT parameter acts like a local variable; you can change its value and reference the value in any way. An IN OUT parameter acts like an initialized variable; you can assign it a value, which can be assigned to another variable. For summary information about the parameter modes, see [Table 8-3](#) on page 8-10.

Unlike OUT and IN OUT parameters, IN parameters can be initialized to default values. For more information, see ["Using Default Values for Subprogram Parameters"](#) on page 8-10.

Before exiting a procedure, explicitly assign values to all OUT formal parameters. An OUT actual parameter can have a value before the subprogram is invoked. However, when you invoke the subprogram, the value is lost unless you specify the compiler hint `NOCOPY` or the subprogram exits with an unhandled exception.

You can write the procedure spec and body as a unit. Or, you can separate the procedure spec from its body. That way, you can hide implementation details by placing the procedure in a package. You can define procedures in a package body without declaring their specs in the package spec. However, such procedures can be invoked only from inside the package. At least one statement must appear in the executable part of a procedure. The `NULL` statement meets this requirement.

Examples

[Example 1-13, "Creating a Stored Subprogram"](#) on page 1-14

[Example 1-14, "Creating a Package and Package Body"](#) on page 1-15

[Example 1-15, "Invoking a Procedure in a Package"](#) on page 1-16

[Example 10-3, "Creating the emp_admin Package"](#) on page 10-6

Related Topics

["Collection Methods"](#) on page 13-23

["Function Declaration and Definition"](#) on page 13-74

["Package Declaration"](#) on page 13-107

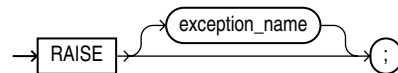
RAISE Statement

The RAISE statement stops normal execution of a PL/SQL block or subprogram and transfers control to an exception handler.

RAISE statements can raise predefined exceptions, such as ZERO_DIVIDE or NO_DATA_FOUND, or user-defined exceptions whose names you decide. For more information, see ["Defining Your Own PL/SQL Exceptions"](#) on page 11-6.

Syntax

raise_statement ::=



Keyword and Parameter Description

exception_name

A predefined or user-defined exception. For a list of the predefined exceptions, see ["Summary of Predefined PL/SQL Exceptions"](#) on page 11-4.

Usage Notes

Raise an exception in a PL/SQL block or subprogram only when an error makes it impractical to continue processing. You can code a RAISE statement for a given exception anywhere within the scope of that exception.

When an exception is raised, if PL/SQL cannot find a handler for it in the current block, the exception propagates to successive enclosing blocks, until a handler is found or there are no more blocks to search. If no handler is found, PL/SQL returns an unhandled exception error to the host environment.

In an exception handler, you can omit the exception name in a RAISE statement, which raises the current exception again. This technique enables you to take some initial corrective action (perhaps just logging the problem), then pass control to another handler that does more extensive correction. When an exception is reraised, the first block searched is the enclosing block, not the current block.

Examples

[Example 1-13, "Creating a Stored Subprogram"](#) on page 1-14

[Example 10-3, "Creating the emp_admin Package"](#) on page 10-6

[Example 11-3, "Scope of PL/SQL Exceptions"](#) on page 11-6

[Example 11-9, "Reraising a PL/SQL Exception"](#) on page 11-12

Related Topics

["Exception Definition"](#) on page 13-48

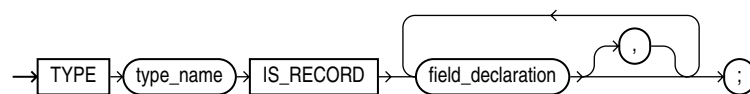
Record Definition

Records are composite variables that can store data values of different types, similar to a `struct` type in C, C++, or Java. For more information, see ["Understanding PL/SQL Records"](#) on page 5-5.

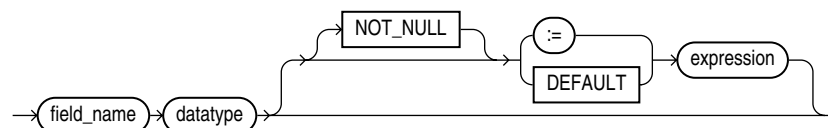
In PL/SQL records are useful for holding data from table rows, or certain columns from table rows. For ease of maintenance, you can declare variables as `table%ROWTYPE` or `cursor%ROWTYPE` instead of creating new record types.

Syntax

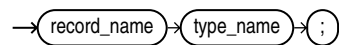
record_type_definition ::=



record_field_declaration ::=



record_type_declaration ::=



Keyword and Parameter Description

datatype

A datatype specifier. For the syntax of `datatype`, see ["Constant and Variable Declaration"](#) on page 13-29.

expression

A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. For the syntax of `expression`, see ["Expression Definition"](#) on page 13-57. When the declaration is elaborated, the value of `expression` is assigned to the field. The value and the field must have compatible datatypes.

field_name

A field in a user-defined record.

NOT NULL

At run time, trying to assign a null to a field defined as `NOT NULL` raises the predefined exception `VALUE_ERROR`. The constraint `NOT NULL` must be followed by an initialization clause.

record_name

A user-defined record.

type_name

A user-defined record type that was defined using the datatype specifier `RECORD`.

:= | DEFAULT

Initializes fields to default values.

Usage Notes

You can define `RECORD` types and declare user-defined records in the declarative part of any block, subprogram, or package.

A record can be initialized in its declaration. You can use the `%TYPE` attribute to specify the datatype of a field. You can add the `NOT NULL` constraint to any field declaration to prevent the assigning of nulls to that field. Fields declared as `NOT NULL` must be initialized. To reference individual fields in a record, you use dot notation. For example, to reference the `dname` field in the `dept_rec` record, use `dept_rec.dname`.

Instead of assigning values separately to each field in a record, you can assign values to all fields at once:

- You can assign one user-defined record to another if they have the same datatype. (Having fields that match exactly is not enough.) You can assign a `%ROWTYPE` record to a user-defined record if their fields match in number and order, and corresponding fields have compatible datatypes.
- You can use the `SELECT` or `FETCH` statement to fetch column values into a record. The columns in the select-list must appear in the same order as the fields in your record.

User-defined records follow the usual scoping and instantiation rules. In a package, they are instantiated when you first reference the package and cease to exist when you end the database session. In a block or subprogram, they are instantiated when you enter the block or subprogram and cease to exist when you exit the block or subprogram.

Like scalar variables, user-defined records can be declared as the formal parameters of procedures and functions. The restrictions that apply to scalar parameters also apply to user-defined records.

You can specify a `RECORD` type in the `RETURN` clause of a function spec. That allows the function to return a user-defined record of the same type. When invoking a function that returns a user-defined record, use the following syntax to reference fields in the record:

```
function_name(parameter_list).field_name
```

To reference nested fields, use this syntax:

```
function_name(parameter_list).field_name.nested_field_name
```

If the function takes no parameters, code an empty parameter list. The syntax follows:

```
function_name().field_name
```

Examples

[Example 1–17, "Declaring a Record Type"](#) on page 1-18

- [Example 5–8, "VARRAY of Records" on page 5-10](#)
- [Example 5–20, "Assigning Values to VARRAYs with Complex Datatypes" on page 5-15](#)
- [Example 5–21, "Assigning Values to Tables with Complex Datatypes" on page 5-16](#)
- [Example 5–41, "Declaring and Initializing a Simple Record Type" on page 5-31](#)
- [Example 5–42, "Declaring and Initializing Record Types" on page 5-31](#)
- [Example 5–44, "Returning a Record from a Function" on page 5-33](#)
- [Example 5–45, "Using a Record as Parameter to a Procedure" on page 5-33](#)
- [Example 5–46, "Declaring a Nested Record" on page 5-33](#)
- [Example 5–47, "Assigning Default Values to a Record" on page 5-34](#)
- [Example 5–50, "Inserting a PL/SQL Record Using %ROWTYPE" on page 5-36](#)
- [Example 5–51, "Updating a Row Using a Record" on page 5-37](#)
- [Example 5–52, "Using the RETURNING INTO Clause with a Record" on page 5-37](#)
- [Example 5–53, "Using BULK COLLECT with a SELECT INTO Statement" on page 5-38](#)
- [Example 6–26, "Cursor Variable Returning a Record Type" on page 6-24](#)
- [Example 10–3, "Creating the emp_admin Package" on page 10-6](#)

Related Topics

- ["Collection Definition" on page 13-19](#)
- ["Function Declaration and Definition" on page 13-74](#)
- ["Package Declaration" on page 13-107](#)
- ["Procedure Declaration and Definition" on page 13-110](#)

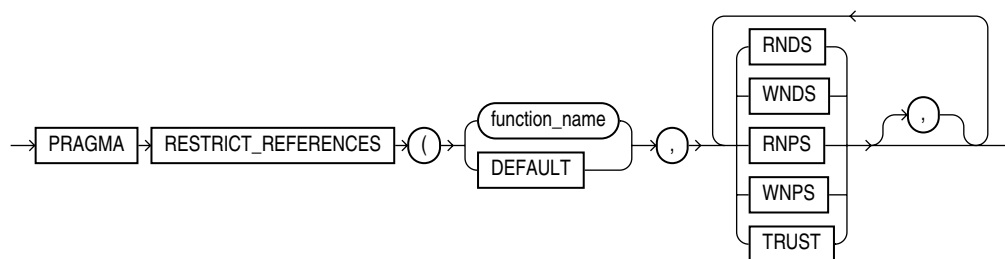
RESTRICT_REFERENCES Pragma

Note: The `RESTRICT REFERENCES` pragma is deprecated. Oracle recommends using `DETERMINISTIC` and `PARALLEL_ENABLE` (described in "Function Declaration and Definition" on page 13-74) instead of `RESTRICT REFERENCES`.

The `RESTRICT REFERENCES` pragma asserts that a subprogram (usually a function) in a package specification or object type specification does not read or write database tables or package variables. Subprograms that do read or write database tables or package variables are difficult to optimize, because any call to the subprogram might produce different results or encounter errors.

Syntax

restrict_references_pragma ::=



Keyword and Parameter Description

DEFAULT

Specifies that the pragma applies to all subprograms in the package specification or object type specification.

function_name

The name of a user-defined subprogram, usually a function.

PRAGMA

Signifies that the statement is a pragma (compiler directive). Pragmas are processed at compile time, not at run time. They pass information to the compiler.

RNDS

Asserts that the subprogram reads no database state (does not query database tables).

RNPS

Asserts that the subprogram reads no package state (does not reference the values of packaged variables)

TRUST

Asserts that the subprogram can be trusted not to violate one or more rules.

WNDS

Asserts that the subprogram writes no database state (does not modify tables).

WNPS

Asserts that the subprogram writes no package state (does not change the values of packaged variables).

Usage Notes

You can declare the pragma `RESTRICT_REFERENCES` only in a package specification or object type specification. You can specify up to four constraints (`RNDS`, `RNPS`, `WNDS`, `WNPS`) in any order. To invoke a subprogram from parallel queries, you must specify all four constraints. No constraint implies another. Typically, this pragma is specified for functions. If a function calls procedures, specify the pragma for those procedures as well.

When you specify `TRUST`, the subprogram body is not checked for violations of the constraints listed in the pragma. The subprogram is trusted not to violate them. Skipping these checks can improve performance. `TRUST` is needed for functions written in C or Java that are invoked from PL/SQL, since PL/SQL cannot verify them at run time.

If you specify `DEFAULT` instead of a subprogram name, the pragma applies to all subprograms in the package specification or object type specification (including the system-defined constructor for object types). You can still declare the pragma for individual subprograms, overriding the `DEFAULT` pragma.

A `RESTRICT_REFERENCES` pragma can apply to only one subprogram declaration. A pragma that references the name of overloaded subprograms always applies to the most recent subprogram declaration.

Examples

[Example 6–48, "Invoking an Autonomous Function"](#) on page 6-46

[Example 8–23, "RESTRICT_REFERENCES Pragma"](#) on page 8-27

Related Topics

["AUTONOMOUS_TRANSACTION Pragma"](#) on page 13-6

["EXCEPTION_INIT Pragma"](#) on page 13-47

["SERIALLY_REUSABLE Pragma"](#) on page 13-132

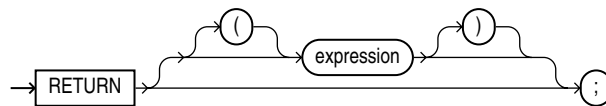
["Controlling Side Effects of PL/SQL Subprograms"](#) on page 8-26

RETURN Statement

The RETURN statement immediately completes the execution of a subprogram and returns control to the invoker. Execution resumes with the statement following the subprogram call. In a function, the RETURN statement also sets the function identifier to the return value. See ["Using the RETURN Statement"](#) on page 8-5.

Syntax

return_statement ::=



Keyword and Parameter Description

expression

A combination of variables, constants, literals, operators, and function calls. The simplest expression consists of a single variable. When the RETURN statement is executed, the value of `expression` is assigned to the function identifier.

Usage Notes

The RETURN statement is different than the RETURN clause in a function spec, which specifies the datatype of the return value.

A subprogram can contain several RETURN statements. Executing any of them completes the subprogram immediately. The RETURN statement might not be positioned as the last statement in the subprogram. The RETURN statement can be used in an anonymous block to exit the block and all enclosing blocks, but the RETURN statement cannot contain an expression.

In procedures, a RETURN statement cannot contain an expression. The statement just returns control to the invoker before the normal end of the procedure is reached. In functions, a RETURN statement must contain an expression, which is evaluated when the RETURN statement is executed. The resulting value is assigned to the function identifier. In functions, there must be at least one execution path that leads to a RETURN statement. Otherwise, PL/SQL raises an exception at run time.

Examples

[Example 1-14, "Creating a Package and Package Body"](#) on page 1-15

[Example 2-15, "Using a Subprogram Name for Name Resolution"](#) on page 2-16

[Example 5-44, "Returning a Record from a Function"](#) on page 5-33

[Example 6-43, "Declaring an Autonomous Function in a Package"](#) on page 6-42

[Example 6-48, "Invoking an Autonomous Function"](#) on page 6-46

[Example 10-3, "Creating the emp_admin Package"](#) on page 10-6

Related Topics

["Function Declaration and Definition"](#) on page 13-74

RETURNING INTO Clause

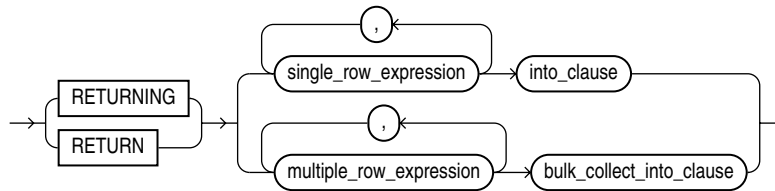
The `RETURNING INTO` clause specifies the variables in which to store the values returned by the statement to which the clause belongs. The variables can be either individual variables or collections. If the statement does not affect any rows, the values of the variables are undefined.

The **static** `RETURNING INTO` clause belongs to a `DELETE`, `INSERT`, or `UPDATE` statement. The **dynamic** `RETURNING INTO` clause belongs to an `EXECUTE IMMEDIATE` statement.

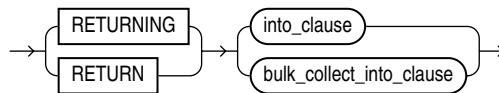
You cannot use the `RETURNING INTO` clause for remote or parallel deletes.

Syntax

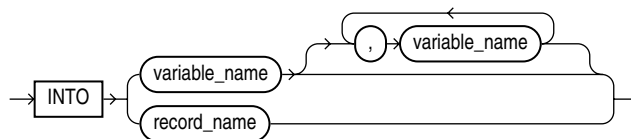
static_returning_clause ::=



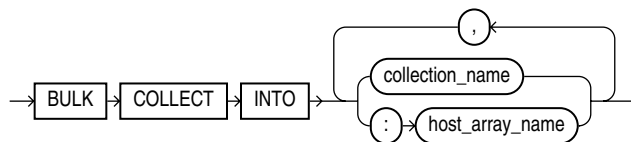
dynamic_returning_clause ::=



into_clause ::=



bulk_collect_into_clause ::=



Keyword and Parameter Description

BULK COLLECT INTO

Used only for a statement that returns multiple rows, this clause specifies one or more collections in which to store the returned rows. This clause must have a corresponding, type-compatible `collection_item` or `:host_array_name` for each `select_item` in the statement to which the `RETURNING INTO` clause belongs.

For the reason to use this clause, see ["Reducing Loop Overhead for DML Statements and Queries with Bulk SQL"](#) on page 12-11.

collection_name

The name of a declared collection, into which returned rows are stored.

host_array_name

An array into which returned rows are stored. The array must be declared in a PL/SQL host environment and passed to PL/SQL as a bind argument (hence the colon (:)) prefix).

INTO

Used only for a statement that returns a single row, this clause specifies the variables or record into which the column values of the returned row are stored. This clause must have a corresponding, type-compatible variable or record field for each `select_item` in the statement to which the RETURNING INTO clause belongs.

multiple_row_expression

An expression that returns multiple rows of a table.

record_name

A record into which a returned row is stored.

single_row_expression

An expression that returns a single row of a table.

variable_name

Either the name of a variable into which a column value of the returned row is stored, or the name of a cursor variable that is declared in a PL/SQL host environment and passed to PL/SQL as a bind argument. The datatype of the cursor variable is compatible with the return type of any PL/SQL cursor variable.

Usage

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

At run time, bind arguments or define variables replace corresponding placeholders in the dynamic SQL statement. Every placeholder must be associated with a bind argument in the USING clause or RETURNING INTO clause (or both) or with a define variable in the INTO clause.

The value of a bind argument cannot be a Boolean literal (TRUE, FALSE, or NULL). To pass the value NULL to the dynamic SQL statement, see ["Uninitialized Variable for NULL in USING Clause"](#) on page 7-4.

Examples

[Example 5-52, "Using the RETURNING INTO Clause with a Record"](#) on page 5-37

[Example 6-1, "Data Manipulation with PL/SQL"](#) on page 6-1

[Example 12-15, "Using BULK COLLECT with the RETURNING INTO Clause"](#) on page 12-22

[Example 12–16, "Using FORALL with BULK COLLECT"](#) on page 12-23

Related Topics

["DELETE Statement"](#) on page 13-45

["EXECUTE IMMEDIATE Statement"](#) on page 13-51

["SELECT INTO Statement"](#) on page 13-128

["UPDATE Statement"](#) on page 13-144

ROLLBACK Statement

The ROLLBACK statement is the inverse of the COMMIT statement. It undoes some or all database changes made during the current transaction. For more information, see ["Overview of Transaction Processing in PL/SQL"](#) on page 6-32.

The SQL ROLLBACK statement can be embedded as static SQL in PL/SQL. For syntax details on the SQL ROLLBACK statement, see *Oracle Database SQL Language Reference*.

Usage Notes

All savepoints marked after the savepoint to which you roll back are erased. The savepoint to which you roll back is not erased. For example, if you mark savepoints A, B, C, and D in that order, then roll back to savepoint B, only savepoints C and D are erased.

An implicit savepoint is marked before executing an INSERT, UPDATE, or DELETE statement. If the statement fails, a rollback to this implicit savepoint is done. Normally, 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.

Examples

[Example 2-14, "Using a Block Label for Name Resolution"](#) on page 2-15

[Example 6-37, "Using ROLLBACK"](#) on page 6-34

[Example 6-38, "Using SAVEPOINT with ROLLBACK"](#) on page 6-35

[Example 6-39, "Reusing a SAVEPOINT with ROLLBACK"](#) on page 6-35

Related Topics

["COMMIT Statement"](#) on page 13-28

["SAVEPOINT Statement"](#) on page 13-127

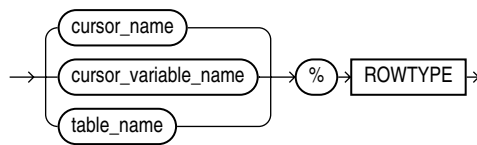
%ROWTYPE Attribute

The %ROWTYPE attribute provides a record type that represents a row in a database table. The record can store an entire row of data selected from the table or fetched from a cursor or cursor variable. Variables declared using %ROWTYPE are treated like those declared using a datatype name. You can use the %ROWTYPE attribute in variable declarations as a datatype specifier.

Fields in a record and corresponding columns in a row have the same names and datatypes. However, fields in a %ROWTYPE record do not inherit constraints, such as the NOT NULL column or check constraint, or default values. For more information, see ["Using the %ROWTYPE Attribute"](#) on page 2-12.

Syntax

%rowtype_attribute ::=



Keyword and Parameter Description

cursor_name

An explicit cursor previously declared within the current scope.

cursor_variable_name

A PL/SQL strongly typed cursor variable, previously declared within the current scope.

table_name

A database table or view that must be accessible when the declaration is elaborated.

Usage Notes

Declaring variables as the type *table_name*%ROWTYPE is a convenient way to transfer data between database tables and PL/SQL. You create a single variable rather than a separate variable for each column. You do not need to know the name of every column. You refer to the columns using their real names instead of made-up variable names. If columns are later added to or dropped from the table, your code can keep working without changes.

To reference a field in the record, use dot notation (*record_name.field_name*). You can read or write one field at a time this way.

There are two ways to assign values to all fields in a record at once:

- First, PL/SQL allows aggregate assignment between entire records if their declarations refer to the same table or cursor.
- You can assign a list of column values to a record by using the SELECT or FETCH statement. The column names must appear in the order in which they were

declared. Select-items fetched from a cursor associated with %ROWTYPE must have simple names or, if they are expressions, must have aliases.

Examples

[Example 1–7, "Using %ROWTYPE with an Explicit Cursor" on page 1-9](#)
[Example 2–8, "Using %ROWTYPE with Table Rows" on page 2-12](#)
[Example 2–9, "Using the %ROWTYPE Attribute" on page 2-12](#)
[Example 2–10, "Assigning Values to a Record with a %ROWTYPE Declaration" on page 2-13](#)
[Example 3–14, "Using SUBTYPE with %TYPE and %ROWTYPE" on page 3-25](#)
[Example 5–7, "Specifying Collection Element Types with %TYPE and %ROWTYPE" on page 5-9](#)
[Example 5–20, "Assigning Values to VARRAYs with Complex Datatypes" on page 5-15](#)
[Example 5–42, "Declaring and Initializing Record Types" on page 5-31](#)
[Example 6–24, "Cursor Variable Returning a %ROWTYPE Variable" on page 6-23](#)
[Example 6–25, "Using the %ROWTYPE Attribute to Provide the Datatype" on page 6-24](#)

Related Topics

["Constant and Variable Declaration" on page 13-29](#)
["Cursor Declaration" on page 13-41](#)
["Cursor Variables" on page 13-37](#)
["FETCH Statement" on page 13-67](#)

SAVEPOINT Statement

The `SAVEPOINT` statement names and marks the current point in the processing of a transaction. With the `ROLLBACK TO` statement, savepoints undo parts of a transaction instead of the whole transaction. For more information, see ["Overview of Transaction Processing in PL/SQL"](#) on page 6-32.

The SQL `SAVEPOINT` statement can be embedded as static SQL in PL/SQL. For syntax details on the SQL `SAVEPOINT` statement, see *Oracle Database SQL Language Reference*.

Syntax

savepoint_statement ::=

→ `SAVEPOINT` (savepoint_name) ;

Usage Notes

A simple rollback or commit erases all savepoints. When you roll back to a savepoint, any savepoints marked after that savepoint are erased. The savepoint to which you roll back remains.

You can reuse savepoint names within a transaction. The savepoint moves from its old position to the current point in the transaction.

If you mark a savepoint within a recursive subprogram, new instances of the `SAVEPOINT` statement are executed at each level in the recursive descent. You can only roll back to the most recently marked savepoint.

An implicit savepoint is marked before executing an `INSERT`, `UPDATE`, or `DELETE` statement. If the statement fails, a rollback to the implicit savepoint is done. Normally, just the failed SQL statement is rolled back, not the whole transaction; if the statement raises an unhandled exception, the host environment (such as SQL*Plus) determines what is rolled back.

Examples

[Example 6–38, "Using SAVEPOINT with ROLLBACK"](#) on page 6-35

[Example 6–39, "Reusing a SAVEPOINT with ROLLBACK"](#) on page 6-35

Related Topics

["COMMIT Statement"](#) on page 13-28

["ROLLBACK Statement"](#) on page 13-124

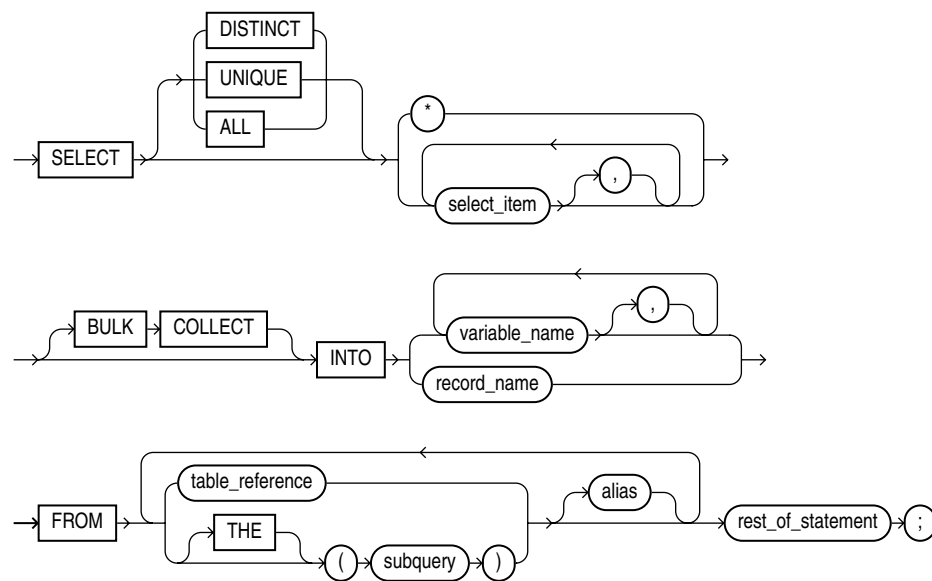
SELECT INTO Statement

The `SELECT INTO` statement retrieves values from one or more database tables (as the SQL `SELECT` statement does) and stores them in either variables or a record (which the SQL `SELECT` statement does not do). For information about the SQL `SELECT` statement, see *Oracle Database SQL Language Reference*.

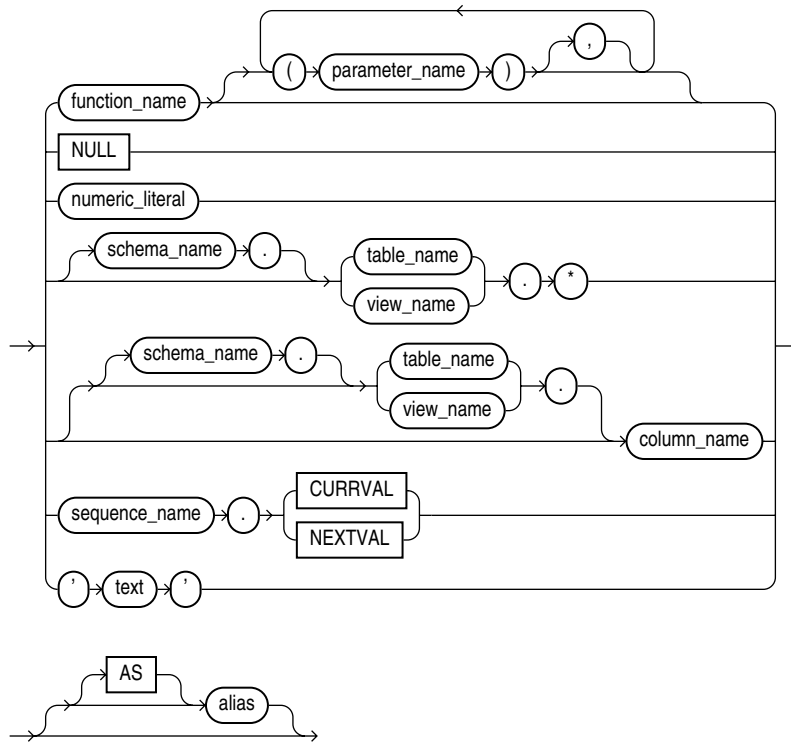
By default, the `SELECT INTO` statement retrieves one or more columns from a single row. With the `BULK COLLECT INTO` clause, this statement retrieves an entire result set at once.

Syntax

select_into_statement ::=



select_item ::=



Keyword and Parameter Description

alias

Another (usually short) name for the referenced column, table, or view.

BULK COLLECT INTO

Stores result values in one or more collections, for faster queries than loops with `FETCH` statements. For more information, see ["Reducing Loop Overhead for DML Statements and Queries with Bulk SQL"](#) on page 12-11.

collection_name

A declared collection into which `select_item` values are fetched. For each `select_item`, there must be a corresponding, type-compatible collection in the list.

function_name

A user-defined function.

host_array_name

An array (declared in a PL/SQL host environment and passed to PL/SQL as a bind argument) into which `select_item` values are fetched. For each `select_item`, there must be a corresponding, type-compatible array in the list. Host arrays must be prefixed with a colon.

numeric_literal

A literal that represents a number or a value implicitly convertible to a number.

parameter_name

A formal parameter of a user-defined function.

record_name

A user-defined or %ROWTYPE record into which rows of values are fetched. For each `select_item` value returned by the query, there must be a corresponding, type-compatible field in the record.

rest_of_statement

Anything that can follow the FROM clause in a SQL SELECT statement (except the SAMPLE clause).

schema_name

The schema containing the table or view. If you omit `schema_name`, Oracle assumes the table or view is in your schema.

subquery

A SELECT statement that provides a set of rows for processing. Its syntax is similar to that of `select_into_statement` without the INTO clause.

table_reference

A table or view that must be accessible when you execute the SELECT statement, and for which you must have SELECT privileges. For the syntax of `table_reference`, see "[DELETE Statement](#)" on page 13-45.

TABLE (subquery2)

The operand of TABLE is a SELECT statement that returns a single column value, which must be a nested table or a varray. Operator TABLE informs Oracle that the value is a collection, not a scalar value.

variable_name

A previously declared variable into which a `select_item` value is fetched. For each `select_item` value returned by the query, there must be a corresponding, type-compatible variable in the list.

Usage Notes

By default, a SELECT INTO statement must return only one row. Otherwise, PL/SQL raises the predefined exception `TOO_MANY_ROWS` and the values of the variables in the INTO clause are undefined. Make sure your WHERE clause is specific enough to only match one row

If no rows are returned, PL/SQL raises `NO_DATA_FOUND`. You can guard against this exception by selecting the result of the aggregate function `COUNT (*)`, which returns a single value, even if no rows match the condition.

A SELECT BULK COLLECT INTO statement can return multiple rows. You must set up collection variables to hold the results. You can declare associative arrays or nested tables that grow as needed to hold the entire result set.

The implicit cursor SQL and its attributes %NOTFOUND, %FOUND, %ROWCOUNT, and %ISOPEN provide information about the execution of a SELECT INTO statement.

Examples

- [Example 1–4, "Using SELECT INTO to Assign Values to Variables" on page 1-6](#)
- [Example 1–5, "Assigning Values to Variables as Parameters of a Subprogram" on page 1-7](#)
- [Example 1–10, "Using WHILE-LOOP for Control" on page 1-11](#)
- [Example 5–51, "Updating a Row Using a Record" on page 5-37](#)
- [Example 5–52, "Using the RETURNING INTO Clause with a Record" on page 5-37](#)
- [Example 6–5, "Using CURRVAL and NEXTVAL" on page 6-4](#)
- [Example 6–37, "Using ROLLBACK" on page 6-34](#)
- [Example 6–38, "Using SAVEPOINT with ROLLBACK" on page 6-35](#)
- [Example 6–43, "Declaring an Autonomous Function in a Package" on page 6-42](#)
- [Example 7–12, "Dynamic SQL" on page 7-14](#)

Related Topics

- ["Assignment Statement" on page 13-3](#)
- ["FETCH Statement" on page 13-67](#)
- ["%ROWTYPE Attribute" on page 13-125](#)

SERIALLY_REUSABLE Pragma

The `SERIALLY_REUSABLE` pragma indicates that the package state is needed only for the duration of one call to the server (for example, an OCI call to the database or a stored procedure call through a database link). After this call, the storage for the package variables can be reused, reducing the memory overhead for long-running sessions. For more information, see *Oracle Database Advanced Application Developer's Guide*.

Syntax

serially_reusable_pragma ::=

→ PRAGMA → SERIALLY_REUSABLE → ;

Keyword and Parameter Description

PRAGMA

Signifies that the statement is a pragma (compiler directive). Pragmas are processed at compile time, not at run time. They pass information to the compiler.

Usage Notes

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

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

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

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

Examples

[Example 13-5](#) creates a serially reusable package.

Example 13-5 Creating a Serially Reusable Package

```
CREATE PACKAGE pkg1 IS
  PRAGMA SERIALLY_REUSABLE;
  num NUMBER := 0;
  PROCEDURE init_pkg_state(n NUMBER);
  PROCEDURE print_pkg_state;
END pkg1;
/
CREATE PACKAGE BODY pkg1 IS
  PRAGMA SERIALLY_REUSABLE;
  PROCEDURE init_pkg_state (n NUMBER) IS
  BEGIN
    pkg1.num := n;
```

```
END;  
PROCEDURE print_pkg_state IS  
BEGIN  
    DBMS_OUTPUT.PUT_LINE('Num: ' || pkg1.num);  
END;  
END pkg1;  
/
```

Related Topics

["AUTONOMOUS_TRANSACTION Pragma"](#) on page 13-6

["EXCEPTION_INIT Pragma"](#) on page 13-47

["INLINE Pragma"](#) on page 13-82

["RESTRICT_REFERENCES Pragma"](#) on page 13-118

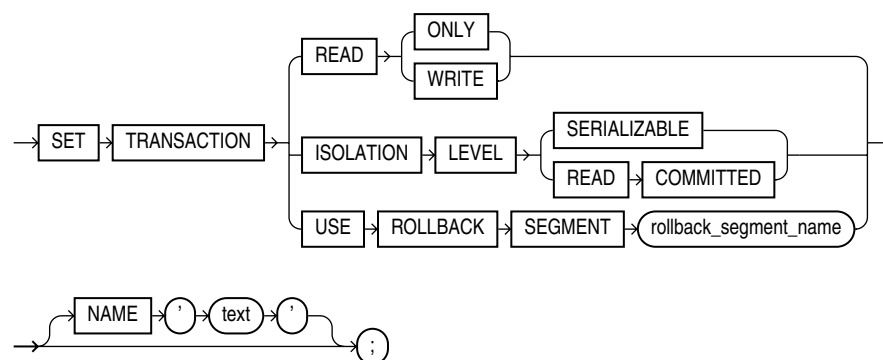
SET TRANSACTION Statement

The `SET TRANSACTION` statement begins a read-only or read/write transaction, establishes an isolation level, or assigns the current transaction to a specified rollback segment. Read-only transactions are useful for running multiple queries against one or more tables while other users update the same tables. For more information, see ["Setting Transaction Properties \(SET TRANSACTION Statement\)"](#) on page 6-37.

For additional information on the `SET TRANSACTION` SQL statement, see *Oracle Database SQL Language Reference*.

Syntax

set_transaction_statement ::=



Keyword and Parameter Description

READ ONLY

Establishes the current transaction as read-only, so that subsequent queries see only changes committed before the transaction began. The use of `READ ONLY` does not affect other users or transactions.

READ WRITE

Establishes the current transaction as read/write. The use of `READ WRITE` does not affect other users or transactions. If the transaction executes a data manipulation statement, Oracle assigns the transaction to a rollback segment.

ISOLATION LEVEL

Specifies how to handle transactions that modify the database.

SERIALIZABLE: If a serializable transaction tries to execute a SQL data manipulation statement that modifies any table already modified by an uncommitted transaction, the statement fails.

SERIALIZABLE

To enable `SERIALIZABLE` mode, your DBA must set the Oracle initialization parameter `COMPATIBLE` to 7.3.0 or higher.

READ COMMITTED: If a transaction includes SQL data manipulation statements that require row locks held by another transaction, the statement waits until the row locks are released.

USE ROLLBACK SEGMENT

Assigns the current transaction to the specified rollback segment and establishes the transaction as read/write. You cannot use this parameter with the `READ ONLY` parameter in the same transaction because read-only transactions do not generate rollback information.

NAME

Specifies a name or comment text for the transaction. This is better than using the `COMMIT COMMENT` feature because the name is available while the transaction is running, making it easier to monitor long-running and in-doubt transactions.

Usage Notes

The `SET TRANSACTION` statement must be the first SQL statement in the transaction and can appear only once in the transaction.

Examples

[Example 6-40, "Using SET TRANSACTION to Begin a Read-only Transaction"](#) on page 6-37

Related Topics

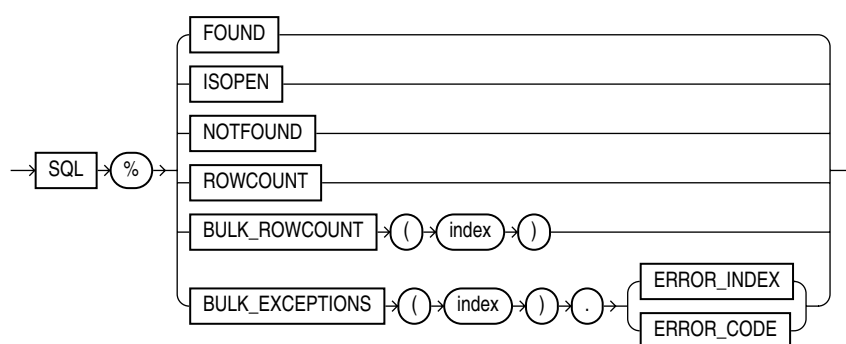
["COMMIT Statement"](#) on page 13-28
["ROLLBACK Statement"](#) on page 13-124
["SAVEPOINT Statement"](#) on page 13-127

SQL Cursor

Oracle implicitly opens a cursor to process each SQL statement not associated with an explicit cursor. In PL/SQL, you can refer to the most recent implicit cursor as the SQL cursor, which always has the attributes `%FOUND`, `%ISOPEN`, `%NOTFOUND`, and `%ROWCOUNT`. They provide information about the execution of data manipulation statements. The SQL cursor has additional attributes, `%BULK_ROWCOUNT` and `%BULK_EXCEPTIONS`, designed for use with the `FORALL` statement. For more information, see ["Querying Data with PL/SQL"](#) on page 6-16.

Syntax

sql_cursor ::=



Keyword and Parameter Description

%BULK_ROWCOUNT

A composite attribute designed for use with the `FORALL` statement. This attribute acts like an index-by table. Its *i*th element stores the number of rows processed by the *i*th execution of an `UPDATE` or `DELETE` statement. If the *i*th execution affects no rows, `%BULK_ROWCOUNT(i)` returns zero.

%BULK_EXCEPTIONS

An associative array that stores information about any exceptions encountered by a `FORALL` statement that uses the `SAVE EXCEPTIONS` clause. You must loop through its elements to determine where the exceptions occurred and what they were. For each index value *i* between 1 and `SQL%BULK_EXCEPTIONS.COUNT`, `SQL%BULK_EXCEPTIONS(i).ERROR_INDEX` specifies which iteration of the `FORALL` loop caused an exception. `SQL%BULK_EXCEPTIONS(i).ERROR_CODE` specifies the Oracle error code that corresponds to the exception.

%FOUND

Returns `TRUE` if an `INSERT`, `UPDATE`, or `DELETE` statement affected one or more rows or a `SELECT INTO` statement returned one or more rows. Otherwise, it returns `FALSE`.

%ISOPEN

Always returns `FALSE`, because Oracle closes the SQL cursor automatically after executing its associated SQL statement.

%NOTFOUND

The logical opposite of %FOUND. It returns TRUE if an INSERT, UPDATE, or DELETE statement affected no rows, or a SELECT INTO statement returned no rows. Otherwise, it returns FALSE.

%ROWCOUNT

Returns the number of rows affected by an INSERT, UPDATE, or DELETE statement, or returned by a SELECT INTO statement.

SQL

The name of the Oracle implicit cursor.

Usage Notes

You can use cursor attributes in procedural statements but not in SQL statements. Before Oracle opens the SQL cursor automatically, the implicit cursor attributes return NULL. The values of cursor attributes always refer to the most recently executed SQL statement, wherever that statement appears. It might be in a different scope. If you want to save an attribute value for later use, assign it to a variable immediately.

If a SELECT INTO statement fails to return a row, PL/SQL raises the predefined exception NO_DATA_FOUND, whether you check SQL%NOTFOUND on the next line or not. A SELECT INTO statement that invokes a SQL aggregate function never raises NO_DATA_FOUND, because those functions always return a value or a NULL. In such cases, SQL%NOTFOUND returns FALSE. %BULK_ROWCOUNT is not maintained for bulk inserts because a typical insert affects only one row. See ["Counting Rows Affected by FORALL \(%BULK_ROWCOUNT Attribute\)"](#) on page 12-16.

You can use the scalar attributes %FOUND, %NOTFOUND, and %ROWCOUNT with bulk binds. For example, %ROWCOUNT returns the total number of rows processed by all executions of the SQL statement. Although %FOUND and %NOTFOUND refer only to the last execution of the SQL statement, you can use %BULK_ROWCOUNT to infer their values for individual executions. For example, when %BULK_ROWCOUNT(i) is zero, %FOUND and %NOTFOUND are FALSE and TRUE, respectively.

Examples

[Example 6-7, "Using SQL%FOUND"](#) on page 6-8
[Example 6-8, "Using SQL%ROWCOUNT"](#) on page 6-8
[Example 6-10, "Fetching with a Cursor"](#) on page 6-11
[Example 6-14, "Using %FOUND"](#) on page 6-13
[Example 6-15, "Using %ISOPEN"](#) on page 6-14
[Example 6-16, "Using %NOTFOUND"](#) on page 6-14
[Example 6-17, "Using %ROWCOUNT"](#) on page 6-15
[Example 12-7, "Using %BULK_ROWCOUNT with the FORALL Statement"](#) on page 12-16

Related Topics

["Cursor Declaration"](#) on page 13-41
["Cursor Attributes"](#) on page 13-35
["FORALL Statement"](#) on page 13-71

SQLCODE Function

The function `SQLCODE` returns the number code of the most recent exception.

For internal exceptions, `SQLCODE` returns the number of the associated Oracle error. The number that `SQLCODE` returns is negative unless the Oracle error is *no data found*, in which case `SQLCODE` returns +100. For user-defined exceptions, `SQLCODE` returns +1, or a value you assign if the exception is associated with an Oracle error number through pragma `EXCEPTION_INIT`.

Syntax

sqlcode_function ::=

→ SQLCODE →

Usage Notes

`SQLCODE` is only useful in an exception handler. Outside a handler, `SQLCODE` always returns 0. `SQLCODE` is especially useful in the `OTHERS` exception handler, because it lets you identify which internal exception was raised. You cannot use `SQLCODE` directly in a SQL statement. Assign the value of `SQLCODE` to a local variable first.

When using pragma `RESTRICT_REFERENCES` to assert the purity of a stored function, you cannot specify the constraints `WNPS` and `RNPS` if the function invokes `SQLCODE`.

Examples

[Example 11-11, "Displaying SQLCODE and SQLERRM"](#) on page 11-15

Related Topics

["Exception Definition"](#) on page 13-48

["SQLERRM Function"](#) on page 13-140

["Retrieving the Error Code and Error Message \(SQLCODE and SQLERRM Functions\)"](#) on page 11-14

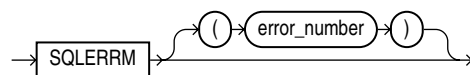
SQLERRM Function

The function `SQLERRM` returns the error message associated with its `error-number` argument. If the argument is omitted, it returns the error message associated with the current value of `SQLCODE`. `SQLERRM` with no argument is useful only in an exception handler. Outside a handler, `SQLERRM` with no argument always returns the `normal, successful completion` message. For internal exceptions, `SQLERRM` returns the message associated with the Oracle error that occurred. The message begins with the Oracle error code.

For user-defined exceptions, `SQLERRM` returns the message user-defined exception, unless you used the pragma `EXCEPTION_INIT` to associate the exception with an Oracle error number, in which case `SQLERRM` returns the corresponding error message. For more information, see ["Retrieving the Error Code and Error Message \(SQLCODE and SQLERRM Functions\)"](#) on page 11-14.

Syntax

sqlerrm_function ::=



Keyword and Parameter Description

error_number

A valid Oracle error number. For a list of Oracle errors (which are prefixed by ORA-), see *Oracle Database Error Messages*.

Usage Notes

`SQLERRM` is especially useful in the `OTHERS` exception handler, where it lets you identify which internal exception was raised. The error number passed to `SQLERRM` must be negative. Passing a zero to `SQLERRM` always returns the following message:

```
ORA-0000: normal, successful completion
```

Passing a positive number to `SQLERRM` always returns the User-Defined Exception message unless you pass `+100`, in which case `SQLERRM` returns the `ORA-01403: no data found` message.

You cannot use `SQLERRM` directly in a SQL statement. Assign the value of `SQLERRM` to a local variable first.

When using pragma `RESTRICT_REFERENCES` to assert the purity of a stored function, you cannot specify the constraints `WNPS` and `RNPS` if the function invokes `SQLERRM`.

Examples

[Example 11-11, "Displaying SQLCODE and SQLERRM"](#) on page 11-15

Related Topics

["Exception Definition"](#) on page 13-48

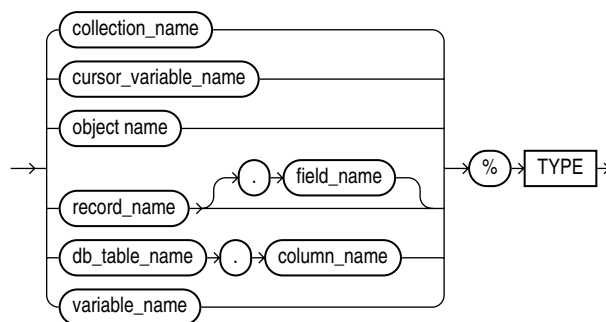
["SQLCODE Function"](#) on page 13-139

%TYPE Attribute

The %TYPE attribute lets use the datatype of a field, record, nested table, database column, or variable in your own declarations, rather than hardcoding the type names. You can use the %TYPE attribute as a datatype specifier when declaring constants, variables, fields, and parameters. If the types that you reference change, your declarations are automatically updated. This technique saves you from making code changes when, for example, the length of a VARCHAR2 column is increased. Note that column constraints, such as the NOT NULL and check constraint, or default values are not inherited by items declared using %TYPE. For more information, see ["Using the %TYPE Attribute"](#) on page 2-10.

Syntax

%type_attribute ::=



Keyword and Parameter Description

collection_name

A nested table, index-by table, or varray previously declared within the current scope.

cursor_variable_name

A PL/SQL cursor variable previously declared within the current scope. Only the value of another cursor variable can be assigned to a cursor variable.

db_table_name.column_name

A table and column that must be accessible when the declaration is elaborated.

object_name

An instance of an object type, previously declared within the current scope.

record_name

A user-defined or %ROWTYPE record, previously declared within the current scope.

record_name.field_name

A field in a user-defined or %ROWTYPE record, previously declared within the current scope.

variable_name

A variable, previously declared in the same scope.

Usage Notes

The %TYPE attribute is particularly useful when declaring variables, fields, and parameters that refer to database columns. Your code can keep working even when the lengths or types of the columns change.

Examples

- [Example 1–16, "Using a PL/SQL Collection Type" on page 1-18](#)
- [Example 2–6, "Using %TYPE with the Datatype of a Variable" on page 2-10](#)
- [Example 2–7, "Using %TYPE with Table Columns" on page 2-11](#)
- [Example 2–15, "Using a Subprogram Name for Name Resolution" on page 2-16](#)
- [Example 2–10, "Assigning Values to a Record with a %ROWTYPE Declaration" on page 2-13](#)
- [Example 3–14, "Using SUBTYPE with %TYPE and %ROWTYPE" on page 3-25](#)
- [Example 5–5, "Declaring a Procedure Parameter as a Nested Table" on page 5-9](#)
- [Example 5–7, "Specifying Collection Element Types with %TYPE and %ROWTYPE" on page 5-9](#)
- [Example 5–42, "Declaring and Initializing Record Types" on page 5-31](#)
- [Example 6–1, "Data Manipulation with PL/SQL" on page 6-1](#)
- [Example 6–13, "Fetching Bulk Data with a Cursor" on page 6-12](#)

Related Topics

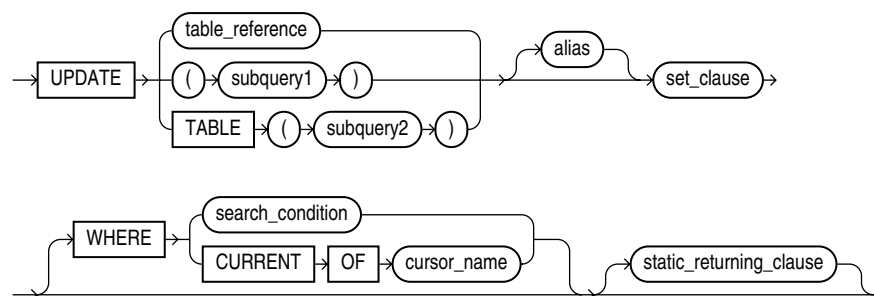
- ["Constant and Variable Declaration" on page 13-29](#)
- ["%ROWTYPE Attribute" on page 13-125](#)

UPDATE Statement

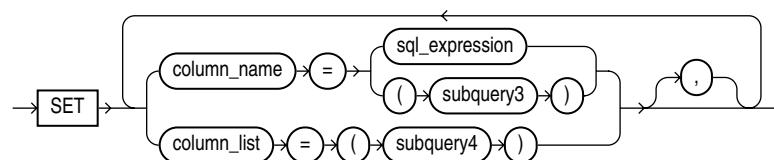
The `UPDATE` statement changes the values of specified columns in one or more rows in a table or view. For a full description of the SQL statement `UPDATE`, see *Oracle Database SQL Language Reference*.

Syntax

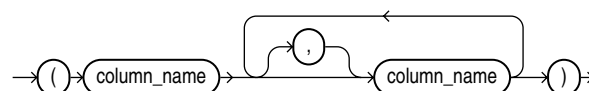
update_statement ::=



set_clause ::=



column_list ::=



Keyword and Parameter Description

alias

Another (usually short) name for the referenced table or view, typically used in the `WHERE` clause.

column_name

The column (or one of the columns) to be updated. It must be the name of a column in the referenced table or view. A column name cannot be repeated in the `column_name` list. Column names need not appear in the `UPDATE` statement in the same order that they appear in the table or view.

SET column_name = sql_expression

This clause assigns the value of `sql_expression` to the column identified by `column_name`. If `sql_expression` contains references to columns in the table being updated, the references are resolved in the context of the current row. The old column values are used on the right side of the equal sign.

SET column_name = (subquery3)

Assigns the value retrieved from the database by `subquery3` to the column identified by `column_name`. The subquery must return exactly one row and one column.

SET (column_name, column_name, ...) = (subquery4)

Assigns the values retrieved from the database by `subquery4` to the columns in the `column_name` list. The subquery must return exactly one row that includes all the columns listed. The column values returned by the subquery are assigned to the columns in the column list in order. The first value is assigned to the first column in the list, the second value is assigned to the second column in the list, and so on.

sql_expression

Any valid SQL expression. For more information, see *Oracle Database SQL Language Reference*.

static_returning_clause

Returns the column values of the updated rows, in either individual variables or collections (eliminating the need to `SELECT` the rows first). For details, see "[RETURNING INTO Clause](#)" on page 13-121.

subquery

A `SELECT` statement that provides a set of rows for processing. Its syntax is like that of `select_into_statement` without the `INTO` clause. See "[SELECT INTO Statement](#)" on page 13-128.

table_reference

A table or view that must be accessible when you execute the `UPDATE` statement, and for which you must have `UPDATE` privileges. For the syntax of `table_reference`, see "[DELETE Statement](#)" on page 13-45.

TABLE (subquery2)

The operand of `TABLE` is a `SELECT` statement that returns a single column value, which must be a nested table or a varray. Operator `TABLE` informs Oracle that the value is a collection, not a scalar value.

WHERE CURRENT OF cursor_name

Refers to the latest row processed by the `FETCH` statement associated with the specified cursor. The cursor must be `FOR UPDATE` and must be open and positioned on a row. If the cursor is not open, the `CURRENT OF` clause causes an error. If the cursor is open, but no rows were fetched or the last fetch returned no rows, PL/SQL raises the predefined exception `NO_DATA_FOUND`.

WHERE search_condition

Chooses which rows to update in the database table. Only rows that meet the search condition are updated. If you omit this clause, all rows in the table are updated.

Usage Notes

You can use the `UPDATE WHERE CURRENT OF` statement after a fetch from an open cursor (including fetches done by a cursor `FOR` loop), provided the associated query is `FOR UPDATE`. This statement updates the row that was just fetched.

The implicit cursor `SQL` and the cursor attributes `%NOTFOUND`, `%FOUND`, `%ROWCOUNT`, and `%ISOPEN` let you access useful information about the execution of an `UPDATE` statement.

Examples

[Example 1–13, "Creating a Stored Subprogram" on page 1-14](#)
[Example 4–1, "Using a Simple IF-THEN Statement" on page 4-2](#)
[Example 5–51, "Updating a Row Using a Record" on page 5-37](#)
[Example 5–52, "Using the RETURNING INTO Clause with a Record" on page 5-37](#)
[Example 6–1, "Data Manipulation with PL/SQL" on page 6-1](#)
[Example 6–5, "Using CURRVAL and NEXTVAL" on page 6-4](#)
[Example 6–6, "Using ROWNUM" on page 6-6](#)
[Example 6–38, "Using SAVEPOINT with ROLLBACK" on page 6-35](#)
[Example 6–41, "Using CURRENT OF to Update the Latest Row Fetched from a Cursor" on page 6-38](#)
[Example 7–12, "Dynamic SQL" on page 7-14](#)
[Example 12–6, "Using Rollbacks with FORALL" on page 12-15](#)
[Example 12–9, "Bulk Operation that Continues Despite Exceptions" on page 12-18](#)

Related Topics

["Data Manipulation Language \(DML\) Statements" on page 6-1](#)
["DELETE Statement" on page 13-45](#)
["FETCH Statement" on page 13-67](#)
["INSERT Statement" on page 13-85](#)

Wrapping PL/SQL Source Code

This appendix describes how to use the standalone `wrap` utility and subprograms of the `DBMS_DDL` package to obfuscate, or wrap, PL/SQL source code. When you obfuscate (hide) PL/SQL units, you can deliver PL/SQL applications without exposing your source code and implementation details.

Topics:

- [What is Wrapping?](#)
- [Wrapping PL/SQL Code with PL/SQL wrap Utility](#)
- [Wrapping PL/QL Code with DBMS_DDL Subprograms](#)
- [Using DBMS_DDL create_wrapped Procedure](#)

See Also: For information on the `DBMS_DDL` package, see *Oracle Database PL/SQL Packages and Types Reference*

What is Wrapping?

Obfuscation, or wrapping, of a PL/SQL unit is the process of hiding the PL/SQL source code. Wrapping can be done with the `wrap` utility and `DBMS_DDL` subprograms. The `wrap` utility is run from the command line and processes an input SQL file, such as a SQL*Plus installation script. The `DBMS_DDL` subprograms wrap a single PL/SQL unit, such as a single `CREATE PROCEDURE` statement, that was generated dynamically.

The advantages of obfuscating, or hiding, the source code of PL/SQL units with the `wrap` utility or `wrap` subprograms of the `DBMS_DDL` package are:

- It is difficult for other developers to misuse your application, or business competitors to see your algorithms.
- Your source code is not visible through the static data dictionary views `*_SOURCE`.
- SQL*Plus can process the obfuscated source files.
- The Import and Export utilities accept wrapped files. You can back up or move wrapped subprograms.

Topics:

- [Guidelines for Wrapping PL/SQL Units](#)
- [Limitations of Wrapping](#)

Guidelines for Wrapping PL/SQL Units

When obfuscating (wrapping) PL/SQL units note the following:

- When wrapping a package or object type, wrap only the body, not the specification. This allows other developers see the information they need to use the package or type, but they cannot see its implementation.
- PL/SQL source inside wrapped files cannot be edited. To change wrapped PL/SQL code, edit the original source file and wrap it again. You can either hold off on wrapping your code until it is ready for shipment to end-users, or include the wrapping operation as part of your build environment.
- To be sure that all the important parts of your source code are obfuscated, view the wrapped file in a text editor before distributing it.

Limitations of Wrapping

The following are limitations when obfuscating PL/SQL source code:

- Although wrapping a compilation unit helps to hide the algorithm and makes reverse-engineering difficult, Oracle does not recommend it as a secure method for hiding passwords or table names. Obfuscating a PL/SQL unit prevents most users from examining the source code, but might not stop all attempts.
- The wrapping does not obfuscate the source code for triggers. To hide the workings of a trigger, you can write a one-line trigger that invokes a wrapped subprogram.
- Wrapping only detects tokenization errors, such as a runaway string, when obfuscating PL/SQL code. Wrapping does not detect syntax or semantic errors, such as tables or views that do not exist. Those errors are detected during PL/SQL compilation or when executing the output file in SQL*Plus.
- Obfuscated PL/SQL program units cannot be imported into a database of a previous (lower) release. Wrapped compilation units are upward-compatible between Oracle releases, but are not downward-compatible. For example, you can load files processed by the V8.1.5 `wrap` utility into a V8.1.6 Oracle database, but you cannot load files processed by the V8.1.6 `wrap` utility into a V8.1.5 Oracle database.

Topics:

- [Limitations of the PL/SQL wrap Utility](#)
- [Limitations of the DBMS_DDL wrap Function](#)

Limitations of the PL/SQL wrap Utility

- Because the source code is parsed by the PL/SQL compiler, not by SQL*Plus, you cannot include substitution variables using the SQL*Plus `DEFINE` notation inside the PL/SQL code. You can use substitution variables in other SQL statements that are not obfuscated.
- Most of the comments are removed in wrapped files. See "[Input and Output Files for the PL/SQL wrap Utility](#)" on page A-3.

Limitations of the DBMS_DDL wrap Function

- If you invoke `DBMS_SQL.PARSE` (when using an overload where the statement formal has datatype `VARCHAR2A` or `VARCHAR2S` for text which exceeds 32767 bytes) on the output of `DBMS_DDL.WRAP`, then you need to set the `LFFLG`

parameter to FALSE. Otherwise DBMS_SQL.PARSE adds newlines to the wrapped unit which corrupts the unit.

Wrapping PL/SQL Code with PL/SQL wrap Utility

The `wrap` utility processes an input SQL file and obfuscates only the PL/SQL units in the file, such as a package specification, package body, function, procedure, type specification, or type body. It does not obfuscate PL/SQL content in anonymous blocks or triggers or non-PL/SQL code.

To run the wrap utility, enter the `wrap` command at your operating system prompt using the following syntax (with no spaces around the equal signs):

```
wrap iname=input_file [ oname=output_file ]
```

input_file is the name of a file containing SQL statements, that you typically run using SQL*Plus. If you omit the file extension, an extension of `.sql` is assumed. For example, the following commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql
```

You can also specify a different file extension:

```
wrap iname=/mydir/myfile.src
```

output_file is the name of the obfuscated file that is created. The defaults to that of the input file and its extension default is `.plb`. For example, the following commands are equivalent:

```
wrap iname=/mydir/myfile
wrap iname=/mydir/myfile.sql oname=/mydir/myfile.plb
```

You can use the option `oname` to specify a different file name and extension:

```
wrap iname=/mydir/myfile oname=/yourdir/yourfile.out
```

Topics:

- [Input and Output Files for the PL/SQL wrap Utility](#)
- [Running the wrap Utility](#)

Input and Output Files for the PL/SQL wrap Utility

The input file can contain any combination of SQL statements. Most statements are passed through unchanged. CREATE statements that define subprograms, packages, or object types are obfuscated; their bodies are replaced by a scrambled form that the PL/SQL compiler understands.

The following CREATE statements are obfuscated:

```
CREATE [OR REPLACE] FUNCTION function_name
CREATE [OR REPLACE] PROCEDURE procedure_name
CREATE [OR REPLACE] PACKAGE package_name
CREATE [OR REPLACE] PACKAGE BODY package_name
CREATE [OR REPLACE] TYPE type_name AS OBJECT
CREATE [OR REPLACE] TYPE type_name UNDER type_name
CREATE [OR REPLACE] TYPE BODY type_name
```

The CREATE [OR REPLACE] TRIGGER statement, and [DECLARE] BEGIN-END anonymous blocks, are not obfuscated. All other SQL statements are passed unchanged to the output file.

All comment lines in the unit being wrapped are deleted, except for those in a CREATE OR REPLACE header and C-style comments (delimited by /* */).

The output file is a text file, which you can run as a script in SQL*Plus to set up your PL/SQL subprograms and packages. Run a wrapped file as follows:

```
SQL> @wrapped_file_name.plb;
```

Running the wrap Utility

For example, assume that the wrap_test.sql file contains the following:

```
CREATE PROCEDURE wraptest IS
  TYPE emp_tab IS TABLE OF employees%ROWTYPE INDEX BY PLS_INTEGER;
  all_emps      emp_tab;
BEGIN
  SELECT * BULK COLLECT INTO all_emps FROM employees;
  FOR i IN 1..10 LOOP
    DBMS_OUTPUT.PUT_LINE('Emp Id: ' || all_emps(i).employee_id);
  END LOOP;
END;
/
```

To wrap the file, run the following from the operating system prompt:

```
wrap iname=wrap_test.sql
```

The output of the wrap utility is similar to the following:

```
PL/SQL Wrapper: Release 10.2.0.0.0 on Tue Apr 26 16:47:39 2005
Copyright (c) 1993, 2005, Oracle. All rights reserved.
Processing wrap_test.sql to wrap_test.plb
```

If you view the contents of the wrap_test.plb text file, the first line is CREATE PROCEDURE wraptest wrapped and the rest of the file contents is hidden.

You can run wrap_test.plb in SQL*Plus to execute the SQL statements in the file:

```
SQL> @wrap_test.plb
```

After the wrap_test.plb is run, you can execute the procedure that was created:

```
SQL> CALL wraptest();
```

Wrapping PL/QL Code with DBMS_DDL Subprograms

The DBMS_DDL package contains procedures for obfuscating a single PL/SQL unit, such as a package specification, package body, function, procedure, type specification, or type body. These overloaded subprograms provide a mechanism for obfuscating dynamically generated PL/SQL program units that are created in a database.

The DBMS_DDL package contains the WRAP function and the CREATE_WRAPPED procedure. The CREATE_WRAPPED both wraps the text and creates the PL/SQL unit. When invoking the wrap procedures, use the fully-qualified package name, SYS.DBMS_DDL, to avoid any naming conflicts and the possibility that someone might create a local package called DBMS_DDL or define the DBMS_DDL public synonym. The input CREATE OR REPLACE statement executes with the privileges of the user who invokes DBMS_DDL.WRAP or DBMS_DDL.CREATE_WRAPPED.

The DBMS_DDL package also provides the MALFORMED_WRAP_INPUT exception (ORA-24230) which is raised if the input to the wrap procedures is not a valid PL/SQL unit.

Using DBMS_DDL create_wrapped Procedure

In [Example A-1](#) CREATE_WRAPPED is used to dynamically create and wrap a package specification and a package body in a database.

Example A-1 Using the create_wrapped Procedure to Wrap a Package

```

DECLARE
-- the package_text variable contains the text
-- to create the package spec and body
package_text VARCHAR2(32767);
FUNCTION generate_spec (pkgname VARCHAR2) RETURN VARCHAR2 AS
BEGIN
    RETURN 'CREATE PACKAGE ' || pkgname || ' AS
        PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER);
        PROCEDURE fire_employee (emp_id NUMBER);
        END ' || pkgname || ';' ;
END generate_spec;
FUNCTION generate_body (pkgname VARCHAR2) RETURN VARCHAR2 AS
BEGIN
    RETURN 'CREATE PACKAGE BODY ' || pkgname || ' AS
        PROCEDURE raise_salary (emp_id NUMBER, amount NUMBER) IS
        BEGIN
            UPDATE employees
            SET salary = salary + amount WHERE employee_id = emp_id;
        END raise_salary;
        PROCEDURE fire_employee (emp_id NUMBER) IS
        BEGIN
            DELETE FROM employees WHERE employee_id = emp_id;
        END fire_employee;
        END ' || pkgname || ';' ;
END generate_body;

BEGIN
    -- generate package spec
    package_text := generate_spec('emp_actions')
    -- create and wrap the package spec
    SYS.DBMS_DDL.CREATE_WRAPPED(package_text);
    -- generate package body
    package_text := generate_body('emp_actions');
    -- create and wrap the package body
    SYS.DBMS_DDL.CREATE_WRAPPED(package_text);
END;
/

-- invoke a procedure from the wrapped package
CALL emp_actions.raise_salary(120, 100);

```

When you check the static data dictionary views *_SOURCE, the source is wrapped, or hidden, so that others cannot view the code details. For example:

```
SELECT text FROM USER_SOURCE WHERE name = 'EMP_ACTIONS';
```

The resulting output is similar to the following:

```
TEXT
```

```
-----  
PACKAGE emp_actions WRAPPED  
a000000  
1f  
abcd  
...
```

How PL/SQL Resolves Identifier Names

This appendix explains how PL/SQL resolves references to names in potentially ambiguous SQL and procedural statements.

Topics:

- [What is Name Resolution?](#)
- [Examples of Qualified Names and Dot Notation](#)
- [How Name Resolution Differs in PL/SQL and SQL](#)
- [What is Capture?](#)
- [Avoiding Inner Capture in DML Statements](#)

What is Name Resolution?

During compilation, the PL/SQL compiler determines which objects are associated with each name in a PL/SQL subprogram. A name might refer to a local variable, a table, a package, a subprogram, a schema, and so on. When a subprogram is recompiled, that association might change if objects were created or deleted.

A declaration or definition in an inner scope can hide another in an outer scope. In [Example B-1](#), the declaration of variable `client` hides the definition of datatype `Client` because PL/SQL names are not case sensitive:

Example B-1 Resolving Global and Local Variable Names

```
BEGIN
  <<block1>>
  DECLARE
    TYPE Client IS RECORD (
      first_name VARCHAR2(20), last_name VARCHAR2(25));
    TYPE Customer IS RECORD (
      first_name VARCHAR2(20), last_name VARCHAR2(25));
  BEGIN
    DECLARE
      client Customer;
      -- hides definition of type Client in outer scope
      -- lead1 Client;
      -- not allowed; Client resolves to the variable client
      lead2 block1.Client;
      -- OK; refers to type Client
    BEGIN
      -- no processing, just an example of name resolution
      NULL;
    END;
  END;
```

```

    END;
END;
/

```

You can refer to datatype `Client` by qualifying the reference with block label `block1`.

In the following set of `CREATE TYPE` statements, the second statement generates a warning. Creating an attribute named `manager` hides the type named `manager`, so the declaration of the second attribute does not execute correctly.

```

CREATE TYPE manager AS OBJECT (dept NUMBER);
/
CREATE TYPE person AS OBJECT (manager NUMBER, mgr manager)
    -- raises a warning;
/

```

Examples of Qualified Names and Dot Notation

During name resolution, the compiler can encounter various forms of references including simple unqualified names, dot-separated chains of identifiers, indexed components of a collection, and so on. This is shown in [Example B-2](#).

Example B-2 Using the Dot Notation to Qualify 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
        -- (1) unqualified name
        n := m;
        -- (2) dot-separated chain of identifiers
        -- (package name used as scope qualifier
        -- followed by variable name)
        n := pkg1.m;
        -- (3) dot-separated chain of identifiers
        -- (package name used as scope
        -- qualifier followed by function name
        -- also used as scope qualifier
        -- followed by parameter name)
        n := pkg1.f1.p1;
        -- (4) dot-separated chain of identifiers
        -- (variable name followed by
        -- component selector)
        n := v1.a;
        -- (5) dot-separated chain of identifiers
        -- (package name used as scope
        -- qualifier followed by variable name
        -- followed by component selector)
        n := pkg1.v1.a;
    END;

```

```

-- (6) indexed name followed by component selector
      n := v2(10).a;
-- (7) function call followed by component selector
      n := f1(10).a;
-- (8) function call followed by indexing followed by
--     component selector
      n := f2(10)(10).a;
-- (9) function call (which is a dot-separated
--     chain of identifiers, including schema name used
--     as scope qualifier followed by package name used
--     as scope qualifier followed by function name)
--     followed by component selector of the returned
--     result followed by indexing followed by component selector
      n := hr.pkg1.f2(10)(10).a;
-- (10) variable name followed by component selector
      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;
/

```

Note that an outside reference to a private variable declared in a function body is not legal. For example, an outside reference to the variable `n` declared in function `f1`, such as `hr.pkg1.f1.n` from function `f2`, raises an error. See ["Private and Public Items in PL/SQL Packages"](#) on page 10-9.

Dot notation is used for identifying record fields, object attributes, and items inside packages or other schemas. When you combine these items, you might need to use expressions with multiple levels of dots, where it is not always clear what each dot refers to. Here are some of the combinations:

- 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 object type, for example:

```

record_name.field_name.attribute_name
record_name.field_name.member_name()

```

How Name Resolution Differs in PL/SQL and SQL

The name resolution rules for PL/SQL and SQL are similar. You can avoid the few differences if you follow the capture avoidance rules. For compatibility, the SQL rules are more permissive than the PL/SQL rules. SQL rules, which are mostly context sensitive, recognize as legal more situations and DML statements than the PL/SQL rules.

- PL/SQL uses the same name-resolution rules as SQL when the PL/SQL compiler processes a SQL statement, such as a DML statement. For example, for a name such as HR.JOBS, SQL matches objects in the HR schema first, then packages, types, tables, and views in the current schema.
- PL/SQL uses a different order to resolve names in PL/SQL statements such as assignments and subprogram calls. In the case of a name HR.JOBS, PL/SQL searches first for packages, types, tables, and views named HR in the current schema, then for objects in the HR schema.

For information on SQL naming rules, see *Oracle Database SQL Language Reference*.

What is Capture?

When a declaration or type definition in another scope prevents the compiler from resolving a reference correctly, that declaration or definition is said to capture the reference. Usually this is the result of migration or schema evolution. There are three kinds of capture: inner, same-scope, and outer. Inner and same-scope capture apply only in SQL scope.

Topics:

- [Inner Capture](#)
- [Same-Scope Capture](#)
- [Outer Capture](#)

Inner Capture

An inner capture occurs when a name in an inner scope no longer refers to an entity in an outer scope:

- The name might now resolve to an entity in an inner scope.
- The program might cause an error, if some part of the identifier is captured in an inner scope and the complete reference cannot be resolved.

If the reference points to a different but valid name, you might not know why the program is acting differently.

In the following example, the reference to `col2` in the inner `SELECT` statement binds to column `col2` in table `tab1` because table `tab2` has no column named `col2`:

```
CREATE TABLE tab1 (col1 NUMBER, col2 NUMBER);
INSERT INTO tab1 VALUES (100, 10);
CREATE TABLE tab2 (col1 NUMBER);
INSERT INTO tab2 VALUES (100);

CREATE OR REPLACE PROCEDURE proc AS
  CURSOR c1 IS SELECT * FROM tab1
    WHERE EXISTS (SELECT * FROM tab2 WHERE col2 = 10);
BEGIN
  NULL;
```

```
END;
/
```

In the preceding example, if you add a column named `col2` to table `tab2`:

```
ALTER TABLE tab2 ADD (col2 NUMBER);
```

then procedure `proc` is invalidated and recompiled automatically upon next use. However, upon recompilation, the `col2` in the inner `SELECT` statement binds to column `col2` in table `tab2` because `tab2` is in the inner scope. Thus, the reference to `col2` is captured by the addition of column `col2` to table `tab2`.

Using collections and object types can cause more inner capture situations. In the following example, the reference to `hr.tab2.a` resolves to attribute `a` of column `tab2` in table `tab1` through table alias `hr`, which is visible in the outer scope of the query:

```
CREATE TYPE type1 AS OBJECT (a NUMBER);
/
CREATE TABLE tab1 (tab2 type1);
INSERT INTO tab1 VALUES ( type1(10) );
CREATE TABLE tab2 (x NUMBER);
INSERT INTO tab2 VALUES ( 10 );

-- in the following,
-- alias tab1 with same name as schema name,
-- which is not a good practice
-- but is used here for illustration purpose
-- note lack of alias in second SELECT
SELECT * FROM tab1 hr
      WHERE EXISTS (SELECT * FROM hr.tab2 WHERE x = hr.tab2.a);
```

In the preceding example, you might add a column named `a` to table `hr.tab2`, which appears in the inner subquery. When the query is processed, an inner capture occurs because the reference to `hr.tab2.a` resolves to column `a` of table `tab2` in schema `hr`. You can avoid inner captures by following the rules given in "[Avoiding Inner Capture in DML Statements](#)" on page B-5. According to those rules, revise the query as follows:

```
SELECT * FROM hr.tab1 p1
      WHERE EXISTS (SELECT * FROM hr.tab2 p2 WHERE p2.x = p1.tab2.a);
```

Same-Scope Capture

In SQL scope, a same-scope capture occurs when a column is added to one of two tables used in a join, so that the same column name exists in both tables. Previously, you could refer to that column name in a join query. To avoid an error, now you must qualify the column name with the table name.

Outer Capture

An outer capture occurs when a name in an inner scope, which once resolved to an entity in an inner scope, is resolved to an entity in an outer scope. SQL and PL/SQL are designed to prevent outer captures. You do not need to take any action to avoid this condition.

Avoiding Inner Capture in DML Statements

You can avoid inner capture in DML statements by following these rules:

- Specify an alias for each table in the DML statement.
- Keep table aliases unique throughout the DML statement.
- Avoid table aliases that match schema names used in the query.
- Qualify each column reference with the table alias.

Qualifying a reference with `schema_name.table_name` does not prevent inner capture if the statement refers to tables with columns of a user-defined object type.

Columns of a user-defined object type allow for more inner capture situations. To minimize problems, the name-resolution algorithm includes the following rules for the use of table aliases.

Topics:

- [Qualifying References to Attributes and Methods](#)
- [Qualifying References to Row Expressions](#)

Qualifying References to Attributes and Methods

All references to attributes and methods must be qualified by a table alias. When referencing a table, if you reference the attributes or methods of an object stored in that table, the table name must be accompanied by an alias. As the following examples show, column-qualified references to an attribute or method are not allowed if they are prefixed with a table name:

```
CREATE TYPE t1 AS OBJECT (x NUMBER);
/
CREATE TABLE tb1 (col1 t1);

BEGIN
-- following inserts are allowed without an alias
-- because there is no column list
  INSERT INTO tb1 VALUES ( t1(10) );
  INSERT INTO tb1 VALUES ( t1(20) );
  INSERT INTO tb1 VALUES ( t1(30) );
END;
/
BEGIN
  UPDATE tb1 SET col1.x = 10
  WHERE col1.x = 20; -- error, not allowed
END;
/
BEGIN
  UPDATE tb1 SET tb1.col1.x = 10
  WHERE tb1.col1.x = 20; -- not allowed
END;
/
BEGIN
  UPDATE hr.tb1 SET hr.tb1.col1.x = 10
  WHERE hr.tb1.col1.x = 20; -- not allowed
END;
/
BEGIN -- following allowed with table alias
  UPDATE hr.tb1 t set t.col1.x = 10
  WHERE t.col1.x = 20;
END;
/
DECLARE
  y NUMBER;
```

```

BEGIN -- following allowed with table alias
  SELECT t.col1.x INTO y FROM tbl t
  WHERE t.col1.x = 30;
END;
/
BEGIN
  DELETE FROM tbl
  WHERE tbl.col1.x = 10; -- not allowed
END;
/
BEGIN -- following allowed with table alias
  DELETE FROM tbl t
  WHERE t.col1.x = 10;
END;
/

```

Qualifying References to Row Expressions

Row expressions must resolve as references to table aliases. You can pass row expressions to operators REF and VALUE, and you can use row expressions in the SET clause of an UPDATE statement. Some examples follow:

```

CREATE TYPE t1 AS OBJECT (x number);
/
CREATE TABLE ot1 OF t1;

BEGIN
-- following inserts are allowed without an alias
-- because there is no column list
  INSERT INTO ot1 VALUES ( t1(10) );
  INSERT INTO ot1 VALUES ( 20 );
  INSERT INTO ot1 VALUES ( 30 );
END;
/
BEGIN
  UPDATE ot1 SET VALUE(ot1.x) = t1(20)
  WHERE VALUE(ot1.x) = t1(10); -- not allowed
END;
/
BEGIN -- following allowed with table alias
  UPDATE ot1 o SET o = (t1(20)) WHERE o.x = 10;
END;
/
DECLARE
  n_ref REF t1;
BEGIN -- following allowed with table alias
  SELECT REF(o) INTO n_ref FROM ot1 o
  WHERE VALUE(o) = t1(30);
END;
/
DECLARE
  n t1;
BEGIN -- following allowed with table alias
  SELECT VALUE(o) INTO n FROM ot1 o
  WHERE VALUE(o) = t1(30);
END;
/
DECLARE
  n NUMBER;
BEGIN -- following allowed with table alias

```

```
        SELECT o.x INTO n FROM ot1 o WHERE o.x = 30;
    END;
    /
    BEGIN
        DELETE FROM ot1
        WHERE VALUE(ot1) = (t1(10)); -- not allowed
    END;
    /
    BEGIN -- folowing allowed with table alias
        DELETE FROM ot1 o
        WHERE VALUE(o) = (t1(20));
    END;
    /
```

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 meta-notation called Interface Definition Language (IDL). DIANA is used internally by compilers and other tools.

At compile time, PL/SQL source code is translated into machine-readable code. Both the DIANA and machine-readable 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 machine-readable code is simply executed.

In the shared memory pool, a package spec, object type spec, standalone subprogram, or anonymous block is limited to $67108864 (2^{26})$ DIANA nodes which correspond to tokens such as identifiers, keywords, operators, and so on. This allows for ~6,000,000 lines of code unless you exceed limits imposed by the PL/SQL compiler, some of which are given in [Table C-1](#).

Table C-1 PL/SQL Compiler Limits

Item	Limit
bind variables passed to a program unit	32768
exception handlers in a program unit	65536
fields in a record	65536
levels of block nesting	255
levels of record nesting	32
levels of subquery nesting	254
levels of label nesting	98
levels of nested collections	no predefined limit
magnitude of a PLS_INTEGER or BINARY_INTEGERvalue	-2147483648..2147483647
number of formal parameters in an explicit cursor, function, or procedure	65536
objects referenced by a program unit	65536
precision of a FLOAT value (binary digits)	126
precision of a NUMBER value (decimal digits)	38
precision of a REAL value (binary digits)	63

Table C-1 (Cont.) PL/SQL Compiler Limits

Item	Limit
size of an identifier (characters)	30
size of a string literal (bytes)	32767
size of a CHAR value (bytes)	32767
size of a LONG value (bytes)	32760
size of a LONG RAW value (bytes)	32760
size of a RAW value (bytes)	32767
size of a VARCHAR2 value (bytes)	32767
size of an NCHAR value (bytes)	32767
size of an NVARCHAR2 value (bytes)	32767
size of a BFILE value (bytes)	4G * value of DB_BLOCK_SIZE parameter
size of a BLOB value (bytes)	4G * value of DB_BLOCK_SIZE parameter
size of a CLOB value (bytes)	4G * value of DB_BLOCK_SIZE parameter
size of an NCLOB value (bytes)	4G * value of DB_BLOCK_SIZE parameter

To estimate how much memory a program unit requires, you can query the static data dictionary view `USER_OBJECT_SIZE`. The column `PARSED_SIZE` returns the size (in bytes) of the "flattened" DIANA. For example:

```
SQL> SELECT * FROM user_object_size WHERE name = 'PKG1';
```

NAME	TYPE	SOURCE_SIZE	PARSED_SIZE	CODE_SIZE	ERROR_SIZE
PKG1	PACKAGE	46	165	119	0
PKG1	PACKAGE BODY	82	0	139	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 object 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 additional information about the limits on datatypes, see [Chapter 3, "PL/SQL Datatypes"](#). For limits on collection subscripts, see ["Referencing Collection Elements"](#) on page 5-12.

PL/SQL Reserved Words and Keywords

Both **reserved words** and **keywords** have special syntactic meaning to PL/SQL. The difference between reserved words and keywords is that you cannot use reserved words as identifiers. You can use keywords as as identifiers, but it is not recommended.

[Table D-1](#) lists the PL/SQL reserved words.

[Table D-2](#) lists the PL/SQL keywords.

Some of the words in this appendix are also reserved by SQL. You can display them with the dynamic performance view `V$RESERVED_WORDS`, which is described in *Oracle Database Reference*.

Table D-1 PL/SQL Reserved Words

Begins with:	Reserved Words
A	ALL, ALTER, AND, ANY, ARRAY, ARROW, AS, ASC, AT
B	BEGIN, BETWEEN, BY
C	CASE, CHECK, CLUSTERS, CLUSTER, COLAUTH, COLUMNS, COMPRESS, CONNECT, CRASH, CREATE, CURRENT
D	DECIMAL, DECLARE, DEFAULT, DELETE, DESC, DISTINCT, DROP
E	ELSE, END, EXCEPTION, EXCLUSIVE, EXISTS
F	FETCH, FORM, FOR, FROM
G	GOTO, GRANT, GROUP
H	HAVING
I	IDENTIFIED, IF, IN, INDEXES, INDEX, INSERT, INTERSECT, INTO, IS
L	LIKE, LOCK
M	MINUS, MODE
N	NOCOMPRESS, NOT, NOWAIT, NULL
O	OF, ON, OPTION, OR, ORDER, OVERLAPS
P	PRIOR, PROCEDURE, PUBLIC
R	RANGE, RECORD, RESOURCE, REVOKE
S	SELECT, SHARE, SIZE, SQL, START, SUBTYPE
T	TABAUTH, TABLE, THEN, TO, TYPE
U	UNION, UNIQUE, UPDATE, USE
V	VALUES, VIEW, VIEWS
W	WHEN, WHERE, WITH

Table D–2 PL/SQL Keywords

Begins with:	Keywords
A	A, ADD, AGENT, AGGREGATE, ARRAY, ATTRIBUTE, AUTHID, AVG
B	BFILE_BASE, BINARY, BLOB_BASE, BLOCK, BODY, BOTH, BOUND, BULK, BYTE
C	C, CALL, CALLING, CASCADE, CHAR, CHAR_BASE, CHARACTER, CHARACTERFORM, CHARACTERID, CHARSET, CLOB_BASE, CLOSE, COLLECT, COMMENT, COMMIT, COMMITTED, COMPILED, CONSTANT, CONSTRUCTOR, CONTEXT, CONTINUE, CONVERT, COUNT, CURSOR, CUSTOMDATUM
D	DANGLING, DATA, DATE, DATE_BASE, DAY, DEFINE, DETERMINISTIC, DOUBLE, DURATION
E	ELEMENT, ELSIF, EMPTY, ESCAPE, EXCEPT, EXCEPTIONS, EXECUTE, EXIT, EXTERNAL
F	FINAL, FIXED, FLOAT, FORALL, FORCE, FUNCTION
G	GENERAL
H	HASH, HEAP, HIDDEN, HOUR
I	IMMEDIATE, INCLUDING, INDICATOR, INDICES, INFINITE, INSTANTIABLE, INT, INTERFACE, INTERVAL, INVALIDATE, ISOLATION
J	JAVA
L	LANGUAGE, LARGE, LEADING, LENGTH, LEVEL, LIBRARY, LIKE2, LIKE4, LIKEC, LIMIT, LIMITED, LOCAL, LONG, LOOP
M	MAP, MAX, MAXLEN, MEMBER, MERGE, MIN, MINUTE, MOD, MODIFY, MONTH, MULTISSET
N	NAME, NAN, NATIONAL, NATIVE, NCHAR, NEW, NOCOPY, NUMBER_BASE
O	OBJECT, OCICOLL, OCIDATETIME, OCIDATE, OCIDURATION, OCINTERVAL, OCIOBLOCATOR, OCINUMBER, OCIRAW, OCIREFCURSOR, OCIREF, OCIROWID, OCISTRING, OCITYPE, ONLY, OPAQUE, OPEN, OPERATOR, ORACLE, ORADATA, ORGANIZATION, ORLANY, ORLVARY, OTHERS, OUT, OVERRIDING
P	PACKAGE, PARALLEL_ENABLE, PARAMETER, PARAMETERS, PARTITION, PASCAL, PIPE, PIPELINED, PRAGMA, PRECISION, PRIVATE
R	RAISE, RANGE, RAW, READ, RECORD, REF, REFERENCE, RELIES_ON, REM, REMAINDER, RENAME, RESULT, RESULT_CACHE, RETURN, RETURNING, REVERSE, ROLLBACK, ROW
S	SAMPLE, SAVE, SAVEPOINT, SB1, SB2, SB4, SECOND, SEGMENT, SELF, SEPARATE, SEQUENCE, SERIALIZABLE, SET, SHORT, SIZE_T, SOME, SPARSE, SQLCODE, SQLDATA, SQLNAME, SQLSTATE, STANDARD, STATIC, STDDEV, STORED, STRING, STRUCT, STYLE, SUBMULTISET, SUBPARTITION, SUBSTITUTABLE, SUBTYPE, SUM, SYNONYM
T	TDO, THE, TIME, TIMESTAMP, TIMEZONE_ABBR, TIMEZONE_HOUR, TIMEZONE_MINUTE, TIMEZONE_REGION, TRAILING, TRANSACTION, TRANSACTIONAL, TRUSTED, TYPE
U	UB1, UB2, UB4, UNDER, UNSIGNED, UNTRUSTED, USE, USING
V	VALIST, VALUE, VARIABLE, VARIANCE, VARRAY, VARYING, VOID
W	WHILE, WORK, WRAPPED, WRITE
Y	YEAR
Z	ZONE

Symbols

`%BULK_EXCEPTIONS`. See `BULK_EXCEPTIONS` cursor attribute
`%BULK_ROWCOUNT`. See `BULK_ROWCOUNT` cursor attribute
`%FOUND`. See `FOUND` cursor attribute
`%ISOPEN`. See `ISOPEN` cursor attribute
`%NOTFOUND`. See `NOTFOUND` cursor attribute
`%ROWCOUNT`. See `ROWCOUNT` cursor attribute
`%ROWTYPE`. *see* `ROWTYPE` attribute
`%TYPE` *see* `TYPE` attribute
`:=` assignment operator, 1-6
`||` concatenation operator, 2-25
`.` item separator, 2-2
`<<` label delimiter, 2-2
`..` range operator, 2-2, 4-12
`=, !=, <>, and ~=` relational operators, 2-24
`<, >, <=, and >=` relational operators, 2-24
`@` remote access indicator, 2-2, 2-14
`--` single-line comment delimiter, 2-2
`;` statement terminator, 2-2, 13-14
`-` subtraction/negation operator, 2-2

A

`ACCESS INTO NULL` exception, 11-4
actual parameters, 6-21
address
 `REF CURSOR`, 6-22
advantages
 `PL/SQL`, 1-1
`AFTER` triggers
 auditing and, 9-31, 9-33
 correlation names and, 9-19
 specifying, 9-6
aggregate assignment, 2-13
aggregate functions
 and `PL/SQL`, 6-3
aliases
 using with a select list, 2-13
aliasing
 for expression values in a cursor `FOR` loop, 6-18
 parameters, 8-27
`ALL` row operator, 6-3, 6-7
`ALL_PLSQL_OBJECT_SETTINGS` view

`PL/SQL` compiler parameter settings, 12-1
`ALTER PROCEDURE`
 using to recompile a procedure, 11-21
`ALTER TABLE` statement
 `DISABLE ALL TRIGGERS` clause, 9-28
 `ENABLE ALL TRIGGERS` clause, 9-28
`ALTER TRIGGER` statement
 `DISABLE` clause, 9-28
 `ENABLE` clause, 9-28
anonymous blocks, 1-5
`ANSI/ISO SQL` standard, 6-1
apostrophes, 2-7
architecture
 `PL/SQL`, 1-20
`ARRAY`
 `VARYING`, 5-7
arrays
 associative, 5-2
 index-by-tables, 5-2
 variable-size, 5-2
assignment operator, 1-6
assignment statement
 links to examples, 13-5
 syntax, 13-3
assignments
 aggregate, 2-13
 collection, 5-13
 field, 5-34
 `IN OUT` parameters, 1-7
 records, 5-34
 variables, 1-6
associative arrays
 nested tables and, 5-6
 sets of key-value pairs, 5-3
 syntax, 13-19
 understanding, 5-2
 `VARCHAR2` keys and globalization settings, 5-4
asynchronous operations, 10-10
attributes
 `%ROWTYPE`, 1-9, 2-12
 `%TYPE`, 1-9, 2-10
 explicit cursors, 6-13
auditing
 triggers and, 9-30
`AUTHID` clause
 specifying privileges for a subprogram, 8-18

- using to specify privileges of invoker, 8-17
- autonomous functions
 - invoking from SQL, 6-46
 - RESTRICT_REFERENCES pragma, 6-46
- autonomous transactions
 - advantages, 6-41
 - avoiding errors, 6-45
 - comparison with nested transactions, 6-43
 - controlling, 6-44
 - in PL/SQL, 6-40
 - SQL%ROWCOUNT attribute, 6-9
- autonomous triggers
 - using, 6-45
- AUTONOMOUS_TRANSACTION pragma
 - defining, 6-41
 - links to examples, 13-7
 - syntax, 13-6
- avoiding SQL injection, 7-9

B

- basic loops, 4-8
- BEFORE triggers
 - complex security authorizations, 9-40
 - correlation names and, 9-19
 - derived column values, 9-41
 - specifying, 9-6
- BEGIN
 - start of executable PL/SQL block, 13-12
 - syntax, 13-12
- BETWEEN clause
 - FORALL, 13-71
- BETWEEN comparison operator, 2-25
 - expressions, 13-61
- BFILE datatype, 3-22
- BINARY_DOUBLE datatype, 3-5
- BINARY_FLOAT and BINARY_DOUBLE datatypes
 - for computation-intensive programs, 12-24
- BINARY_FLOAT datatype, 3-5
- BINARY_INTEGER datatype
 - see PLS_INTEGER datatype
- bind arguments
 - preventing SQL injection with, 7-12
- bind variables, 1-7
- binding
 - bulk, 12-11
 - variables, 12-11
- BLOB datatype, 3-22
- blocks
 - label, 2-18
 - links to examples, 13-15
 - nesting, 1-5
 - PL/SQL
 - syntax, 13-8
 - structure, 1-5
- BODY
 - CREATE PACKAGE SQL statement, 1-14, 10-1, 13-107
 - CREATE TYPE SQL statement, 13-100
 - with SQL CREATE PACKAGE statement, 1-14,

- 10-1
- body
 - cursor, 10-12
 - package, 10-5
 - packages, 13-108
- Boolean
 - assigning values, 2-20
 - expressions, 2-26
 - literals, 2-7
- BOOLEAN datatype, 3-15
- bulk
 - fetches, 12-20
 - returns, 12-22
- bulk binding, 12-11
 - limitations, 12-11
- BULK clause
 - with COLLECT, 12-19
- BULK COLLECT clause, 12-19
 - checking whether no results are returned, 12-20
 - FETCH, 13-68
 - retrieving DML results, 12-22
 - retrieving query results with, 12-19
 - returning multiple rows, 6-16
 - SELECT INTO, 13-129
 - using LIMIT clause, 12-20, 12-21
 - using ROWNUM pseudocolumn, 12-20
 - using SAMPLE clause, 12-20
 - using with FORALL statement, 12-23
- BULK COLLECT INTO clause
 - in EXECUTE IMMEDIATE statement, 13-52
 - in RETURNING INTO clause, 13-122
- bulk SQL
 - using to reduce loop overhead, 12-11
- BULK_EXCEPTIONS cursor attribute
 - ERROR_CODE field, 12-17
 - ERROR_INDEX field, 12-17
 - example, 12-18
 - handling FORALL exceptions, 12-17
 - using ERROR_CODE field with SQLERRM, 12-17
- BULK_ROWCOUNT cursor attribute
 - affected by FORALL, 12-16
- by-reference parameter passing, 8-27
- by-value parameter passing, 8-27

C

- CALL
 - SQL statement, 1-16
- call specification, 10-2
- calls
 - inter-language, 8-25
 - resolving subprogram, 8-16
 - subprograms, 8-7
- CARDINALITY operator
 - for nested tables, 5-17
- carriage returns, 2-2
- CASE expressions, 2-28
 - overview, 1-10
- case sensitivity
 - identifier, 2-4

- string literal, 2-7
- CASE statement
 - links to examples, 13-17
 - searched, 4-6
 - syntax, 13-16
 - using, 4-4
- CASE_NOT_FOUND exception, 11-4
- CATPROC.SQL script, 9-46
- CHAR datatype, 3-8
 - differences with VARCHAR2, 3-9
- character literals, 2-6
- character sets
 - PL/SQL, 2-1
- CHARACTER subtype, 3-9
- character values
 - comparing, 3-10
- CHECK constraint
 - triggers and, 9-35, 9-39
- clauses
 - AUTHID, 8-17, 8-18
 - BULK COLLECT, 12-19
 - LIMIT, 12-21
- CLOB datatype, 3-22
- CLOSE statement
 - disables cursor, 6-13
 - disabling cursor variable
 - closing, 6-29
 - links to examples, 13-18
 - syntax, 13-18
- collating sequence, 2-26
- COLLECT clause
 - with BULK, 12-19
- collection exceptions
 - when raised, 5-29
- collection methods
 - syntax, 13-23
 - usage, 5-20
- COLLECTION_IS_NULL exception, 11-4
- collections
 - allowed subscript ranges, 5-12
 - applying methods to parameters, 5-28
 - assigning, 5-13
 - associative arrays and nested tables, 5-6
 - avoiding exceptions, 5-28
 - bulk binding, 5-38, 12-11
 - choosing the type to use, 5-5
 - comparing, 5-17
 - constructors, 5-10
 - COUNT method, 5-21
 - declaring variables, 5-8
 - defining types, 5-7
 - DELETE method, 5-27
 - element types, 5-7
 - EXISTS method, 5-21
 - EXTEND method, 5-25
 - FIRST method, 5-22
 - initializing, 5-10
 - LAST method, 5-22
 - LIMIT method, 5-22
 - links to examples, 13-22, 13-25
 - methods, 5-20
 - multilevel, 5-19
 - NEXT method, 5-23
 - operators to transform nested tables, 5-13
 - ordered group of elements, 5-1
 - overview, 1-17
 - PRIOR method, 5-23
 - referencing, 5-10
 - referencing elements, 5-12
 - scope, 5-7
 - syntax, 13-19
 - testing for null, 5-17
 - TRIM method, 5-26
 - types in PL/SQL, 5-1
 - understanding, 5-2
 - varrays and nested tables, 5-6
- column aliases
 - expression values in a cursor loop, 6-18
 - when needed, 2-13
- columns
 - accessing in triggers, 9-19
 - generating derived values with triggers, 9-41
 - listing in an UPDATE trigger, 9-5, 9-21
- COMMENT clause
 - using with transactions, 6-34
- comments
 - in PL/SQL, 2-8
 - links to examples, 13-27
 - restrictions, 2-9
 - syntax, 13-27
- COMMIT statement, 6-33
 - links to examples, 13-28
 - syntax, 13-28
- comparison operators, 6-6
- comparisons
 - of character values, 3-10
 - of expressions, 2-26
 - of null collections, 5-17
 - operators, 2-24
 - PL/SQL, 2-21
 - with NULLs, 2-30
- compiler parameter settings
 - ALL_PLSQL_OBJECT_SETTINGS view, 12-1
- compiler parameters
 - and REUSE SETTINGS clause, 12-3
 - PL/SQL, 12-1
- compiling
 - conditional, 1-13, 2-34
- composite types
 - collection and records, 5-1
- Compound triggers, 9-12
- concatenation operator, 2-25
 - treatment of nulls, 2-31
- conditional compilation, 1-13, 2-34
 - availability for previous Oracle database
 - releases, 2-34
 - control tokens, 2-34
 - examples, 2-40
 - inquiry directives, 2-35
 - limitations, 2-41

- NLS_LENGTH_SEMANTICS initialization
 - parameter, 2-36
- PLSQL_CCFLAGS initialization parameter, 2-36
- PLSQL_CODE_TYPE initialization
 - parameter, 2-36
- PLSQL_DEBUG initialization parameter, 2-36
- PLSQL_LINE flag, 2-36
- PLSQL_OPTIMIZE_LEVEL initialization
 - parameter, 2-36
- PLSQL_UNIT flag, 2-36
- PLSQL_WARNINGS initialization
 - parameter, 2-36
- restrictions, 2-41
- static constants, 2-38
- using PLSQL_CCFLAGS initialization
 - parameter, 2-39
- using static expressions with, 2-36
- using with DBMS_DB_VERSION, 2-39
- using with DBMS_PREPROCESSOR, 2-41
- conditional control, 4-2
- conditional predicates
 - trigger bodies, 9-17, 9-21
- conditional statement
 - guidelines, 4-6
- CONSTANT
 - declaration, 13-30
 - for declaring constants, 1-8, 2-10
- Constants
 - inlining, 8-19
- constants
 - declaring, 1-8, 2-9, 2-10
 - links to examples, 13-31
 - static, 2-38
 - syntax, 13-29
 - understanding PL/SQL, 1-5
- constraining tables, 9-24
- constraints
 - NOT NULL, 2-10
 - triggers and, 9-2, 9-34
- constructors
 - collection, 5-10
- context
 - transactions, 6-43
- CONTINUE statement
 - links to examples, 13-33
 - syntax, 13-33
- CONTINUE-WHEN statement, 1-12
- control structures
 - conditional, 4-2
 - overview of PL/SQL, 4-1
 - sequential, 4-16
 - understanding, 1-10
- conventions
 - PL/SQL naming, 2-14
- conversions
 - datatype, 3-26
- correlated subqueries, 6-20
- correlation names, 9-12
 - NEW, 9-19
 - OLD, 9-19
- REFERENCING option and, 9-20
 - when preceded by a colon, 9-19
- COUNT method
 - collections, 5-21, 13-24
- CREATE
 - with PROCEDURE statement, 1-13, 1-21
- CREATE FUNCTION statement, 1-13, 1-21
- CREATE PROCEDURE statement, 1-13, 1-21
- CREATE statement
 - packages, 1-14, 10-1
 - with FUNCTION, 1-13, 1-21
- CREATE TRIGGER statement, 9-3
 - REFERENCING option, 9-20
- CREATE_WRAPPED function
 - obfuscation, A-4
 - using, A-5
- creating
 - functions, 1-13, 1-21
 - packages, 1-14, 10-1
 - procedures, 1-13, 1-21
- cross-session PL/SQL function result cache, 8-29
- CURRENT OF clause
 - with UPDATE, 6-38
- current user
 - definer's rights subprograms, 8-19
 - invoker's rights subprograms, 8-19
 - subprograms, 8-19
- CURRVAL
 - pseudocolumn, 6-4
- cursor attributes
 - %BULK_EXCEPTIONS, 12-17
 - %BULK_ROWCOUNT, 12-16
 - %FOUND, 6-8, 6-13
 - %ISOPEN, 6-8, 6-14
 - %NOTFOUND, 6-8, 6-14
 - %ROWCOUNT, 6-8, 6-15
 - DBMS_SQL package and, 7-6
 - explicit, 6-13
 - implicit, 6-7
 - links to examples, 13-36
 - native dynamic SQL and, 7-2
 - syntax, 13-35
 - values after OPEN, FETCH, and CLOSE, 6-15
- cursor declarations
 - links to examples, 13-43
 - syntax, 13-41
- cursor expressions
 - examples, 6-32
 - REF CURSORS, 6-32
 - restrictions, 6-31
 - using, 6-31
- cursor FOR loops
 - passing parameters to, 6-21
- cursor subqueries
 - using, 6-31
- cursor variables, 6-22
 - advantages of, 6-22
 - as parameters to table functions, 12-35
 - avoiding errors with, 6-30
 - closing, 6-29

- declaring, 6-23
- defining, 6-23
- fetching from, 6-28
- links to examples, 13-39
- opening, 6-25
- passing as parameters, 6-24
- reducing network traffic, 6-29
- restrictions, 6-30
- syntax, 13-37
- using as a host variable, 6-27
- CURSOR_ALREADY_OPEN exception, 11-4
- cursors
 - advantages of using cursor variables, 6-22
 - attributes of explicit, 6-13
 - attributes of implicit, 6-7
 - closing explicit, 6-13
 - declaring explicit, 6-10
 - definition, 1-17
 - explicit, 1-17, 6-9
 - explicit FOR loops, 6-18
 - expressions, 6-31
 - fetching from, 6-11
 - guidelines for implicit, 6-9
 - implicit, 1-17
 - opening explicit, 6-10
 - packaged, 10-12
 - parameterized, 6-21
 - REF CURSOR variables, 6-22
 - RETURN clause, 10-12
 - scope rules for explicit, 6-10
 - SYS_REFCURSOR type, 12-35
 - variables, 6-22

D

- data abstraction
 - understanding PL/SQL, 1-17
- database character set, 2-1
- database events
 - attributes, 9-46
 - tracking, 9-43
- Database Resident Connection Pool, 10-11
- database triggers, 1-22
 - autonomous, 6-45
- datatypes
 - BFILE, 3-22
 - BLOB, 3-22
 - BOOLEAN, 3-15
 - CHAR, 3-8
 - CLOB, 3-22
 - DATE, 3-16
 - explicit conversion, 3-26
 - implicit conversion, 3-27
 - INTERVAL DAY TO SECOND, 3-19
 - INTERVAL YEAR TO MONTH, 3-19
 - LONG, 3-13
 - national character, 3-12
 - NCHAR, 3-12, 3-13
 - NCLOB, 3-22
 - NUMBER, 3-6

- PL/SQL
 - see PL/SQL datatypes
- RAW, 3-11
- RECORD, 5-1
- REF CURSOR, 6-22
- ROWID, 3-14
- TABLE, 5-7
- TIMESTAMP, 3-17
- TIMESTAMP WITH LOCAL TIME ZONE, 3-18
- TIMESTAMP WITH TIME ZONE, 3-18
- UROWID, 3-14
- VARRAY, 5-3, 5-7
- DATE datatype, 3-16
- datetime
 - arithmetic, 3-20
 - datatypes, 3-15
 - literals, 2-7
- DAY
 - datatype field, 3-15
- DB_ROLE_CHANGE system manager event, 9-50
- DBMS_ALERT package, 10-10
- DBMS_ASSERT package, 7-14
- DBMS_CONNECTION_CLASS package, 10-11
- DBMS_DB_VERSION package
 - using with conditional compilation, 2-39
- DBMS_DDL package
 - functions for hiding PL/SQL source code, A-1
 - limitations, A-2
 - obfuscating PL/SQL code, A-1
 - using, A-5
 - wrap functions, A-4
 - wrapping PL/SQL code, A-1
- DBMS_OUTPUT package
 - displaying output, 1-16
 - displaying output from PL/SQL, 10-10
 - using PUT_LINE to display output, 1-7
- DBMS_PIPE package, 10-11
- DBMS_PREPROCESSOR package
 - using with conditional compilation, 2-41
- DBMS_PROFILE package
 - gathering statistics for tuning, 12-10
- DBMS_SQL package, 7-6
 - upgrade to dynamic SQL, 12-24
- DBMS_SQL.TO_NUMBER function, 7-6
- DBMS_SQL.TO_REFCURSOR function, 7-6
- DBMS_TRACE package
 - tracing code for tuning, 12-10
- DBMS_WARNING package
 - controlling warning messages in PL/SQL, 11-20
- dbmsupbin.sql script
 - interpreted compilation, 12-29
- dbmsupgnv.sql script
 - for PL/SQL native compilation, 12-30
- deadlocks
 - how handled by PL/SQL, 6-36
- debugging
 - triggers, 9-28
- DEC
 - NUMBER subtype, 3-7
- DECIMAL

- NUMBER subtype, 3-7
- declarations
 - collection, 5-8
 - constants, 1-8, 2-10
 - cursor variables, 6-23
 - exceptions in PL/SQL, 11-6
 - explicit cursor, 6-10
 - PL/SQL functions, 1-8
 - PL/SQL procedures, 1-8
 - PL/SQL subprograms, 1-8
 - PL/SQL variables, 1-8
 - restrictions, 2-14
 - subprograms, 8-5
 - using %ROWTYPE, 2-12
 - using %TYPE attribute, 2-10
 - using DEFAULT, 2-10
 - using NOT NULL constraint, 2-10
 - variables, 1-5, 2-9
- declarative part
 - of PL/SQL block, 1-5
- DECLARE
 - start of declarative part of a PL/SQL block, 13-13
 - syntax, 13-13
- DECODE function
 - treatment of nulls, 2-31
- DEFAULT keyword
 - for assignments, 2-10
- DEFAULT option
 - FUNCTION, 13-77
 - RESTRICT_REFERENCES, 13-118
- default parameter values, 8-10
- default value
 - effect on %ROWTYPE declaration, 2-12
 - effect on %TYPE declaration, 2-11
- DEFINE
 - limitations of use with wrap utility, A-2
- definer's rights
 - privileges on subprograms, 8-17
- DELETE method
 - collections, 5-27, 13-24
- DELETE statement
 - column values and triggers, 9-19
 - links to examples, 13-46
 - syntax, 13-45
 - triggers for referential integrity, 9-36, 9-37
- delimiters, 2-2
- dense collections
 - arrays and nested tables, 5-3
- dependencies
 - in stored triggers, 9-27
 - schema objects
 - trigger management, 9-24
- DETERMINISTIC option
 - function syntax, 13-76
- dictionary_obj_owner event attribute, 9-47
- dictionary_obj_owner_list event attribute, 9-47
- dictionary_obj_type event attribute, 9-47
- digits of precision, 3-6
- disabled trigger
 - definition, 9-2

- disabling
 - triggers, 9-2
- displaying output
 - from PL/SQL, 1-7
 - setting SERVEROUTPUT, 1-16, 10-10
 - using DBMS_OUTPUT.PUT_LINE, 1-7
 - with DBMS_OUTPUT, 1-16
- DISTINCT row operator, 6-3, 6-7
- distributed databases
 - triggers and, 9-24
- dot notation, 1-9
 - for collection methods, 5-20
 - for global variables, 4-14
 - for package contents, 10-5
- DOUBLE PRECISION
 - NUMBER subtype, 3-7
- DROP TRIGGER statement, 9-28
- dropping
 - triggers, 9-28
- DUP_VAL_ON_INDEX exception, 11-5
- dynamic multiple-row queries, 7-4
- dynamic SQL, 7-1
 - DBMS_SQL package, 7-6
 - native, 7-2
 - switching between native dynamic SQL and DBMS_SQL package, 7-6
 - tuning, 12-24
- dynamic wrapping
 - DBMS_DDL package, A-1

E

- element types
 - collection, 5-7
- ELSE clause
 - using, 4-2
- ELSIF clause
 - using, 4-3
- enabled trigger
 - definition, 9-2
- enabling
 - triggers, 9-2
- END
 - end of a PL/SQL block, 13-13
 - syntax, 13-13
- END IF
 - end of IF statement, 4-2
- END LOOP
 - end of LOOP statement, 4-11
- error handling
 - in PL/SQL, 11-1
 - overview, 1-19
- error messages
 - maximum length, 11-15
- ERROR_CODE
 - BULK_EXCEPTIONS cursor attribute field, 12-17
 - using with SQLERRM, 12-17
- ERROR_INDEX
 - BULK_EXCEPTIONS cursor attribute field, 12-17
- evaluation

- short-circuit, 2-23
- event attribute functions, 9-46
- event publication, 9-44 to 9-45
 - triggering, 9-44
- events
 - attribute, 9-46
 - tracking, 9-43
- EXCEPTION
 - exception-handling part of a block, 13-13
 - syntax in PL/SQL block, 13-13
- exception definition
 - syntax, 13-48
- exception handlers
 - OTHERS handler, 11-2
 - overview, 1-19
 - using RAISE statement in, 11-12, 11-13
 - using SQLCODE function in, 11-14
 - using SQLERRM function in, 11-14
 - WHEN clause, 11-13
- EXCEPTION_INIT pragma
 - links to examples, 13-47
 - syntax, 13-47
 - using with RAISE_APPLICATION_ERROR, 11-8
 - with exceptions, 11-7
- exception-handling part
 - of PL/SQL block, 1-5
- exceptions
 - advantages of PL/SQL, 11-3
 - branching with GOTO, 11-14
 - catching unhandled in PL/SQL, 11-16
 - continuing after an exception is raised, 11-16
 - controlling warning messages, 11-19
 - declaring in PL/SQL, 11-6
 - definition, 13-48
 - during trigger execution, 9-21
 - handling in PL/SQL, 11-1
 - links to examples, 13-49
 - list of predefined in PL/SQL, 11-4
 - locator variables to identify exception locations, 11-18
 - OTHERS handler in PL/SQL, 11-13
 - PL/SQL compile-time warnings, 11-18
 - PL/SQL error condition, 11-1
 - PL/SQL warning messages, 11-19
 - predefined in PL/SQL, 11-4
 - propagation in PL/SQL, 11-10
 - raise_application_error procedure, 11-8
 - raised in a PL/SQL declaration, 11-14
 - raised in handlers, 11-14
 - raising in PL/SQL, 11-9
 - raising predefined explicitly, 11-9
 - raising with RAISE statement, 11-9
 - redeclaring predefined in PL/SQL, 11-9
 - reraising in PL/SQL, 11-12
 - retrying a transaction after, 11-17
 - scope rules in PL/SQL, 11-6
 - tips for handling PL/SQL errors, 11-16
 - user-defined in PL/SQL, 11-6
 - using EXCEPTION_INIT pragma, 11-7
 - using SQLCODE, 11-14
 - using SQLERRM, 11-14
 - using the DBMS_WARNING package, 11-20
 - using WHEN and OR, 11-13
 - WHEN clause, 11-13
- executable part
 - of PL/SQL block, 1-5
- EXECUTE IMMEDIATE statement, 7-2
 - links to examples, 13-53
 - syntax, 13-51
- EXECUTE privilege
 - subprograms, 8-21
- EXISTS method
 - collections, 5-21, 13-24
- EXIT statement
 - early exit of LOOP, 4-15
 - links to examples, 13-55
 - syntax, 13-55
 - using, 4-8, 4-9
- EXIT-WHEN statement, 1-12
 - using, 4-9, 4-10
- explicit cursors, 6-9
- explicit datatype conversion, 3-26
- explicit declarations
 - cursor FOR loop record, 6-18
- expressions
 - as default parameter values, 8-11
 - in cursors, 6-22
 - Boolean, 2-26
 - CASE, 2-28
 - examples, 13-66
 - PL/SQL, 2-21
 - static, 2-36
 - syntax, 13-57
- EXTEND method
 - collections, 5-25, 13-24
- external
 - references, 8-18
 - routines, 8-25
 - subprograms, 8-25

F

- FALSE value, 2-7
- features, new, xxiii
- FETCH statement
 - links to examples, 13-69
 - syntax, 13-67
 - using explicit cursors, 6-11
 - with cursor variable, 6-28
- fetching
 - across commits, 6-39
 - bulk, 12-20
- fields
 - of records, 5-1
- file I/O, 10-11
- FIRST method
 - collections, 5-22, 13-24
- FLOAT
 - NUMBER subtype, 3-7
- FOR EACH ROW clause, 9-11

- FOR loops
 - explicit cursors, 6-18
 - nested, 4-14
- FOR UPDATE clause, 6-10
 - when to use, 6-38
- FORALL statement
 - links to examples, 13-73
 - syntax, 13-71
 - using, 12-12
 - using to improve performance, 12-12
 - using with BULK COLLECT clause, 12-23
 - with rollbacks, 12-15
- FOR-LOOP statement
 - syntax, 13-95
 - using, 4-12
- formal parameters, 6-21
- forward
 - declarations of subprograms, 8-5
 - references, 2-14
- FOUND cursor attribute
 - explicit, 6-13
 - implicit, 6-8
- FUNCTION
 - with CREATE statement, 1-13, 1-21
- function declaration
 - syntax, 13-74
- function result cache, 8-29
- functions
 - creating, 1-13, 1-21
 - declaration, 13-74
 - in PL/SQL, 8-1
 - invoking, 8-3
 - links to examples, 13-78
 - pipelined, 12-31
 - RETURN statement, 8-5
 - SQL, in PL/SQL, 2-33
 - table, 12-31

G

- GOTO statement
 - branching into or out of exception handler, 11-14
 - label, 4-16
 - links to examples, 13-79
 - overview, 1-12
 - restrictions, 4-17
 - syntax, 13-79
 - using, 4-16
- grantee event attribute, 9-47
- GROUP BY clause, 6-3

H

- handlers
 - exception in PL/SQL, 11-2
- handling errors
 - PL/SQL, 11-1
- handling exceptions
 - PL/SQL, 11-1
 - raised in as PL/SQL declaration, 11-14

- raised in handler, 11-14
 - using OTHERS handler, 11-13
 - handling of nulls, 2-30
 - hash tables
 - simulating with associative arrays, 5-5
 - hiding PL/SQL code, A-1
 - host arrays
 - bulk binds, 12-23
 - HOUR
 - datatype field, 3-15
 - HTF package, 10-11
 - HTP package, 10-11
 - hypertext markup language (HTML), 10-11
 - hypertext transfer protocol (HTTP), 1-3
 - UTL_HTTP package, 10-11

I

- identifiers
 - forming, 2-3
 - maximum length, 2-4
 - quoted, 2-4
 - scope rules, 2-16
- IF statement, 4-2
 - ELSE clause, 4-2
 - links to examples, 13-81, 13-83
 - syntax, 13-80
 - using, 4-2
- IF-THEN statement
 - using, 4-2
- IF-THEN-ELSE statement
 - overview, 1-10
 - using, 4-2
- IF-THEN-ELSEIF statement
 - using, 4-3
- implicit cursors
 - attributes, 6-7
 - guidelines, 6-9
- implicit datatype conversion, 3-27
- implicit datatype conversions
 - performance, 12-7
- implicit declarations
 - FOR loop counter, 4-14
- IN comparison operator, 2-25
- IN OUT parameter mode
 - subprograms, 8-10
- IN parameter mode
 - subprograms, 8-9
- INDEX BY
 - collection definition, 13-20
- index-by tables
 - See associative arrays
- INDICES OF clause
 - FORALL, 13-71
 - with FORALL, 12-12
- infinite loops, 4-8
- initialization
 - collections, 5-10
 - package, 10-6
 - using DEFAULT, 2-10

- variable, 2-19
 - with NOT NULL constraint, 2-10
- initialization parameters
 - PL/SQL compilation, 12-1
- in-line LOB locators, 3-21
- INLINE pragma
 - syntax, 13-82
- Inlining constants, 8-19
- Inlining subprograms, 12-3
- INSERT statement
 - column values and triggers, 9-19
 - links to examples, 13-86
 - syntax, 13-85
 - with a record variable, 5-36
- instance_num event attribute, 9-47
- INSTEAD OF triggers, 9-7
 - on nested table view columns, 9-20
- INT
 - NUMBER subtype, 3-7
- INTEGER
 - NUMBER subtype, 3-7
- inter-language calls, 8-25
- interpreted compilation
 - dbmsupbin.sql script, 12-29
 - recompiling all PL/SQL modules, 12-29
- INTERSECT set operator, 6-7
- interval
 - arithmetic, 3-20
- INTERVAL DAY TO SECOND datatype, 3-19
- INTERVAL YEAR TO MONTH datatype, 3-19
- intervals
 - datatypes, 3-15
- INTO
 - SELECT INTO statement, 13-128
- INTO clause
 - with FETCH statement, 6-28
- INTO list
 - using with explicit cursors, 6-11
- INVALID_CURSOR exception, 11-5
- INVALID_NUMBER exception, 11-5
- invoker's rights
 - advantages, 8-18
 - privileges on subprograms, 8-17
- invoking
 - Java stored procedures, 8-25
 - procedures, 1-16
 - stored subprograms, 1-21
- IS A SET operator, 5-17
- IS EMPTY operator, 5-17
- IS NULL comparison operator, 2-24
 - expressions, 13-64
- is_alter_column event attribute, 9-47
- ISOLATION LEVEL parameter
 - READ COMMITTED, 13-134
 - SERIALIZABLE, 13-134
 - setting transactions, 13-134
- ISOPEN cursor attribute
 - explicit, 6-14
 - implicit, 6-8

J

- JAVA
 - use for invoking external subprograms, 8-26
- Java
 - call specs, 8-26
- Java stored procedures
 - invoking from PL/SQL, 8-25

K

- keywords
 - use in PL/SQL, 2-4
- keywords, PL/SQL, D-1
 - list of, D-2

L

- labels
 - block, 2-18
 - block structure, 13-14
 - exiting loops, 4-10
 - GOTO statement, 4-16
 - loops, 4-10
 - syntax, 13-14
- LANGUAGE
 - use for invoking external subprograms, 8-26
- language elements
 - of PL/SQL, 13-1
- large object (LOB) datatypes, 3-21
- LAST method
 - collections, 5-22, 13-24
- LEVEL
 - pseudocolumn, 6-5
- LEVEL parameter
 - with ISOLATION to set transactions, 13-134
- lexical units
 - PL/SQL, 2-1
- LIKE comparison operator, 2-25
 - expressions, 13-64
- LIMIT clause
 - FETCH, 13-68
 - using to limit rows for a Bulk FETCH operation, 12-21
- LIMIT method
 - collections, 5-22, 13-24
- limitations
 - bulk binding, 12-11
 - DBMS_DDL package, A-2
 - of PL/SQL programs, C-1
 - PL/SQL compiler, C-1
 - wrap utility, A-2
- limits
 - on PL/SQL programs, C-1
- literals
 - Boolean, 2-7
 - character, 2-6
 - datetime, 2-7
 - examples, 13-90
 - NCHAR string, 2-7
 - NUMBER datatype, 2-6

- numeric, 2-5
- numeric datatypes, 2-5
- string, 2-6
- syntax, 13-88
- types of PL/SQL, 2-5
- LOB (large object) datatypes, 3-21
- LOB datatypes
 - use in triggers, 9-19
- LOB locators, 3-21
- local subprograms, 1-21
- locator variables
 - used with exceptions, 11-18
- LOCK TABLE statement
 - examples, 13-92
 - locking a table, 6-39
 - syntax, 13-91
- locks
 - modes, 6-33
 - overriding, 6-37
 - transaction processing, 6-33
 - using FOR UPDATE clause, 6-38
- logical operators, 2-22
- logical rowids, 3-14
- LOGIN_DENIED exception, 11-5
- LONG datatype, 3-13
 - maximum length, 3-13
 - use in triggers, 9-24
- LOOP statement, 4-7
 - links to examples, 13-97
 - overview, 1-11
 - syntax, 13-93
 - using, 4-8
- loops
 - counters, 4-12
 - dynamic ranges, 4-13
 - exiting using labels, 4-10
 - implicit declaration of counter, 4-14
 - iteration, 4-13
 - labels, 4-10
 - reversing the counter, 4-12
 - scope of counter, 4-14

M

- maximum precision, 3-6
- maximum size
 - CHAR value, 3-8
 - identifier, 2-4
 - LONG value, 3-13
 - Oracle error message, 11-15
 - RAW value, 3-11
- MEMBER OF operator, 5-17
- membership test, 2-25
- memory
 - avoid excessive overhead, 12-8
- MERGE statement
 - syntax, 13-98
- Method 4, 7-6
- methods
 - collection, 5-20

- MINUS set operator, 6-7
- MINUTE
 - datatype field, 3-15
- modularity, 1-13
 - packages, 10-3
- MONTH
 - datatype field, 3-15
- multilevel collections
 - using, 5-19
- multi-line comments, 2-8
- multiple-row queries
 - dynamic, 7-4
- MULTISET EXCEPT operator, 5-13
- MULTISET INTERSECT operator, 5-13
- MULTISET UNION operator, 5-13
- mutating table
 - definition, 9-24
- mutating tables
 - trigger restrictions, 9-24

N

- NAME
 - for invoking external subprograms, 8-26
- NAME parameter
 - setting transactions, 13-135
 - transactions, 6-37
- name resolution, 2-15
 - differences between PL/SQL and SQL, B-4
 - global and local variables, B-1
 - inner capture in DML statements, B-5
 - overriding in subprograms, 8-21
 - qualified names and dot notation, B-2
 - qualifying references to attributes and methods, B-6
 - understanding, B-1
 - understanding capture, B-4
 - with synonyms, 8-21
- names
 - explicit cursor, 6-10
 - qualified, 2-14
 - savepoint, 6-35
 - variable, 2-15
- naming conventions
 - PL/SQL, 2-14
- national character datatypes, 3-12
- national character set, 2-1
- native compilation
 - dbmsupgnv.sql script, 12-30
 - dependencies, 12-28
 - how it works, 12-27
 - invalidation, 12-28
 - modifying databases for, 12-28
 - revalidation, 12-28
 - setting up databases, 12-28
 - utlrlp.sql script, 12-30
- native dynamic SQL, 7-2
- NATURAL
 - BINARY_INTEGER subtype, 3-3
- NATURALN

- BINARY_INTEGER subtype, 3-3
- NCHAR datatype, 3-12, 3-13
- NCLOB datatype, 3-22
- nested collections, 5-19
- nested cursors
 - using, 6-31
- nested tables
 - associative arrays and, 5-6
 - sets of values, 5-2
 - syntax, 13-19
 - transforming with operators, 5-13
 - understanding, 5-2
 - varrays and, 5-6
- nesting
 - block, 1-5
 - FOR loops, 4-14
 - record, 5-32
- NEW correlation name, 9-19
- new features, xxiii
- NEXT method
 - collections, 5-23, 13-25
- NEXTVAL
 - pseudocolumn, 6-4
- NLS_LENGTH_SEMANTICS initialization parameter
 - use with conditional compilation, 2-36
- NO COPY hint
 - FUNCTION, 13-76
- NO_DATA_FOUND exception, 11-5
- NOCOPY compiler hint
 - for tuning, 12-25
 - restrictions on, 12-26
- NOT logical operator
 - treatment of nulls, 2-31
- NOT NULL
 - declaration, 13-30
- NOT NULL constraint
 - effect on %ROWTYPE declaration, 2-12
 - effect on %TYPE declaration, 2-11
 - restriction on explicit cursors, 6-10
 - using in collection declaration, 5-10
 - using in variable declaration, 2-10
- NOT NULL option
 - record definition, 13-116
- NOT_LOGGED_ON exception, 11-5
- notation
 - positional and named, 8-7
- NOTFOUND cursor attribute
 - explicit, 6-14
 - implicit, 6-8
- NOWAIT parameter
 - using with FOR UPDATE, 6-38
- NVL function
 - treatment of nulls, 2-32
- null handling, 2-30
- NULL statement
 - links to examples, 13-99
 - syntax, 13-99
 - using, 4-18
- NULL value
 - dynamic SQL and, 7-3

- NUMBER datatype, 3-6
 - range of literals, 2-6
 - range of values, 3-6
- NUMERIC
 - NUMBER subtype, 3-7
- numeric literals, 2-5
 - PL/SQL datatypes, 2-5

O

- obfuscating
 - PL/SQL code, A-1
 - understanding, A-1
- obfuscation
 - CREATE_WRAPPED function, A-4
 - DBMS_DDL package, A-4
 - hiding PL/SQL code, A-1
 - limitations, A-2
 - recommendations, A-2
 - tips, A-2
 - understanding, A-1
 - using DBMS_DDL CREATE_WRAPPED function, A-5
 - viewing source, A-1, A-5
 - WRAP function, A-4
 - wrap utility, A-3
- object type declaration
 - syntax, 13-100
- object types
 - overview, 1-18
 - using with invoker's-rights subprograms, 8-23
- OBJECT_VALUE pseudocolumn, 9-21
- OLD correlation name, 9-19
- ONLY parameter
 - with READ to set transactions, 13-134
- on-the-fly wrapping
 - DBMS_DDL package, A-1
- OPEN statement
 - explicit cursors, 6-10
 - links to examples, 13-103
 - syntax, 13-102
- OPEN-FOR statement, 6-25
 - links to examples, 13-105
 - syntax, 13-104
- OPEN-FOR-USING statement
 - syntax, 13-104
- operators
 - comparison, 2-24
 - logical, 2-22
 - precedence, 2-21
 - relational, 2-24
- optimizing
 - PL/SQL programs, 12-3
- OR keyword
 - using with EXCEPTION, 11-13
- ora_dictionary_obj_owner event attribute, 9-47
- ora_dictionary_obj_owner_list event attribute, 9-47
- ora_dictionary_obj_type event attribute, 9-47
- ora_grantee event attribute, 9-47
- ora_instance_num event attribute, 9-47

- ora_is_alter_column event, 9-47
- ora_is_creating_nested_table event attribute, 9-47
- ora_is_drop_column event attribute, 9-47
- ora_is_servererror event attribute, 9-47
- ora_login_user event attribute, 9-47
- ora_privileges event attribute, 9-47
- ora_revokee event attribute, 9-47
- ora_server_error event attribute, 9-47
- ora_sysevent event attribute, 9-47
- ora_with_grant_option event attribute, 9-49
- order of evaluation, 2-21, 2-23
- OTHERS clause
 - exception handling, 13-48
- OTHERS exception handler, 11-2, 11-13
- OUT parameter mode
 - subprograms, 8-9
- out-of-line LOB locators, 3-21
- overloading
 - guidelines, 8-13
 - packaged subprograms, 10-9
 - restrictions, 8-14
 - subprogram names, 8-12

P

- PACKAGE
 - with SQL CREATE statement, 1-14, 10-1
- PACKAGE BODY
 - with SQL CREATE statement, 1-14, 10-1
- package declaration
 - syntax, 13-107
- packaged cursors, 10-12
- packaged subprograms, 1-21
- packages
 - advantages, 10-3
 - bodiless, 10-4
 - body, 10-1, 10-5, 13-108
 - call specification, 10-2
 - contents of, 10-2
 - creating, 1-14, 10-1
 - cursor specifications, 10-12
 - cursors, 10-12
 - declaration, 13-107
 - dot notation, 10-5
 - examples of features, 10-6
 - global variables, 10-9
 - guidelines for writing, 10-12
 - hidden declarations, 10-2
 - initializing, 10-6
 - invoking subprograms, 10-5
 - links to examples, 13-108
 - modularity, 10-3
 - overloading subprograms, 10-9
 - overview, 1-14
 - overview of Oracle supplied, 10-10
 - private and public objects, 10-9
 - product-specific, 10-10
 - product-specific for use with PL/SQL, 1-3
 - referencing, 10-5
 - restrictions on referencing, 10-5

- scope, 10-4
- specification, 10-1, 13-108
- specifications, 10-4
- STANDARD package, 10-9
- understanding, 10-1
- visibility of contents, 10-2
- PARALLEL_ENABLE option
 - FUNCTION, 13-76
- parameter passing
 - by reference, 8-27
 - by value, 8-27
- parameters
 - actual, 6-21
 - actual and formal, 8-6
 - aliasing, 8-27
 - cursor, 6-21
 - default values, 8-10
 - formal, 6-21
 - IN mode, 8-9
 - IN OUT mode, 8-10
 - modes, 8-9
 - OUT mode, 8-9
 - summary of modes, 8-10
- parentheses, 2-22
- parse tree, 9-26
- pattern matching, 2-25
- performance
 - avoid memory overhead, 12-8
 - avoiding problems, 12-4
- physical rowids, 3-14
- pipe, 10-11
- PIPE ROW statement
 - for returning rows incrementally, 12-33
- PIPELINED
 - function option, 12-32, 13-77
- pipelined functions
 - exception handling, 12-38
 - fetching from results of, 12-34
 - for querying a table, 12-31
 - overview, 12-31
 - passing data with cursor variables, 12-35
 - performing DML operations inside, 12-37
 - performing DML operations on, 12-38
 - returning results from, 12-33
 - transformation of data, 12-31
 - transformations, 12-32
 - writing, 12-32
- pipelines
 - between table functions, 12-34
 - returning results from table functions, 12-33
 - support collection types, 12-32
 - using table functions, 12-32
 - writing table functions, 12-32
- pipelining
 - definition, 12-31
- PLS_INTEGER datatype, 3-2
 - overflow condition, 3-3
- PL/SQL
 - advantages, 1-1
 - anonymous blocks, 1-5

- architecture, 1-20
- assigning Boolean values, 2-20
- assigning query result to variable, 2-20
- assigning values to variables, 2-19
- blocks
 - structure, 1-5
 - syntax, 13-8
- CASE expressions, 2-28
- character sets, 2-1
- collection types, 5-1
- collections
 - overview, 1-17
- comments, 2-8
- comparisons, 2-21
- compiler limitations, C-1
- compiler parameter settings, 12-1
- compiler parameters, 12-1
- compile-time warnings, 11-18
- conditional compilation, 1-13, 2-34
- constants, 1-5
- control structures, 1-10, 4-1
- creating Web applications and pages, 2-42
- data abstraction, 1-17
- declarations
 - constants, 2-9
- displaying output, 1-7, 10-10
- engine, 1-20
- environment, 10-9
- error handling
 - overview, 1-19
- errors, 11-1
- exceptions, 11-1
- expressions, 2-21
- functions, 8-1
- hiding or obfuscating source code, A-1
- input data, 1-16
- lexical units, 2-1
- limitations of programs, C-1
- limits on programs, C-1
- literals, 2-5
- logical operators, 2-22
- name resolution, B-1
- naming conventions, 2-14
- new features, xxiii
- output data, 1-16
- performance problems, 12-4
- portability, 1-3
- procedural aspects, 1-4
- procedures, 8-1
- profiling and tracing programs, 12-9
- querying data, 6-16
- records
 - overview, 1-18
- scope of identifiers, 2-16
- Server Pages (PSPs), 2-43
- statements, 13-1
- subprograms, 8-1
 - invoking, 1-21
- syntax of language elements, 13-1
- transaction processing, 6-32
- trigger bodies, 9-17, 9-19
- tuning code, 12-4
- tuning computation-intensive programs, 12-24
- tuning dynamic SQL programs, 12-24
- using NOCOPY for tuning, 12-25
- using transformation pipelines, 12-31
- variables, 1-5
- warning messages, 11-19
- Web applications, 2-42
- writing reusable code, 1-13
- PL/SQL datatypes, 3-1
 - predefined, 3-1
- PLSQL datatypes
 - numeric literals, 2-5
- PL/SQL function result cache, 8-29
- PLSQL_CCFLAGS initialization parameter
 - conditional compilation, 2-39
 - use with conditional compilation, 2-36
- PLSQL_CODE_TYPE initialization parameter
 - setting PL/SQL native compilation, 12-28
 - use with conditional compilation, 2-36
- PLSQL_DEBUG initialization parameter
 - use with conditional compilation, 2-36
- PLSQL_LINE flag
 - use with conditional compilation, 2-36
- PLSQL_OPTIMIZE_LEVEL initialization parameter, 12-3
 - optimizing PL/SQL programs, 12-3
 - use with conditional compilation, 2-36
- PLSQL_UNIT flag
 - use with conditional compilation, 2-36
- PLSQL_WARNINGS initialization parameter, 11-19
 - use with conditional compilation, 2-36
- pointers
 - REF CURSOR, 6-22
- portability, 1-3
- POSITIVE
 - BINARY_INTEGER subtype, 3-3
- POSITIVEN
 - BINARY_INTEGER subtype, 3-3
- PRAGMA
 - compiler directive with AUTONOMOUS_TRANSACTION, 13-6
 - compiler directive with EXCEPTION_INIT, 13-47
 - compiler directive with RESTRICT_REFERENCES, 13-118
 - compiler directive with SERIALLY_REUSABLE, 13-132
- pragmas
 - AUTONOMOUS_TRANSACTION, 6-41, 13-6
 - compiler directives, 11-7
 - EXCEPTION_INIT, 11-7, 13-47
 - INLINE, 13-82
 - RESTRICT_REFERENCES, 6-46, 8-27, 13-118
 - SERIALLY_REUSABLE, 13-132
- precedence, operator, 2-21
- precision of digits
 - specifying, 3-6
- predefined exceptions
 - raising explicitly, 11-9

- redeclaring, 11-9
- predefined PL/SQL datatypes, 3-1
- predicates, 6-6
- PRIOR method
 - collections, 5-23, 13-25
- PRIOR row operator, 6-5
- private objects
 - packages, 10-9
- privileges
 - creating triggers, 9-3
 - dropping triggers, 9-28
 - recompiling triggers, 9-27
- PROCEDURE
 - with CREATE statement, 1-13, 1-21
- procedure declaration
 - syntax, 13-110
- procedures
 - creating, 1-13, 1-21
 - declaration, 13-110
 - in PL/SQL, 8-1
 - invoked by triggers, 9-24
 - invoking, 1-16, 8-3
 - links to examples, 13-113
 - recompiling with ALTER PROCEDURE, 11-21
- productivity, 1-3
- Profiler API
 - gathering statistics for tuning, 12-10
- program units, 1-13
- PROGRAM_ERROR exception, 11-5
- propagation
 - exceptions in PL/SQL, 11-10
- pseudocolumns
 - CURRVAL, 6-4
 - LEVEL, 6-5
 - modifying views, 9-8
 - NEXTVAL, 6-4
 - ROWID, 6-5
 - ROWNUM, 6-5
 - SQL, 6-3
 - UROWID, 6-5
 - use in PL/SQL, 6-3
- public objects
 - packages, 10-9
- purity rules, 8-26
- PUT_LINE
 - displaying output with, 1-7

Q

- qualifiers
 - using subprogram names as, 2-16
 - when needed, 2-14, 2-18
- queries
 - multiple-row
 - dynamic, 7-4
- query work areas, 6-22
- querying data
 - BULK COLLECT clause, 6-16
 - cursor FOR loop, 6-17
 - implicit cursor FOR loop, 6-17

- looping through multiple rows, 6-17
- maintaining, 6-21
- performing complicated processing, 6-17
- SELECT INTO, 6-16
- using explicit cursors, 6-17
- using implicit cursors, 6-17
- with PL/SQL, 6-16
- work areas, 6-22
- quoted identifiers, 2-4

R

- RAISE statement
 - exceptions in PL/SQL, 11-9
 - links to examples, 13-114
 - syntax, 13-114
 - using in exception handler, 11-12, 11-13
- raise_application_error procedure
 - for raising PL/SQL exceptions, 11-8
- raising an exception
 - in PL/SQL, 11-9
- raising exceptions
 - triggers, 9-21
- range operator, 4-12
- RAW datatype, 3-11
 - maximum length, 3-11
- READ ONLY parameter
 - setting transactions, 13-134
 - transactions, 6-37
- READ WRITE parameter
 - setting transactions, 13-134
- readability, 2-2
 - with NULL statement, 4-18
- read-only transaction, 6-37
- REAL
 - NUMBER subtype, 3-7
- RECORD datatype, 5-1
- record definition
 - syntax, 13-115
- records
 - %ROWTYPE, 6-18
 - assigning values, 5-34
 - bulk-binding collections of, 5-38
 - comparing, 5-36
 - declaring, 5-31
 - defining, 5-31
 - definition, 1-9, 13-115
 - group of fields, 5-5
 - group of related data items, 5-1
 - implicit declaration, 6-18
 - inserting, 5-36
 - links to examples, 13-117
 - manipulating, 5-32
 - nesting, 5-32
 - overview, 1-18
 - passing as parameters, 5-32
 - restriction on assignments, 5-34
 - restrictions on inserts and updates of, 5-38
 - returning into, 5-37
 - ROWTYPE attribute, 5-5

- updating, 5-36
 - using as function return values, 5-32
- recursion
 - using with PL/SQL subprograms, 8-25
- REF CURSOR datatype, 6-22
 - cursor variables, 6-22
 - defining, 6-23
 - using with cursor subqueries, 6-32
- REF CURSOR variables
 - as parameters to table functions, 12-35
 - predefined SYS_REFCURSOR type, 12-35
- references
 - external, 8-18
 - resolving external, 8-19
- referencing
 - collections, 5-10
- referencing elements
 - allowed subscript ranges, 5-12
- REFERENCING option, 9-20
- referential integrity
 - self-referential constraints, 9-37
 - triggers and, 9-35 to 9-38
- regular expression functions
 - REGEXP_LIKE, 6-11
- relational operators, 2-24
- RELIES ON clause, 13-77
- remote access indicator, 2-14
- remote exception handling, 9-22
- REPEAT UNTIL structure
 - PL/SQL equivalent, 4-11
- REPLACE function
 - treatment of nulls, 2-32
- reraising an exception, 11-12
- reserved words
 - syntactic meaning in PL/SQL, 2-4
- reserved words, PL/SQL, D-1
 - list of, D-1
- resolution
 - name, 2-15
 - references to names, B-1
- RESTRICT_REFERENCES pragma, 8-27
 - links to examples, 13-119
 - syntax, 13-118
 - using with autonomous functions, 6-46
- restrictions
 - cursor expressions, 6-31
 - cursor variables, 6-30
 - overloading subprograms, 8-14
 - system triggers, 9-26
- result cache, 8-29
- result sets, 6-10
- RESULT_CACHE clause, 13-77
- RETURN clause
 - cursor, 10-12
 - cursor declaration, 13-38
 - FUNCTION, 13-77
- RETURN statement
 - functions, 8-5
 - links to examples, 13-120
 - syntax, 13-120
- return types
 - REF CURSOR, 6-23
- RETURNING clause
 - links to examples, 13-123
 - with a record variable, 5-37
- RETURNING INTO clause
 - syntax, 13-121
- returns
 - bulk, 12-22
- REUSE SETTINGS clause
 - with compiler parameters, 12-3
- REVERSE
 - with LOOP counter, 4-12
- REVERSE option
 - LOOP, 13-96
- RNDS option
 - RESTRICT_REFERENCES, 13-118
- RNPS option
 - RESTRICT_REFERENCES, 13-119
- ROLLBACK statement, 6-34
 - effect on savepoints, 6-35
 - links to examples, 13-124
 - syntax, 13-124
- rollbacks
 - implicit, 6-36
 - of FORALL statement, 12-15
- routines
 - external, 8-25
- row locks
 - with FOR UPDATE, 6-38
- row operators, 6-7
- row triggers
 - defining, 9-11
 - REFERENCING option, 9-20
 - timing, 9-6
 - UPDATE statements and, 9-5, 9-21
- ROWCOUNT cursor attribute
 - explicit, 6-15
 - implicit, 6-8
- ROWID
 - pseudocolumn, 6-5
- ROWID datatype, 3-14
- rowids, 3-14
- ROWIDTOCHAR function, 6-5
- ROWNUM
 - pseudocolumn, 6-5
- ROWTYPE attribute
 - declaring, 1-9
 - effect of default value, 2-12
 - effect of NOT NULL constraint, 2-12
 - inherited properties from columns, 13-125
 - links to examples, 13-126
 - records, 5-31
 - syntax, 13-125
 - using, 2-12
 - with SUBTYPE, 3-23
- ROWTYPE_MISMATCH exception, 11-5
- RPC (remote procedure call)
 - and exceptions, 11-10
- rules

- purity, 8-26
- run-time errors
 - PL/SQL, 11-1

S

- SAVE EXCEPTIONS clause
 - FORALL, 13-72
- SAVEPOINT statement, 6-35
 - links to examples, 13-127
 - syntax, 13-127
- savepoints
 - reusing names, 6-35
- scalar datatypes, 3-1
- scale
 - specifying, 3-7
- scientific notation, 2-5
- scope, 2-16
 - collection, 5-7
 - definition, 2-16
 - exceptions in PL/SQL, 11-6
 - explicit cursor, 6-10
 - explicit cursor parameter, 6-10
 - identifier, 2-16
 - loop counter, 4-14
 - package, 10-4
- searched CASE expression, 2-29
- searched CASE statement, 4-6
- SECOND
 - datatype field, 3-15
- security risks, 7-9
- SELECT INTO statement
 - links to examples, 13-131
 - returning one row, 6-16
 - syntax, 13-128
- selector, 2-28
- SELF_IS_NULL exception, 11-5
- semantics
 - string comparison, 3-10
- separators, 2-2
- sequences
 - CURRVAL and NEXTVAL, 6-4
- SERIALLY_REUSABLE pragma
 - examples, 13-132
 - syntax, 13-132
- Server Pages (PSPs)
 - PL/SQL, 2-43
- SERVEROUTPUT
 - displaying output from PL/SQL, 1-16
 - setting ON to display output, 10-10
- SET clause
 - UPDATE, 13-145
- set operators, 6-7
- SET TRANSACTION statement, 6-37
 - links to examples, 13-135
 - syntax, 13-134
- short-circuit evaluation, 2-23
- side effects, 8-9
 - controlling, 8-26
- SIGNTYPE
 - BINARY_INTEGER subtype, 3-3
 - simple CASE expression, 2-28
 - SIMPLE_INTEGER datatype, 3-3
 - single-line comments, 2-8
 - size limit
 - varrays, 5-7
- SMALLINT
 - NUMBER subtype, 3-7
- spaces
 - where allowed, 2-2
- sparse collections
 - nested tables and arrays, 5-3
- specification
 - call, 10-2
 - cursor, 10-12
 - package, 10-4
 - packages, 13-108
- SQL
 - comparisons operators, 6-6
 - data manipulation operations, 6-1
 - define variables and data manipulation statements, 6-3
 - DML operations, 6-1
 - dynamic, 7-1
 - exceptions raised by data manipulation statements, 6-3
 - no rows returned with data manipulation statements, 6-3
 - pseudocolumns, 6-3
 - static, 6-1
- SQL cursor
 - dynamic SQL and, 7-6
 - links to examples, 13-137
 - syntax, 13-136
- SQL functions in PL/SQL, 2-33
- SQL injection, 7-9
- SQL modification, 7-9
- SQL reserved words, D-1
- SQL statements
 - in trigger bodies, 9-19, 9-24
 - not allowed in triggers, 9-24
- SQLCODE function
 - links to examples, 13-139
 - syntax, 13-139
 - using with exception handlers, 11-14
- SQLERRM function
 - links to examples, 13-140
 - syntax, 13-140
 - using with BULK_EXCEPTIONS ERROR_CODE field, 12-17
 - using with exception handlers, 11-14
- standalone subprograms, 1-21
- STANDARD package
 - defining PL/SQL environment, 10-9
- START WITH clause, 6-5
- Statement injection, 7-11
- statement terminator, 13-14
- statement triggers
 - conditional code for statements, 9-21
 - row evaluation order, 9-6

- specifying SQL statement, 9-5
- timing, 9-6
- UPDATE statements and, 9-5, 9-21
- valid SQL statements, 9-24
- statements
 - assignment, 13-3
 - CASE, 13-16
 - CLOSE, 6-13, 6-29, 13-18
 - COMMIT, 13-28
 - CONTINUE, 13-33
 - DELETE, 13-45
 - EXECUTE IMMEDIATE, 13-51
 - EXIT, 13-55
 - FETCH, 6-11, 6-28, 13-67
 - FORALL, 12-12, 13-71
 - FOR-LOOP, 13-95
 - GOTO, 13-79
 - IF, 13-80
 - INSERT, 13-85
 - LOCK TABLE, 13-91
 - LOOP, 4-7, 13-93
 - MERGE, 13-98
 - NULL, 13-99
 - OPEN, 6-10, 13-102
 - OPEN-FOR, 6-25, 13-104
 - OPEN-FOR-USING, 13-104
 - PL/SQL, 13-1
 - RAISE, 13-114
 - RETURN, 13-120
 - ROLLBACK, 13-124
 - SAVEPOINT, 13-127
 - SELECT INTO, 13-128
 - SET TRANSACTION, 13-134
 - UPDATE, 13-144
 - WHILE-LOOP, 13-96
- static constants
 - conditional compilation, 2-38
- static expressions
 - boolean, 2-36
 - PLS_INTEGER, 2-36
 - use with conditional compilation, 2-36
 - VARCHAR2, 2-36
- static SQL, 6-1
- STEP clause
 - equivalent in PL/SQL, 4-13
- STORAGE_ERROR exception, 11-5
 - raised with recursion, 8-25
- store tables, 5-6
- stored subprograms
 - in Oracle database, 1-21
- string comparison semantics, 3-10
- string literals, 2-6
 - NCHAR, 2-7
- STRING subtype, 3-9
- Subprogram inlining, 12-3
- subprograms
 - actual and formal parameters, 8-6
 - advantages in PL/SQL, 8-4
 - AUTHID clause, 8-17, 8-18
 - controlling side effects, 8-26
 - current user during execution, 8-19
 - declaring nested, 8-5
 - declaring PL/SQL, 1-8
 - default parameter modes, 8-10
 - definer's rights, 8-17
 - EXECUTE privilege, 8-21
 - external references, 8-18
 - granting privileges on invoker's-rights, 8-21
 - guidelines for overloading, 8-13
 - how calls are resolved, 8-16
 - IN OUT parameter mode, 8-10
 - IN parameter mode, 8-9
 - in PL/SQL, 8-1
 - invoker's-rights, 8-17
 - invoking external, 8-25
 - invoking from SQL*Plus, 1-21, 1-22
 - invoking with parameters, 8-7
 - local, 1-21
 - mixed notation parameters, 8-7
 - named parameters, 8-7
 - OUT parameter mode, 8-9
 - overloading names, 8-12
 - overriding name resolution, 8-21
 - packaged, 1-21
 - parameter aliasing, 8-27
 - parameter modes, 8-9, 8-10
 - passing parameter by value, 8-27
 - passing parameters, 8-6
 - passing parameters by reference, 8-27
 - positional parameters, 8-7
 - recursive, 8-24
 - resolving external references, 8-19
 - restrictions on overloading, 8-14
 - standalone, 1-21
 - stored, 1-21
 - understanding PL/SQL, 8-1
 - using database links with invoker's-rights, 8-22
 - using recursion, 8-25
 - using triggers with invoker's-rights, 8-22
 - using views with invoker's-rights, 8-22
- subqueries
 - correlated, 6-20
 - using in PL/SQL, 6-19
- SUBSCRIPT_BEYOND_COUNT exception, 11-5
- SUBSCRIPT_OUTSIDE_LIMIT exception, 11-5
- SUBSTR function
 - using with SQLERRM, 11-15
- subtypes
 - CHARACTER, 3-9
 - compatibility, 3-24
 - constrained and unconstrained, 3-23
 - defining, 3-23
 - STRING, 3-9
 - using, 3-23
 - VARCHAR, 3-9
- synonyms
 - name resolution, 8-21
- syntax
 - BEGIN, 13-12
 - collection method, 13-23

- exception definition, 13-48
- FETCH statement, 13-67
- literal declaration, 13-88
- LOOP statement, 13-93
- NULL statement, 13-99
- package declaration, 13-107
- reading diagrams, 13-1
- WHILE-LOOP statement, 13-96

syntax of PL/SQL language elements, 13-1

SYS_INVALID_ROWID exception, 11-5

SYS_REFCURSOR type, 12-35

T

TABLE datatype, 5-7

table functions

- exception handling, 12-38
- fetching from results of, 12-34
- for querying, 12-31
- organizing multiple calls to, 12-34
- passing data with cursor variables, 12-35
- performing DML operations inside, 12-37
- performing DML operations on, 12-38
- pipelining data between, 12-34
- returning results from, 12-33
- setting up transformation pipelines, 12-31
- using transformation pipelines, 12-32
- writing transformation pipelines, 12-32

table, mutating

- definition, 9-24

tables

- constraining, 9-24
- mutating, 9-24

tabs, 2-2

terminator, statement, 2-2

THEN clause

- using, 4-2
- with IF statement, 4-2

TIMEOUT_ON_RESOURCE exception, 11-5

TIMESTAMP datatype, 3-17

TIMESTAMP WITH LOCAL TIME ZONE

- datatype, 3-18

TIMESTAMP WITH TIME ZONE datatype, 3-18

TIMEZONE_ABBR

- datatype field, 3-15

TIMEZONE_HOUR

- datatype field, 3-15

TIMEZONE_MINUTES

- datatype field, 3-15

TIMEZONE_REGION

- datatype field, 3-15

TO_NUMBER function, 7-6

TO_REFCURSOR function, 7-6

TOO_MANY_ROWS exception, 11-5

Trace API

- tracing code for tuning, 12-10

tracking database events, 9-43

transactions, 6-3

- autonomous in PL/SQL, 6-40
- committing, 6-33

- context, 6-43
- ending properly, 6-36
- processing in PL/SQL, 6-3, 6-32
- properties, 6-37
- read-only, 6-37
- restrictions, 6-37
- rolling back, 6-34
- savepoints, 6-35
- visibility, 6-43

trigger

- disabled

 - definition, 9-2

- enabled

 - definition, 9-2

triggering statement

- definition, 9-5

Triggers

- compound, 9-12

triggers

- accessing column values, 9-19
- AFTER, 9-6, 9-19, 9-31, 9-33
- as a stored PL/SQL subprogram, 1-22
- auditing with, 9-30, 9-31
- autonomous, 6-45
- BEFORE, 9-6, 9-19, 9-40, 9-41
- body, 9-17, 9-21, 9-24
- check constraints, 9-39, 9-40
- column list in UPDATE, 9-5, 9-21
- compiled, 9-26
- conditional predicates, 9-17, 9-21
- constraints and, 9-2, 9-34
- creating, 9-3, 9-23
- data access restrictions, 9-40
- debugging, 9-28
- designing, 9-2
- disabling, 9-2
- enabling, 9-2
- error conditions and exceptions, 9-21
- events, 9-5
- examples, 9-30 to 9-41
- FOR EACH ROW clause, 9-11
- generating derived column values, 9-41
- illegal SQL statements, 9-24
- INSTEAD OF triggers, 9-7
- listing information about, 9-29
- modifying, 9-28
- mutating tables and, 9-24
- naming, 9-4
- package variables and, 9-6
- privileges

 - to drop, 9-28

- procedures and, 9-24
- recompiling, 9-27
- REFERENCING option, 9-20
- referential integrity and, 9-35 to 9-38
- remote dependencies and, 9-24
- remote exceptions, 9-22
- restrictions, 9-12, 9-23
- row, 9-11
- row evaluation order, 9-6

- scan order, 9-6
- stored, 9-26
- use of LONG and LONG RAW datatypes, 9-24
- username reported in, 9-26
- WHEN clause, 9-12
- triggers on object tables, 9-21
- TRIM method
 - collections, 5-26, 13-25
- TRUE value, 2-7
- TRUST option
 - RESTRICT_REFERENCES, 13-119
- tuning
 - allocate large VARCHAR2 variables, 12-8
 - avoid memory overhead, 12-8
 - computation-intensive programs, 12-24
 - do not duplicate built-in functions, 12-7
 - dynamic SQL programs, 12-24
 - group related subprograms into a package, 12-8
 - guidelines for avoiding PL/SQL performance problems, 12-4
 - improve code to avoid compiler warnings, 12-9
 - make function calls efficient, 12-5
 - make loops efficient, 12-6
 - make SQL statements efficient, 12-5
 - optimizing PL/SQL programs, 12-3
 - pin packages in the shared memory pool, 12-9
 - PL/SQL code, 12-4
 - profiling and tracing, 12-9
 - reducing loop overhead, 12-11
 - reorder conditional tests to put least expensive first, 12-7
 - use BINARY_FLOAT or BINARY_DOUBLE for floating-point arithmetic, 12-8
 - use PLS_INTEGER for integer arithmetic, 12-7
 - using DBMS_PROFILE and DBMS_TRACE, 12-9
 - using FORALL, 12-12
 - using NOCOPY, 12-25
 - using transformation pipelines, 12-31
- TYPE attribute
 - declaring, 1-9
 - effect of default value, 2-11
 - effect of NOT NULL constraint, 2-11
 - inherited properties from column, 13-142
 - links to examples, 13-143
 - syntax, 13-142
 - using, 2-10
 - with SUBTYPE, 3-23
- TYPE definition
 - associative arrays, 5-7
 - collection, 5-7
 - collection types, 5-7
 - nested tables, 5-7
 - RECORD, 5-31
 - REF CURSOR, 6-23
 - VARRAY, 5-7
- catching, 11-16
- propagating, 11-10
- UNION ALL set operator, 6-7
- UNION set operator, 6-7
- universal rowids, 3-14
- updatable view
 - definition, 9-7
- UPDATE statement
 - column values and triggers, 9-19
 - links to examples, 13-146
 - syntax, 13-144
 - triggers and, 9-5, 9-21
 - triggers for referential integrity, 9-36, 9-37
 - with a record variable, 5-36
- URL (uniform resource locator), 10-11
- UROWID
 - pseudocolumn, 6-5
- UROWID datatype, 3-14
- USE ROLLBACK SEGMENT parameter
 - setting transactions, 13-135
- user-defined
 - exceptions in PL/SQL, 11-6
 - records, 5-1
- usernames
 - as reported in a trigger, 9-26
- USING clause
 - EXECUTE IMMEDIATE, 13-53
 - with OPEN FOR statement, 13-105
- UTL_FILE package, 10-11
- UTL_HTTP package, 10-11
- UTL_SMTP package, 10-12
- utltp.sql script
 - for PL/SQL native compilation, 12-30

V

- V\$RESERVED_WORDS view, D-1
- validation checks
 - avoiding SQL injection with, 7-13
- VALUE_ERROR exception, 11-5
- VALUES clause
 - INSERT, 13-86
- VALUES OF clause, 12-12
- FORALL, 13-71
- VARCHAR subtype, 3-9
- VARCHAR2 datatype
 - differences with CHAR, 3-9
- variables
 - assigning query result to, 2-20
 - assigning values, 1-6, 2-19
 - bind
 - see bind variables
 - declaring, 1-5, 2-9
 - declaring PL/SQL, 1-8
 - global, 10-9
 - initializing, 2-19
 - links to examples, 13-31
 - passing as IN OUT parameter, 1-7
 - REF CURSOR datatype, 6-22
 - syntax, 13-29

U

- underscores, 2-3
- unhandled exceptions

- understanding PL/SQL, 1-5
- variable-size arrays (varrays)
 - understanding, 5-2
- VARRAY datatype, 5-3, 5-7
- varrays
 - nested tables and, 5-6
 - size limit, 5-7
 - syntax, 13-19
 - TYPE definition, 5-7
 - understanding, 5-2
- views
 - containing expressions, 9-8
 - inherently modifiable, 9-8
 - modifiable, 9-8
 - pseudocolumns, 9-8
- visibility
 - of package contents, 10-2
 - scope and, 2-16
 - transaction, 6-43

W

- warning messages
 - controlling PL/SQL, 11-19
- Web applications
 - creating with PL/SQL, 2-42
- Web server pages
 - creating with PL/SQL, 2-42
- WHEN clause, 9-12
 - cannot contain PL/SQL expressions, 9-12
 - correlation names, 9-19
 - examples, 9-11, 9-35
 - EXCEPTION examples, 9-22, 9-35, 9-39, 9-40
 - exception handling, 13-48
 - exceptions, 11-13
 - using, 4-9, 4-10
- WHERE CURRENT OF clause
 - DELETE statement, 13-46
 - UPDATE, 13-146
- WHILE-LOOP statement
 - overview, 1-11
 - syntax, 13-96
 - using, 4-11
- wildcards, 2-25
- WNDS option
 - RESTRICT_REFERENCES, 13-119
- WNPS option
 - RESTRICT_REFERENCES, 13-119
- work areas
 - queries, 6-22
- WRAP function
 - obfuscation, A-4
- wrap utility
 - limitations, A-2
 - obfuscating PL/SQL code, A-1
 - obfuscation of PL/SQL code, A-3
 - running, A-4
- wrapped
 - PL/SQL source code, A-1
- wrapping

- DBMS_DDL package, A-4
 - dynamically, A-1
 - limitations, A-2
 - obfuscating code, A-1
 - on-the-fly, A-1
 - recommendations, A-2
 - tips, A-2
 - understanding, A-1
 - viewing source, A-1, A-5
 - with CREATE_WRAPPED, A-5
- WRITE parameter
 - with READ to set transactions, 13-134

Y

- YEAR
 - datatype field, 3-15

Z

- ZERO_DIVIDE exception, 11-6
- ZONE
 - part of TIMESTAMP datatype, 3-18